MODULE 3

Distributed memory programming with MPI

Recall that the world of parallel multiple instruction, multiple data, or MIMD, computers is, for the most part, divided into **distributed-memory** and **shared-memory** systems. From a programmer's point of view, a distributed-memory system consists of a collection of core-memory pairs connected by a network, and the memory associated with a core is directly accessible only to that core. (See Fig. 3.1.) On the other hand, from a programmer's point of view, a shared-memory system consists of a collection of cores connected to a globally accessible memory, in which each core can have access to any memory location. (See Fig. 3.2.) In this chapter, we're going to start looking at how to program distributed memory systems using **message-passing**.

Recall that in message-passing programs, a program running on one core-memory pair is usually called a **process**, and two processes can communicate by calling functions: one process calls a *send* function and the other calls a *receive* function. The implementation of message-passing that we'll be using is called **MPI**, which is an abbreviation of **Message-Passing Interface**. MPI is not a new programming language. It defines a *library* of functions that can be called from C and Fortran programs. We'll learn about some of MPI's different send and receive functions. We'll also

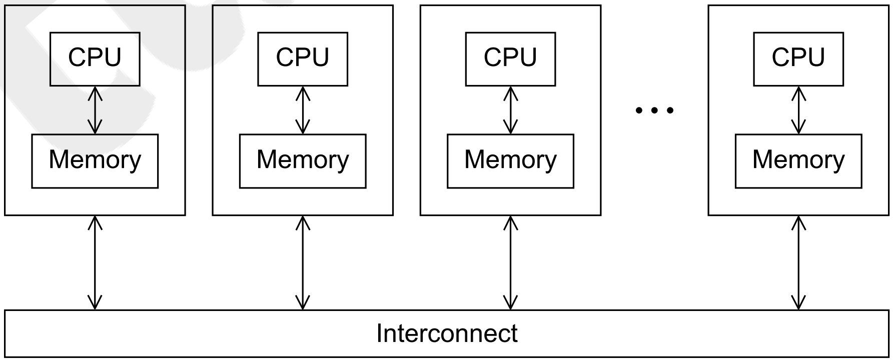

FIGURE 3.1

A distributed memory system.

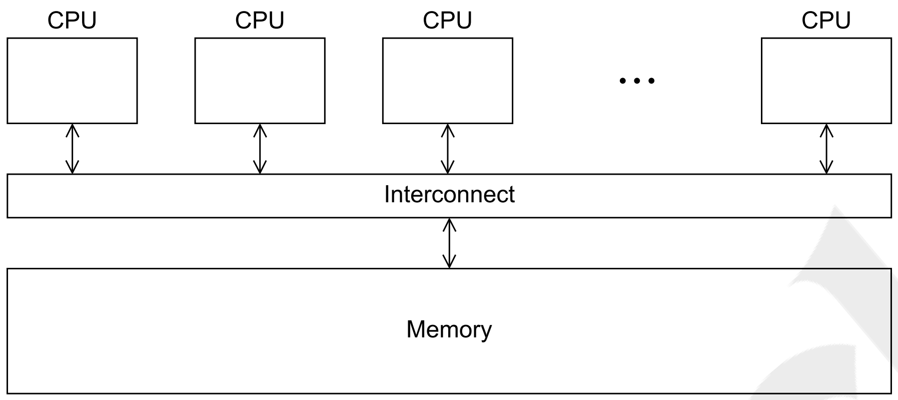

FIGURE 3.2

A shared memory system.

learn about some “global” communication functions that can involve more than two processes. These functions are called **collective** communications. In the process of learning about all of these MPI functions, we’ll also learn about some of the fundamental issues involved in writing message-passing programs—issues such as data partitioning and I/O in distributed-memory systems. We’ll also revisit the issue of parallel program performance.

3.1 Getting started

Perhaps the first program that many of us saw was some variant of the “hello, world” program in Kernighan and Ritchie’s classic text [32]:

```
#include <stdio.h>

int main(void) {
   printf("hello, world\n");

   return 0;
}
```

Let’s write a program similar to “hello, world” that makes some use of MPI. Instead of having each process simply print a message, we’ll designate one process to do the output, and the other processes will send it messages, which it will print.

In parallel programming, it’s common (one might say standard) for the processes to be identified by nonnegative integer *ranks*. So if there are p processes, the processes will have ranks 0, 1, 2, ..., $p-1$. For our parallel "hello, world," let's make process 0 the designated process, and the other processes will send it messages. See Program 3.1.

```c
#include <stdio.h>
#include <string.h>  /* For strlen             */
#include <mpi.h>     /* For MPI functions, etc */

const int MAX_STRING = 100;

int main(void) {
   char       greeting[MAX_STRING];
   int        comm_sz;  /* Number of processes */
   int        my_rank;  /* My process rank     */

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   if (my_rank != 0) {
      sprintf(greeting, "Greetings from process %d of %d!",
            my_rank, comm_sz);
      MPI_Send(greeting, strlen(greeting)+1, MPI_CHAR, 0, 0,
            MPI_COMM_WORLD);
   } else {
      printf("Greetings from process %d of %d!\n",
            my_rank, comm_sz);
      for (int q = 1; q < comm_sz; q++) {
         MPI_Recv(greeting, MAX_STRING, MPI_CHAR, q,
            0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         printf("%s\n", greeting);
      }
   }

   MPI_Finalize();
   return 0;
}  /* main */
```

Program 3.1: MPI program that prints greetings from the processes.

3.1.1 Compilation and execution

The details of compiling and running the program depend on your system, so you may need to check with a local expert. However, recall that when we need to be explicit, we'll assume that we're using a text editor to write the program source, and the command line to compile and run. Many systems use a command called `mpicc` for compilation[1]:

```
$ mpicc −g −Wall −o mpi_hello mpi_hello.c
```

Typically `mpicc` is a script that's a **wrapper** for the C compiler. A **wrapper script** is a script whose main purpose is to run some program. In this case, the program is the C compiler. However, the wrapper simplifies the running of the compiler by telling it where to find the necessary header files, and which libraries to link with the object file.

Many systems also support program startup with `mpiexec`:

```
$ mpiexec −n <number of processes> ./mpi_hello
```

So to run the program with one process, we'd type

```
$ mpiexec −n 1 ./mpi_hello
```

and to run the program with four processes, we'd type

```
$ mpiexec −n 4 ./mpi_hello
```

With one process, the program's output would be

```
Greetings from process 0 of 1!
```

and with four processes, the program's output would be

```
Greetings from process 0 of 4!
Greetings from process 1 of 4!
Greetings from process 2 of 4!
Greetings from process 3 of 4!
```

How do we get from invoking `mpiexec` to one or more lines of greetings? The `mpiexec` command tells the system to start <`number of processes`> instances of our <`mpi_hello`> program. It may also tell the system which core should run each instance of the program. After the processes are running, the MPI implementation takes care of making sure that the processes can communicate with each other.

3.1.2 MPI programs

Let's take a closer look at the program.

The first thing to observe is that this *is* a C program. For example, it includes the standard C header files `stdio.h` and `string.h`. It also has a `main` function, just like any other C program. However, there are many parts of the program that are new. Line 3 includes the `mpi.h` header file. This contains prototypes of MPI functions, macro definitions, type definitions, and so on; it contains all the definitions and declarations needed for compiling an MPI program.

[1] Recall that the dollar sign (`$`) is the shell prompt—so it shouldn't be typed in. Also recall that, for the sake of explicitness, we assume that we're using the Gnu C compiler, `gcc`, and we always use the options `-g, -Wall, and -o`. See Section 2.9 for further information.

The second thing to observe is that all of the identifiers defined by MPI start with the string `MPI_`. The first letter following the underscore is capitalized for function names and MPI-defined types. All of the letters in MPI-defined macros and constants are capitalized. So there's no question about what is defined by MPI and what's defined by the user program.

3.1.3 `MPI_Init` and `MPI_Finalize`

In Line 12, the call to `MPI_Init` tells the MPI system to do all of the necessary setup. For example, it might allocate storage for message buffers, and it might decide which process gets which rank. As a rule of thumb, no other MPI functions should be called before the program calls `MPI_Init`. Its syntax is

```
int MPI_Init(
      int*      argc_p   /* in/out */,
      char***   argv_p   /* in/out */);
```

The arguments, `argc_p` and `argv_p`, are pointers to the arguments to `main`, `argc` and `argv`. However, when our program doesn't use these arguments, we can just pass `NULL` for both. Like most MPI functions, `MPI_Init` returns an **int** error code. In most cases, we'll ignore the error codes, since checking them tends to clutter the code and make it more difficult to understand what it's doing.[2]

In Line 31, the call to `MPI_Finalize` tells the MPI system that we're done using MPI, and that any resources allocated for MPI can be freed. The syntax is quite simple:

```
 int MPI_Finalize(void);
```

In general, no MPI functions should be called after the call to `MPI_Finalize`.

Thus a typical MPI program has the following basic outline:

```
. . .
#include <mpi.h>
. . .
int main(int argc, char* argv[]) {
   . . .
   /* No MPI calls before this */
   MPI_Init(&argc, &argv);
   . . .
   MPI_Finalize();
   /* No MPI calls after this */
   . . .
   return 0;
}
```

[2] Of course, when we're debugging our programs, we may make extensive use of the MPI error codes.

However, we've already seen that it's not necessary to pass pointers to argc and argv to MPI_Init. It's also not necessary that the calls to MPI_Init and MPI_Finalize be in main.

3.1.4 Communicators, MPI_Comm_size, and MPI_Comm_rank

In MPI a **communicator** is a collection of processes that can send messages to each other. One of the purposes of MPI_Init is to define a communicator that consists of all of the processes started by the user when she started the program. This communicator is called MPI_COMM_WORLD. The function calls in Lines 13 and 14 are getting information about MPI_COMM_WORLD. Their syntax is

```
int MPI_Comm_size(
      MPI_Comm   comm        /* in   */,
      int*       comm_sz_p   /* out  */);

int MPI_Comm_rank(
      MPI_Comm   comm        /* in   */,
      int*       my_rank_p   /* out  */);
```

For both functions, the first argument is a communicator and has the special type defined by MPI for communicators, MPI_Comm. MPI_Comm_size returns in its second argument the number of processes in the communicator, MPI_Comm_rank returns in its second argument the calling process's rank in the communicator. We'll often use the variable comm_sz for the number of processes in MPI_COMM_WORLD, and the variable my_rank for the process rank.

3.1.5 SPMD programs

Notice that we compiled a single program—we didn't compile a different program for each process—and we did this in spite of the fact that process 0 is doing something fundamentally different from the other processes: it's receiving a series of messages and printing them, while each of the other processes is creating and sending a message. This is quite common in parallel programming. In fact, *most* MPI programs are written in this way. That is, a single program is written so that different processes carry out different actions, and this can be achieved by simply having the processes branch on the basis of their process rank. Recall that this approach to parallel programming is called **single program, multiple data** or **SPMD**. The **if** –**else** statement in Lines 16–29 makes our program SPMD.

Also notice that our program will, in principle, run with any number of processes. We saw a little while ago that it can be run with one process or four processes, but if our system has sufficient resources, we could also run it with 1000 or even 100,000 processes. Although MPI doesn't require that programs have this property, it's almost always the case that we try to write programs that will run with any number of processes, because we usually don't know in advance the exact resources available to us. For example, we might have a 20-core system available today, but tomorrow we might have access to a 500-core system.

3.1.6 Communication

In Lines 17–18, each process, other than process 0, creates a message it will send to process 0. (The function `sprintf` is very similar to `printf`, except that instead of writing to `stdout`, it writes to a string.) Lines 19–20 actually send the message to process 0. Process 0, on the other hand, simply prints its message using `printf`, and then uses a **for** loop to receive and print the messages sent by processes $1, 2, \ldots,$ `comm_sz` $-$ 1. Lines 25–26 receive the message sent by process q, for $q = 1, 2, \ldots,$ `comm_sz` $-$ 1.

3.1.7 `MPI_Send`

The sends executed by processes $1, 2, \ldots,$ `comm_sz` $-$ 1 are fairly complex, so let's take a closer look at them. Each of the sends is carried out by a call to `MPI_Send`, whose syntax is

```
int MPI_Send(
      void*          msg_buf_p      /* in */,
      int            msg_size       /* in */,
      MPI_Datatype   msg_type       /* in */,
      int            dest           /* in */,
      int            tag            /* in */,
      MPI_Comm       communicator   /* in */);
```

The first three arguments, `msg_buf_p`, `msg_size`, and `msg_type`, determine the contents of the message. The remaining arguments, `dest`, `tag`, and `communicator`, determine the destination of the message.

The first argument, `msg_buf_p`, is a pointer to the block of memory containing the contents of the message. In our program, this is just the string containing the message, `greeting`. (Remember that in C an array, such as a string, is a pointer.) The second and third arguments, `msg_size` and `msg_type`, determine the amount of data to be sent. In our program, the `msg_size` argument is the number of characters in the message, plus one character for the `'\0'` character that terminates C strings. The `msg_type` argument is `MPI_CHAR`. These two arguments together tell the system that the message contains `strlen(greeting)+1` **chars**.

Since C types (**int**, **char**, etc.) can't be passed as arguments to functions, MPI defines a special type, `MPI_Datatype`, that is used for the `msg_type` argument. MPI also defines a number of constant values for this type. The ones we'll use (and a few others) are listed in Table 3.1.

Notice that the size of the string `greeting` is not the same as the size of the message specified by the arguments `msg_size` and `msg_type`. For example, when we run the program with four processes, the length of each of the messages is 31 characters, while we've allocated storage for 100 characters in `greetings`. Of course, the size of the message sent should be less than or equal to the amount of storage in the buffer—in our case the string `greeting`.

The fourth argument, `dest`, specifies the rank of the process that should receive the message. The fifth argument, `tag`, is a nonnegative **int**. It can be used to distinguish

Table 3.1 Some predefined MPI datatypes.

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

messages that are otherwise identical. For example, suppose process 1 is sending **float**s to process 0. Some of the **float**s should be printed, while others should be used in a computation. Then the first four arguments to `MPI_Send` provide no information regarding which **float**s should be printed and which should be used in a computation. So process 1 can use (say) a tag of 0 for the messages that should be printed and a tag of 1 for the messages that should be used in a computation.

The final argument to `MPI_Send` is a communicator. All MPI functions that involve communication have a communicator argument. One of the most important purposes of communicators is to specify communication universes; recall that a communicator is a collection of processes that can send messages to each other. Conversely, a message sent by a process using one communicator cannot be received by a process that's using a different communicator. Since MPI provides functions for creating new communicators, this feature can be used in complex programs to ensure that messages aren't "accidentally received" in the wrong place.

An example will clarify this. Suppose we're studying global climate change, and we've been lucky enough to find two libraries of functions, one for modeling the earth's atmosphere and one for modeling the earth's oceans. Of course, both libraries use MPI. These models were built independently, so they don't communicate with each other, but they do communicate internally. It's our job to write the interface code. One problem we need to solve is ensuring that the messages sent by one library won't be accidentally received by the other. We might be able to work out some scheme with tags: the atmosphere library gets tags $0, 1, \ldots, n-1$, and the ocean library gets tags $n, n+1, \ldots, n+m$. Then each library can use the given range to figure out which tag it should use for which message. However, a much simpler solution is provided by communicators: we simply pass one communicator to the atmosphere library functions and a different communicator to the ocean library functions.

3.1.8 MPI_Recv

The first six arguments to MPI_Recv correspond to the first six arguments of MPI_Send:

```
int MPI_Recv(
      void*          msg_buf_p      /* out */,
      int            buf_size       /* in  */,
      MPI_Datatype   buf_type       /* in  */,
      int            source         /* in  */,
      int            tag            /* in  */,
      MPI_Comm       communicator   /* in  */,
      MPI_Status*    status_p       /* out */);
```

Thus the first three arguments specify the memory available for receiving the message: msg_buf_p points to the block of memory, buf_size determines the number of objects that can be stored in the block, and buf_type indicates the type of the objects. The next three arguments identify the message. The source argument specifies the process from which the message should be received. The tag argument should match the tag argument of the message being sent, and the communicator argument must match the communicator used by the sending process. We'll talk about the status_p argument shortly. In many cases, it won't be used by the calling function, and, as in our "greetings" program, the special MPI constant MPI_STATUS_IGNORE can be passed.

3.1.9 Message matching

Suppose process q calls MPI_Send with

```
MPI_Send(send_buf_p, send_buf_sz, send_type, dest, send_tag,
      send_comm);
```

Also suppose that process r calls MPI_Recv with

```
MPI_Recv(recv_buf_p, recv_buf_sz, recv_type, src, recv_tag,
      recv_comm, &status);
```

Then the message sent by q with the above call to MPI_Send can be received by r with the call to MPI_Recv if

- recv_comm = send_comm,
- recv_tag = send_tag,
- dest = r, and
- src = q.

These conditions aren't quite enough for the message to be *successfully* received, however. The parameters specified by the first three pairs of arguments, send_buf_p/recv_buf_p, send_buf_sz/recv_buf_sz, and send_type/recv_type, must specify compatible buffers. For detailed rules, see the MPI-3 specification [40]. Most of the time, the following rule will suffice:

- If recv_type = send_type and recv_buf_sz $\geq$ send_buf_sz, then the message sent by q can be successfully received by r.

Of course, it can happen that one process is receiving messages from multiple processes, *and* the receiving process doesn't know the order in which the other processes will send the messages. For example, suppose process 0 is doling out work to processes $1, 2, \ldots,$ `comm_sz` $- 1$, and processes $1, 2, \ldots,$ `comm_sz` $- 1$, send their results back to process 0 when they finish the work. If the work assigned to each process takes an unpredictable amount of time, then 0 has no way of knowing the order in which the processes will finish. If process 0 simply receives the results in process rank order—first the results from process 1, then the results from process 2, and so on—and if (say) process `comm_sz`-1 finishes first, it *could* happen that process `comm_sz`-1 could sit and wait for the other processes to finish. To avoid this problem MPI provides a special constant `MPI_ANY_SOURCE` that can be passed to `MPI_Recv`. Then, if process 0 executes the following code, it can receive the results in the order in which the processes finish:

```
for (i = 1; i < comm_sz; i++) {
   MPI_Recv(result, result_sz, result_type, MPI_ANY_SOURCE,
         result_tag, comm, MPI_STATUS_IGNORE);
   Process_result(result);
}
```

Similarly, it's possible that one process can be receiving multiple messages with different tags from another process, and the receiving process doesn't know the order in which the messages will be sent. For this circumstance, MPI provides the special constant `MPI_ANY_TAG` that can be passed to the `tag` argument of `MPI_Recv`.

A couple of points should be stressed in connection with these "wildcard" arguments:

1. Only a receiver can use a wildcard argument. Senders must specify a process rank and a nonnegative tag. So MPI uses a "push" communication mechanism rather than a "pull" mechanism.
2. There is no wildcard for communicator arguments; both senders and receivers must always specify communicators.

3.1.10 The `status_p` argument

If you think about these rules for a minute, you'll notice that a receiver can receive a message without knowing

1. the amount of data in the message,
2. the sender of the message, or
3. the tag of the message.

So how can the receiver find out these values? Recall that the last argument to `MPI_Recv` has type `MPI_Status*`. The MPI type `MPI_Status` is a struct with at least the three members `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR`. Suppose our program contains the definition

```
    MPI_Status status;
```

Then after a call to `MPI_Recv`, in which `&status` is passed as the last argument, we can determine the sender and tags by examining the two members:

```
    status.MPI_SOURCE
    status.MPI_TAG
```

The amount of data that's been received isn't stored in a field that's directly accessible to the application program. However, it can be retrieved with a call to `MPI_Get_count`. For example, suppose that in our call to `MPI_Recv`, the type of the receive buffer is `recv_type` and, once again, we passed in `&status`. Then the call

```
MPI_Get_count(&status, recv_type, &count)
```

will return the number of elements received in the `count` argument. In general, the syntax of `MPI_Get_count` is

```
int MPI_Get_count(
        MPI_Status*   status_p  /* in  */,
        MPI_Datatype  type      /* in  */,
        int*          count_p   /* out */);
```

Note that the count isn't directly accessible as a member of the `MPI_Status` variable, simply because it depends on the type of the received data, and, consequently, determining it would probably require a calculation (e.g., (number of bytes received)/(bytes per object)). And if this information isn't needed, we shouldn't waste a calculation determining it.

3.1.11 Semantics of `MPI_Send` and `MPI_Recv`

What exactly happens when we send a message from one process to another? Many of the details depend on the particular system, but we can make a few generalizations. The sending process will assemble the message. For example, it will add the "envelope" information to the actual data being transmitted—the destination process rank, the sending process rank, the tag, the communicator, and some information on the size of the message. Once the message has been assembled, recall from Chapter 2 that there are essentially two possibilities: the sending process can **buffer** the message or it can **block**. If it buffers the message, the MPI system will place the message (data and envelope) into its own internal storage, and the call to `MPI_Send` will return.

Alternatively, if the system blocks, it will wait until it can begin transmitting the message, and the call to `MPI_Send` may not return immediately. Thus if we use `MPI_Send`, when the function returns, we don't actually know whether the message has been transmitted. We only know that the storage we used for the message, the send buffer, is available for reuse by our program. If we need to know that the message has been transmitted, or if we need for our call to `MPI_Send` to return immediately—regardless of whether the message has been sent—MPI provides alternative functions for sending. We'll learn about one of these alternative functions later.

The exact behavior of MPI_Send is determined by the MPI implementation. However, typical implementations have a default "cutoff" message size. If the size of a message is less than the cutoff, it will be buffered. If the size of the message is greater than the cutoff, MPI_Send will block.

Unlike MPI_Send, MPI_Recv always blocks until a matching message has been received. So when a call to MPI_Recv returns, we know that there is a message stored in the receive buffer (unless there's been an error). There is an alternate method for receiving a message, in which the system checks whether a matching message is available and returns, regardless of whether there is one.

MPI requires that messages be *non-overtaking*. This means that if process q sends two messages to process r, then the first message sent by q must be available to r before the second message. However, there is no restriction on the arrival of messages sent from different processes. That is, if q and t both send messages to r, then even if q sends its message before t sends its message, there is no requirement that q's message become available to r before t's message. This is essentially because MPI can't impose performance on a network. For example, if q happens to be running on a machine on Mars, while r and t are both running on the same machine in San Francisco, and if q sends its message a nanosecond before t sends its message, it would be extremely unreasonable to require that q's message arrive before t's.

3.1.12 Some potential pitfalls

Note that the semantics of MPI_Recv suggests a potential pitfall in MPI programming: If a process tries to receive a message and there's no matching send, then the process will block forever. That is, the process will **hang**. So when we design our programs, we need to be sure that every receive has a matching send. Perhaps even more important, we need to be very careful when we're coding that there are no inadvertent mistakes in our calls to MPI_Send and MPI_Recv. For example, if the tags don't match, or if the rank of the destination process is the same as the rank of the source process, the receive won't match the send, and either a process will hang, or, perhaps worse, the receive may match *another* send.

Similarly, if a call to MPI_Send blocks and there's no matching receive, then the sending process can hang. If, on the other hand, a call to MPI_Send is buffered and there's no matching receive, then the message will be lost.

3.2 The trapezoidal rule in MPI

Printing messages from processes is all well and good, but we're probably not taking the trouble to learn to write MPI programs just to print messages. So let's take a look at a somewhat more useful program—let's write a program that implements the trapezoidal rule for numerical integration.

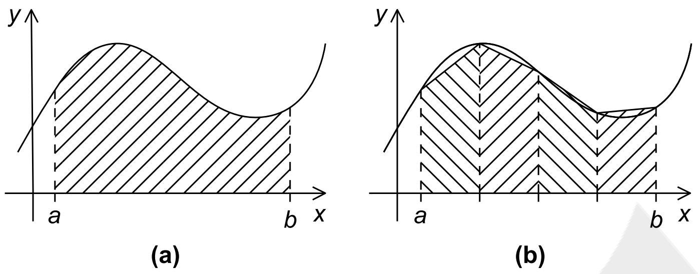

FIGURE 3.3

The trapezoidal rule: (a) area to be estimated and (b) approximate area using trapezoids.

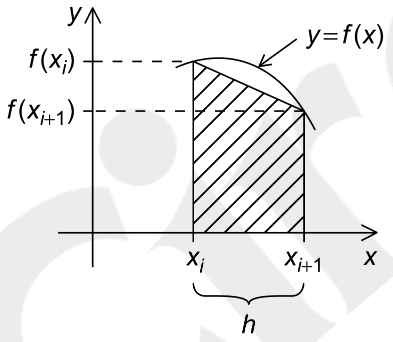

FIGURE 3.4

One trapezoid.

3.2.1 The trapezoidal rule

Recall that we can use the trapezoidal rule to approximate the area between the graph of a function, $y = f(x)$, two vertical lines, and the x-axis. (See Fig. 3.3.) The basic idea is to divide the interval on the x-axis into n equal subintervals. Then we approximate the area lying between the graph and each subinterval by a trapezoid, whose base is the subinterval, whose vertical sides are the vertical lines through the endpoints of the subinterval, and whose fourth side is the secant line joining the points where the vertical lines cross the graph. (See Fig. 3.4.) If the endpoints of the subinterval are x_i and x_{i+1}, then the length of the subinterval is $h = x_{i+1} - x_i$. Also, if the lengths of the two vertical segments are $f(x_i)$ and $f(x_{i+1})$, then the area of the trapezoid is

$$\text{Area of one trapezoid } = \frac{h}{2}[f(x_i) + f(x_{i+1})].$$

Since we chose the n subintervals so that they would all have the same length, we also know that if the vertical lines bounding the region are $x = a$ and $x = b$, then

$$h = \frac{b-a}{n}.$$

Thus if we call the leftmost endpoint x_0, and the rightmost endpoint x_n, we have that

$$x_0 = a,\ x_1 = a+h,\ x_2 = a+2h,\ \ldots,\ x_{n-1} = a+(n-1)h,\ x_n = b,$$

and the sum of the areas of the trapezoids—our approximation to the total area—is

$$\text{Sum of trapezoid areas} = h[f(x_0)/2 + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) + f(x_n)/2].$$

Thus, pseudocode for a serial program might look something like this:

```
/* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```

3.2.2 Parallelizing the trapezoidal rule

It is not the most attractive word, but, as we noted in Chapter 1, people who write parallel programs do use the verb "parallelize" to describe the process of converting a serial program or algorithm into a parallel program.

Recall that we can design a parallel program by using four basic steps:

1. Partition the problem solution into tasks.
2. Identify communication channels between the tasks.
3. Aggregate tasks into composite tasks.
4. Map composite tasks to cores.

In the partitioning phase, we usually try to identify as many tasks as possible. For the trapezoidal rule, we might identify two types of tasks: one type is finding the area of a single trapezoid, and the other is computing the sum of these areas. Then the communication channels will join each of the tasks of the first type to the single task of the second type. (See Fig. 3.5.)

So how can we aggregate the tasks and map them to the cores? Our intuition tells us that the more trapezoids we use, the more accurate our estimate will be. That is, we should use many trapezoids, and we will use many more trapezoids than cores. Thus we need to aggregate the computation of the areas of the trapezoids into groups. A natural way to do this is to split the interval $[a, b]$ up into `comm_sz` subintervals. If `comm_sz` evenly divides n, the number of trapezoids, we can simply apply the trapezoidal rule with $n/$`comm_sz` trapezoids to each of the `comm_sz` subintervals. To finish, we can have one of the processes, say process 0, add the estimates.

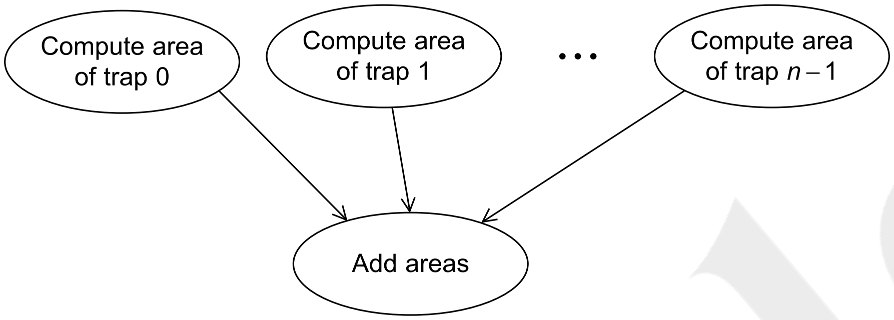

FIGURE 3.5

Tasks and communications for the trapezoidal rule.

Let's make the simplifying assumption that `comm_sz` evenly divides n. Then pseudocode for the program might look something like the following:

```
Get a, b, n;
h = (b−a)/n;
local_n = n/comm_sz;
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
local_integral = Trap(local_a, local_b, local_n, h);
if (my_rank != 0)
   Send local_integral to process 0;
else /* my_rank == 0 */
   total_integral = local_integral;
   for (proc = 1; proc < comm_sz; proc++) {
      Receive local_integral from proc;
      total_integral += local_integral;
   }
}
if (my_rank == 0)
   print result;
```

Let's defer, for the moment, the issue of input and just "hardwire" the values for a, b, and n. When we do this, we get the MPI program shown in Program 3.2. The `Trap` function is just an implementation of the serial trapezoidal rule. See Program 3.3.

Notice that in our choice of identifiers, we try to differentiate between *local* and *global* variables. **Local** variables are variables whose contents are significant only on the process that's using them. Some examples from the trapezoidal rule program are `local_a`, `local_b`, and `local_n`. Variables whose contents are significant to all the processes are sometimes called **global** variables. Some examples from the trapezoidal rule are `a`, `b`, and `n`. Note that this usage is different from the usage you learned in your introductory programming class, where local variables are private to a single function and global variables are accessible to all the functions. However, no confusion should arise, since the context should make the meaning clear.

```
int main(void) {
   int my_rank, comm_sz, n = 1024, local_n;
   double a = 0.0, b = 3.0, h, local_a, local_b;
   double local_int, total_int;
   int source;

   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

   h = (b-a)/n;            /* h is the same for all processes */
   local_n = n/comm_sz;    /* So is the number of trapezoids  */

   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   local_int = Trap(local_a, local_b, local_n, h);

   if (my_rank != 0) {
      MPI_Send(&local_int, 1, MPI_DOUBLE, 0, 0,
            MPI_COMM_WORLD);
   } else {
      total_int = local_int;
      for (source = 1; source < comm_sz; source++) {
         MPI_Recv(&local_int, 1, MPI_DOUBLE, source, 0,
               MPI_COMM_WORLD, MPI_STATUS_IGNORE);
         total_int += local_int;
      }
   }

   if (my_rank == 0) {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("of the integral from %f to %f = %.15e\n",
          a, b, total_int);
   }
   MPI_Finalize();
   return 0;
} /*  main  */
```

Program 3.2: First version of MPI trapezoidal rule.

3.3 Dealing with I/O

Of course, the current version of the parallel trapezoidal rule has a serious deficiency: it will only compute the integral over the interval $[0, 3]$ using 1024 trapezoids. We can edit the code and recompile, but this is quite a bit of work compared to simply typing in three new numbers. We need to address the problem of getting input from

```
double Trap(
      double left_endpt  /* in */,
      double right_endpt /* in */,
      int    trap_count  /* in */,
      double base_len    /* in */) {
   double estimate, x;
   int i;

   estimate = (f(left_endpt) + f(right_endpt))/2.0;
   for (i = 1; i <= trap_count-1; i++) {
      x = left_endpt + i*base_len;
      estimate += f(x);
   }
   estimate = estimate*base_len;

   return estimate;
} /*  Trap  */
```

Program 3.3: `Trap` function in MPI trapezoidal rule.

the user. While we're talking about input to parallel programs, it might be a good idea to also take a look at output. We discussed these two issues in Chapter 2. So if you remember the discussion of nondeterminism and output, you can skip ahead to Section 3.3.2.

3.3.1 Output

In both the "greetings" program and the trapezoidal rule program, we've assumed that process 0 can write to `stdout`, i.e., its calls to `printf` behave as we might expect. Although the MPI standard doesn't specify which processes have access to which I/O devices, virtually all MPI implementations allow *all* the processes in `MPI_COMM_WORLD` full access to `stdout` and `stderr`. So most MPI implementations allow all processes to execute `printf` and `fprintf(stderr, ...)`.

However, most MPI implementations don't provide any automatic scheduling of access to these devices. That is, if multiple processes are attempting to write to, say, `stdout`, the order in which the processes' output appears will be unpredictable. Indeed, it can even happen that the output of one process will be interrupted by the output of another process.

For example, suppose we try to run an MPI program in which each process simply prints a message. (See Program 3.4.) On our cluster, if we run the program with five processes, it often produces the "expected" output:

```
Proc 0 of 5 > Does anyone have a toothpick?
Proc 1 of 5 > Does anyone have a toothpick?
Proc 2 of 5 > Does anyone have a toothpick?
```

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

int main(void) {
   int my_rank, comm_sz;

   MPI_Init(NULL, NULL);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   printf("Proc %d of %d > Does anyone have a toothpick?\n",
         my_rank, comm_sz);

   MPI_Finalize();
   return 0;
}  /* main */
```

Program 3.4: Each process just prints a message.

```
Proc 3 of 5 > Does anyone have a toothpick?
Proc 4 of 5 > Does anyone have a toothpick?
```

However, when we run it with six processes, the order of the output lines is unpredictable:

```
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
```

or

```
Proc 0 of 6 > Does anyone have a toothpick?
Proc 1 of 6 > Does anyone have a toothpick?
Proc 2 of 6 > Does anyone have a toothpick?
Proc 4 of 6 > Does anyone have a toothpick?
Proc 3 of 6 > Does anyone have a toothpick?
Proc 5 of 6 > Does anyone have a toothpick?
```

The reason this happens is that the MPI processes are “competing” for access to the shared output device, `stdout`, and it’s impossible to predict the order in which the processes’ output will be queued up. Such a competition results in **nondeterminism**. That is, the actual output will vary from one run to the next.

In any case, if we don’t want output from different processes to appear in a random order, it’s up to us to modify our program accordingly. For example, we can have each process other than 0 send its output to process 0, and process 0 can print the output in process rank order. This is exactly what we did in the "greetings" program.

3.3.2 Input

Unlike output, most MPI implementations only allow process 0 in MPI_COMM_WORLD access to stdin. This makes sense: If multiple processes have access to stdin, which process should get which parts of the input data? Should process 0 get the first line? process 1 get the second? Or should process 0 get the first character?

So, to write MPI programs that can use scanf, we need to branch on process rank, with process 0 reading in the data, and then sending it to the other processes. For example, we might write the Get_input function shown in Program 3.5 for our parallel trapezoidal rule program. In this function, process 0 simply reads in the values

```
void Get_input(
      int      my_rank   /* in  */,
      int      comm_sz   /* in  */,
      double*  a_p       /* out */,
      double*  b_p       /* out */,
      int*     n_p       /* out */) {
   int dest;

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
      for (dest = 1; dest < comm_sz; dest++) {
         MPI_Send(a_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(b_p, 1, MPI_DOUBLE, dest, 0, MPI_COMM_WORLD);
         MPI_Send(n_p, 1, MPI_INT, dest, 0, MPI_COMM_WORLD);
      }
   } else { /* my_rank != 0 */
      MPI_Recv(a_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
      MPI_Recv(b_p, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
      MPI_Recv(n_p, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
   }
}  /* Get_input */
```

Program 3.5: A function for reading user input.

for a, b, and n, and sends all three values to each process. So this function uses the same basic communication structure as the "greetings" program, except that now process 0 is sending to each process, while the other processes are receiving.

To use this function, we can simply insert a call to it inside our main function, being careful to put it after we've initialized `my_rank` and `comm_sz`:

```
. . .
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);

Get_input(my_rank, comm_sz, &a, &b, &n);

h = (b−a)/n;
. . .
```

3.4 Collective communication

If we pause for a moment and think about our trapezoidal rule program, we can find several things that we might be able to improve on. One of the most obvious is the "global sum" after each process has computed its part of the integral. If we hire eight workers to, say, build a house, we might feel that we weren't getting our money's worth if seven of the workers told the first what to do, and then the seven collected their pay and went home. But this is very similar to what we're doing in our global sum. Each process with rank greater than 0 is "telling process 0 what to do" and then quitting. That is, each process with rank greater than 0 is, in effect, saying "add this number into the total." Process 0 is doing nearly all the work in computing the global sum, while the other processes are doing almost nothing. Sometimes it does happen that this is the best we can do in a parallel program, but if we imagine that we have eight students, each of whom has a number and we want to find the sum of all eight numbers, we can certainly come up with a more equitable distribution of the work than having seven of the eight give their numbers to one of the students and having the first do the addition.

3.4.1 Tree-structured communication

As we already saw in Chapter 1, we might use a "binary tree structure," like that illustrated in Fig. 3.6. In this diagram, initially students or processes 1, 3, 5, and 7 send their values to processes 0, 2, 4, and 6, respectively. Then processes 0, 2, 4, and 6 add the received values to their original values, and the process is repeated twice:

1. **a.** Processes 2 and 6 send their new values to processes 0 and 4, respectively.
   b. Processes 0 and 4 add the received values into their new values.
2. **a.** Process 4 sends its newest value to process 0.
   b. Process 0 adds the received value to its newest value.

This solution may not seem ideal, since half the processes (1, 3, 5, and 7) are doing the same amount of work that they did in the original scheme. However, if

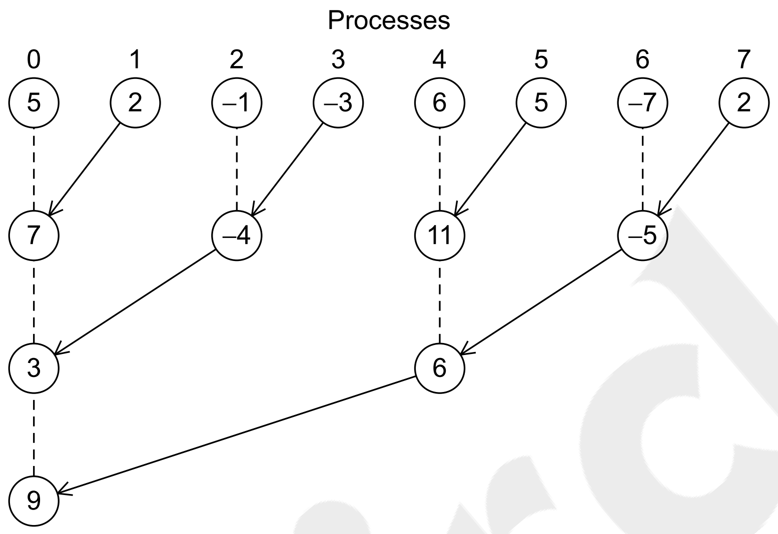

FIGURE 3.6

A tree-structured global sum.

you think about it, the original scheme required `comm_sz` $-\ 1 =$ seven receives and seven adds by process 0, while the new scheme only requires three, and all the other processes do no more than two receives and adds. Furthermore, the new scheme has the property that a lot of the work is done concurrently by different processes. For example, in the first phase, the receives and adds by processes 0, 2, 4, and 6 can all take place simultaneously. So, if the processes start at roughly the same time, the total time required to compute the global sum will be the time required by process 0, i.e., three receives and three additions. So we've reduced the overall time by more than 50%. Furthermore, if we use more processes, we can do even better. For example, if `comm_sz` $= 1024$, then the original scheme requires process 0 to do 1023 receives and additions, while it can be shown (Exercise 3.5) that the new scheme requires process 0 to do only 10 receives and additions. This improves the original scheme by more than a factor of 100!

You may be thinking to yourself, this is all well and good, but coding this tree-structured global sum looks like it would take a quite a bit of work, and you'd be right. (See Programming Assignment 3.3.) In fact, the problem may be even harder. For example, it's perfectly feasible to construct a tree-structured global sum that uses different "process-pairings." For example, we might pair 0 and 4, 1 and 5, 2 and 6, and 3 and 7 in the first phase. Then we could pair 0 and 2, and 1 and 3 in the second, and 0 and 1 in the final. (See Fig. 3.7.) Of course, there are many other possibilities. How can we decide which is the best? Do we need to code each alternative and evaluate their performance? If we do, is it possible that one method works best for "small"

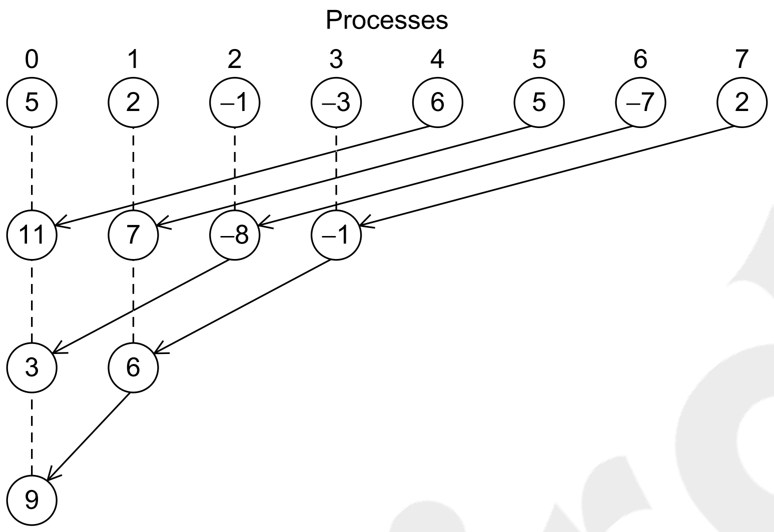

FIGURE 3.7

An alternative tree-structured global sum.

trees, while another works best for “large” trees? Even worse, one approach might work best on system A, while another might work best on system B.

3.4.2 `MPI_Reduce`

With virtually limitless possibilities, it’s unreasonable to expect each MPI programmer to write an optimal global-sum function, so MPI specifically protects programmers against this trap of endless optimization by requiring that MPI implementations include implementations of global sums. This places the burden of optimization on the developer of the MPI implementation, rather than the application developer. The assumption here is that the developer of the MPI implementation should know enough about both the hardware and the system software so that she can make better decisions about implementation details.

Now a “global-sum function” will obviously require communication. However, unlike the `MPI_Send`-`MPI_Recv` pair, the global-sum function may involve more than two processes. In fact, in our trapezoidal rule program it will involve all the processes in `MPI_COMM_WORLD`. In MPI parlance, communication functions that involve all the processes in a communicator are called **collective communications**. To distinguish between collective communications and functions, such as `MPI_Send` and `MPI_Recv`, `MPI_Send` and `MPI_Recv` are often called **point-to-point** communications.

In fact, global-sum is just a special case of an entire class of collective communications. For example, it might happen that instead of finding the sum of a collection of `comm_sz` numbers distributed among the processes, we want to find the maximum or the minimum or the product, or any one of many other possibilities. So MPI general-

Table 3.2 Predefined Reduction Operators in MPI.

Operation Value	Meaning
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical and
MPI_BAND	Bitwise and
MPI_LOR	Logical or
MPI_BOR	Bitwise or
MPI_LXOR	Logical exclusive or
MPI_BXOR	Bitwise exclusive or
MPI_MAXLOC	Maximum and location of maximum
MPI_MINLOC	Minimum and location of minimum

ized the global-sum function so that any one of these possibilities can be implemented with a single function:

```
int MPI_Reduce(
      void*          input_data_p   /* in   */,
      void*          output_data_p  /* out  */,
      int            count          /* in   */,
      MPI_Datatype   datatype       /* in   */,
      MPI_Op         operator       /* in   */,
      int            dest_process   /* in   */,
      MPI_Comm       comm           /* in   */);
```

The key to the generalization is the fifth argument, operator. It has type MPI_Op, which is a predefined MPI type—like MPI_Datatype and MPI_Comm. There are a number of predefined values in this type—see Table 3.2. It's also possible to define your own operators—for details, see the MPI-3 Standard [40].

The operator we want is MPI_SUM. Using this value for the operator argument, we can replace the code in Lines 18–28 of Program 3.2 with the single function call

```
MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM,
      0, MPI_COMM_WORLD);
```

One point worth noting is that by using a count argument greater than 1, MPI_Reduce can operate on arrays instead of scalars. So the following code could be used to add a collection of N-dimensional vectors, one per process:

```
double local_x[N], sum[N];
. . .
MPI_Reduce(local_x, sum, N, MPI_DOUBLE, MPI_SUM, 0,
      MPI_COMM_WORLD);
```

Table 3.3 Multiple Calls to `MPI_Reduce`.

Time	Process 0	Process 1	Process 2
0	`a = 1; c = 2`	`a = 1; c = 2`	`a = 1; c = 2`
1	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`
2	`MPI_Reduce(&c, &d, ...)`	`MPI_Reduce(&a, &b, ...)`	`MPI_Reduce(&c, &d, ...)`

3.4.3 Collective vs. point-to-point communications

It's important to remember that collective communications differ in several ways from point-to-point communications:

1. *All* the processes in the communicator must call the same collective function. For example, a program that attempts to match a call to `MPI_Reduce` on one process with a call to `MPI_Recv` on another process is erroneous, and, in all likelihood, the program will hang or crash.
2. The arguments passed by each process to an MPI collective communication must be "compatible." For example, if one process passes in 0 as the `dest_process` and another passes in 1, then the outcome of a call to `MPI_Reduce` is erroneous, and, once again, the program is likely to hang or crash.
3. The `output_data_p` argument is only used on `dest_process`. However, all of the processes still need to pass in an actual argument corresponding to `output_data_p`, even if it's just `NULL`.
4. Point-to-point communications are matched on the basis of tags and communicators. Collective communications don't use tags. So they're matched solely on the basis of the communicator and the *order* in which they're called. As an example, consider the calls to `MPI_Reduce` shown in Table 3.3. Suppose that each process calls `MPI_Reduce` with operator `MPI_SUM`, and destination process 0. At first glance, it might seem that after the two calls to `MPI_Reduce`, the value of `b` will be 3, and the value of `d` will be 6. However, the names of the memory locations are irrelevant to the matching of the calls to `MPI_Reduce`. The *order* of the calls will determine the matching, so the value stored in `b` will be $1 + 2 + 1 = 4$, and the value stored in `d` will be $2 + 1 + 2 = 5$.

A final caveat: it might be tempting to call `MPI_Reduce` using the same buffer for both input and output. For example, if we wanted to form the global sum of `x` on each process and store the result in `x` on process 0, we might try calling

```
MPI_Reduce(&x, &x, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
```

However, this call is illegal in MPI. So its result will be unpredictable: it might produce an incorrect result, it might cause the program to crash; it might even produce a correct result. It's illegal, because it involves **aliasing** of an output argument. Two arguments are aliased if they refer to the same block of memory, and MPI prohibits aliasing of arguments if one of them is an output or input/output argument. This is because the MPI Forum wanted to make the Fortran and C versions of MPI as similar as possible, and Fortran prohibits aliasing. In some instances MPI provides an alternative construction that effectively avoids this restriction. See Subsection 7.1.9 for an example.

3.4.4 MPI_Allreduce

In our trapezoidal rule program, we just print the result. So it's perfectly natural for only one process to get the result of the global sum. However, it's not difficult to imagine a situation in which *all* of the processes need the result of a global sum to complete some larger computation. In this situation, we encounter some of the same problems we encountered with our original global sum. For example, if we use a tree to compute a global sum, we might "reverse" the branches to distribute the global sum (see Fig. 3.8). Alternatively, we might have the processes *exchange* partial results instead of using one-way communications. Such a communication pattern is sometimes called a *butterfly*. (See Fig. 3.9.) Once again, we don't want to have to decide on which structure to use, or how to code it for optimal performance. Fortunately, MPI provides a variant of MPI_Reduce that will store the result on all the processes in the communicator:

```
int MPI_Allreduce(
      void*         input_data_p    /* in  */,
      void*         output_data_p   /* out */,
      int           count           /* in  */,
      MPI_Datatype  datatype        /* in  */,
      MPI_Op        operator        /* in  */,
      MPI_Comm      comm            /* in  */);
```

The argument list is identical to that for MPI_Reduce, except that there is no dest_process since all the processes should get the result.

3.4.5 Broadcast

If we can improve the performance of the global sum in our trapezoidal rule program by replacing a loop of receives on process 0 with a tree structured communication, we ought to be able to do something similar with the distribution of the input data. In fact, if we simply "reverse" the communications in the tree-structured global sum in Fig. 3.6, we obtain the tree-structured communication shown in Fig. 3.10, and we can use this structure to distribute the input data. A collective communication in which data belonging to a single process is sent to all of the processes in the communicator is called a **broadcast**, and you've probably guessed that MPI provides a broadcast function:

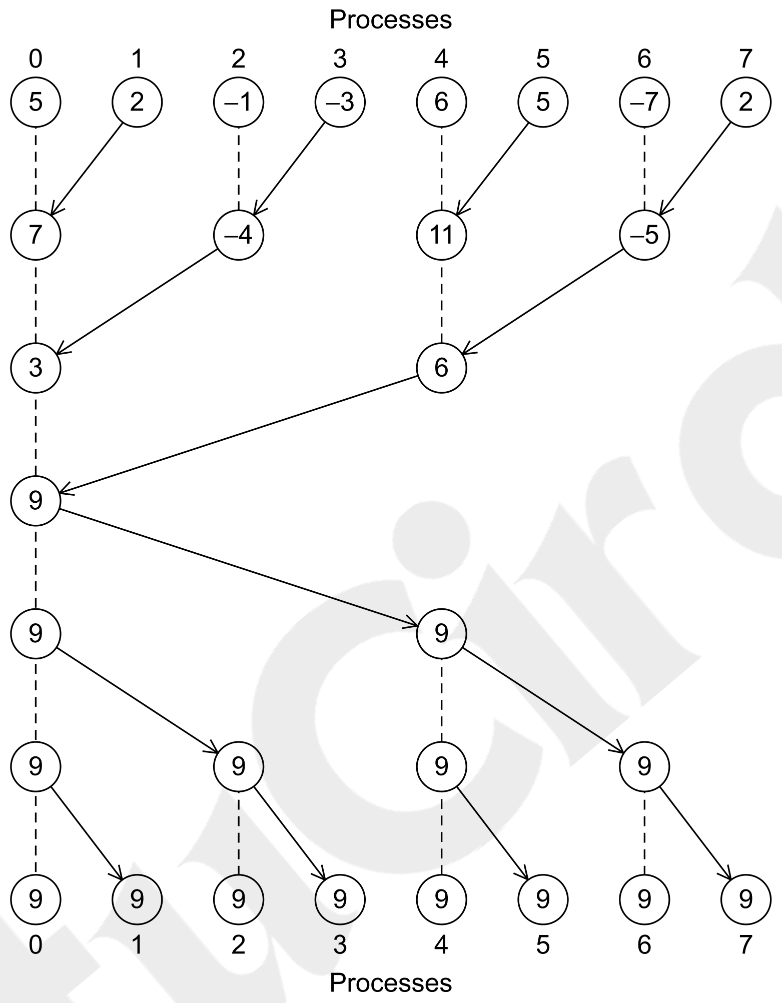

FIGURE 3.8

A global sum followed by distribution of the result.

```
int MPI_Bcast(
          void*          data_p       /* in/out */,
          int            count        /* in     */,
          MPI_Datatype   datatype     /* in     */,
          int            source_proc  /* in     */,
          MPI_Comm       comm         /* in     */);
```

The process with rank `source_proc` sends the contents of the memory referenced by `data_p` to all the processes in the communicator `comm`.

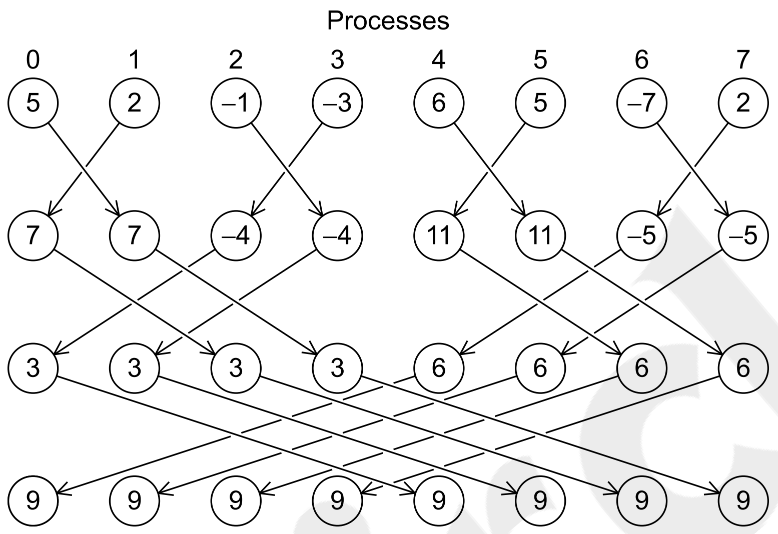

FIGURE 3.9

A butterfly-structured global sum.

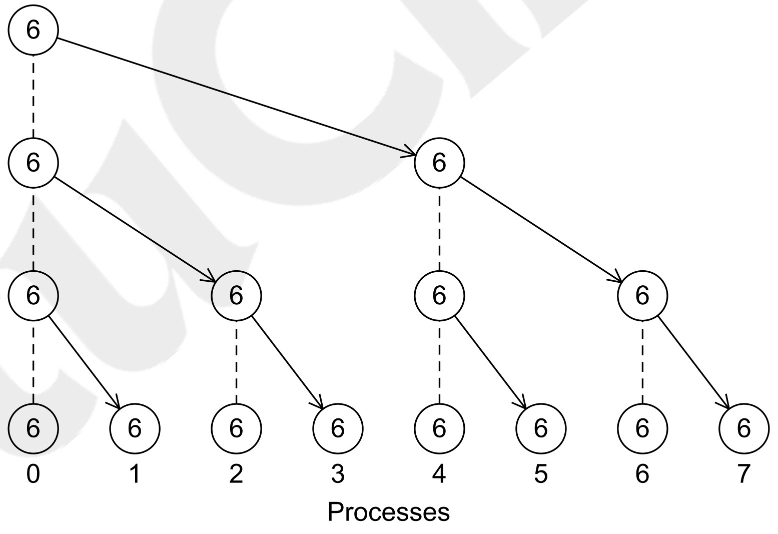

FIGURE 3.10

A tree-structured broadcast.

Program 3.6 shows how to modify the `Get_input` function shown in Program 3.5 so that it uses `MPI_Bcast`, instead of `MPI_Send` and `MPI_Recv`.

```
void Get_input(
      int      my_rank  /* in  */,
      int      comm_sz  /* in  */,
      double*  a_p      /* out */,
      double*  b_p      /* out */,
      int*     n_p      /* out */) {

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(b_p, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Bcast(n_p, 1, MPI_INT, 0, MPI_COMM_WORLD);
}  /* Get_input */
```

Program 3.6: A version of `Get_input` that uses `MPI_Bcast`.

Recall that in serial programs an in/out argument is one whose value is both used and changed by the function. For `MPI_Bcast`, however, the `data_p` argument is an input argument on the process with rank `source_proc` and an output argument on the other processes. Thus when an argument to a collective communication is labeled in/out, it's possible that it's an input argument on some processes and an output argument on other processes.

3.4.6 Data distributions

Suppose we want to write a function that computes a vector sum:

$$\begin{aligned}\mathbf{x}+\mathbf{y} &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1})\\ &= (x_0+y_0, x_1+y_1, \ldots, x_{n-1}+y_{n-1})\\ &= (z_0, z_1, \ldots, z_{n-1})\\ &= \mathbf{z}\end{aligned}$$

If we implement the vectors as arrays of, say, **double**s, we could implement serial vector addition with the code shown in Program 3.7.

How could we implement this using MPI? The work consists of adding the individual components of the vectors, so we might specify that the tasks are just the additions of corresponding components. Then there is no communication between the tasks, and the problem of parallelizing vector addition boils down to aggregating the tasks and assigning them to the cores. If the number of components is n and we have `comm_sz` cores or processes, let's assume that n is evenly divisible by `comm_sz` and define `local_n` $= n/$`comm_sz`. Then we can simply assign blocks of `local_n` consecutive components to each process. The four columns on the left of Table 3.4 show an

```
void Vector_sum(double x[], double y[], double z[], int n) {
   int i;

   for (i = 0; i < n; i++)
      z[i] = x[i] + y[i];
}  /* Vector_sum */
```

Program 3.7: A serial implementation of vector addition.

Table 3.4 Different partitions of a 12-component vector among 3 processes.

Process	Components											
	Block				Cyclic				Block-cyclic Blocksize = 2			
0	0	1	2	3	0	3	6	9	0	1	6	7
1	4	5	6	7	1	4	7	10	2	3	8	9
2	8	9	10	11	2	5	8	11	4	5	10	11

example when $n = 12$ and comm_sz $= 3$. This is often called a **block partition** of the vector.

An alternative to a block partition is a **cyclic partition**. In a cyclic partition, we assign the components in a round-robin fashion. The four columns in the middle of Table 3.4 show an example when $n = 12$ and comm_sz $= 3$. So process 0 gets component 0, process 1 gets component 1, process 2 gets component 2, process 0 gets component 3, and so on.

A third alternative is a **block-cyclic partition**. The idea here is that instead of using a cyclic distribution of individual components, we use a cyclic distribution of *blocks* of components. So a block-cyclic distribution isn't fully specified until we decide how large the blocks are. If comm_sz $= 3$, $n = 12$, and the blocksize $b = 2$, an example is shown in the four columns on the right of Table 3.4.

Once we've decided how to partition the vectors, it's easy to write a parallel vector addition function: each process simply adds its assigned components. Furthermore, regardless of the partition, each process will have local_n components of the vector, and, to save on storage, we can just store these on each process as an array of local_n elements. Thus each process will execute the function shown in Program 3.8. Although the names of the variables have been changed to emphasize the fact that the function is operating on only the process's portion of the vector, this function is virtually identical to the original serial function.

3.4.7 Scatter

Now suppose we want to test our vector addition function. It would be convenient to be able to read the dimension of the vectors and then read in the vectors **x** and **y**. We already know how to read in the dimension of the vectors: process 0 can prompt the

```
void Parallel_vector_sum(
      double  local_x[]  /* in  */,
      double  local_y[]  /* in  */,
      double  local_z[]  /* out */,
      int     local_n    /* in  */) {
   int local_i;

   for (local_i = 0; local_i < local_n; local_i++)
      local_z[local_i] = local_x[local_i] + local_y[local_i];
}  /* Parallel_vector_sum */
```

Program 3.8: A parallel implementation of vector addition.

user, read in the value, and broadcast the value to the other processes. We might try something similar with the vectors: process 0 could read them in and broadcast them to the other processes. However, this could be very wasteful. If there are 10 processes and the vectors have 10,000 components, then each process will need to allocate storage for vectors with 10,000 components, when it is only operating on subvectors with 1000 components. If, for example, we use a block distribution, it would be better if process 0 sent only components 1000 to 1999 to process 1, components 2000 to 2999 to process 2, etc. Using this approach processes 1 to 9 would only need to allocate storage for the components they're actually using.

Thus we might try writing a function that reads in an entire vector on process 0, but only sends the needed components to each of the other processes. For the communication MPI provides just such a function:

```
int MPI_Scatter(
      void*         send_buf_p  /* in  */,
      int           send_count  /* in  */,
      MPI_Datatype  send_type   /* in  */,
      void*         recv_buf_p  /* out */,
      int           recv_count  /* in  */,
      MPI_Datatype  recv_type   /* in  */,
      int           src_proc    /* in  */,
      MPI_Comm      comm        /* in  */);
```

If the communicator comm contains comm_sz processes, then MPI_Scatter divides the data referenced by send_buf_p into comm_sz pieces—the first piece goes to process 0, the second to process 1, the third to process 2, and so on. For example, suppose we're using a block distribution and process 0 has read in all of an n-component vector into send_buf_p. Then process 0 will get the first local_n $= n/$comm_sz components, process 1 will get the next local_n components, and so on. Each process should pass its local vector as the recv_buf_p argument, and the recv_count argument should be local_n. Both send_type and recv_type should be MPI_DOUBLE, and src_proc should be 0. Perhaps surprisingly, send_count should also be local_n—send_count is the *amount of data going to each process*; it's *not* the amount of data in the memory referred to by send_buf_p. If we use a block distribution and MPI_Scatter, we can read in a vector using the function Read_vector shown in Program 3.9.

```
void Read_vector(
      double    local_a[]   /* out */,
      int       local_n     /* in  */,
      int       n           /* in  */,
      char      vec_name[]  /* in  */,
      int       my_rank     /* in  */,
      MPI_Comm  comm        /* in  */) {

   double* a = NULL;
   int i;

   if (my_rank == 0) {
      a = malloc(n*sizeof(double));
      printf("Enter the vector %s\n", vec_name);
      for (i = 0; i < n; i++)
         scanf("%lf", &a[i]);
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
            MPI_DOUBLE, 0, comm);
      free(a);
   } else {
      MPI_Scatter(a, local_n, MPI_DOUBLE, local_a, local_n,
            MPI_DOUBLE, 0, comm);
   }
}  /* Read_vector */
```

Program 3.9: A function for reading and distributing a vector.

One point to note here is that MPI_Scatter sends the first block of send_count objects to process 0, the next block of send_count objects to process 1, and so on. So this approach to reading and distributing the input vectors will only be suitable if we're using a block distribution and *n*, the number of components in the vectors, is evenly divisible by comm_sz. We'll discuss how we might deal with a cyclic or block cyclic distribution in Section 3.5, and we'll look at dealing with the case in which *n* is not evenly divisible by comm_sz in Exercise 3.13.

3.4.8 Gather

Of course, our test program will be useless unless we can see the result of our vector addition. So we need to write a function for printing out a distributed vector. Our function can collect all of the components of the vector onto process 0, and then process 0 can print all of the components. The communication in this function can be carried out by MPI_Gather:

```
int MPI_Gather(
      void*          send_buf_p   /* in   */,
      int            send_count   /* in   */,
      MPI_Datatype   send_type    /* in   */,
      void*          recv_buf_p   /* out  */,
      int            recv_count   /* in   */,
      MPI_Datatype   recv_type    /* in   */,
      int            dest_proc    /* in   */,
      MPI_Comm       comm         /* in   */);
```

The data stored in the memory referred to by `send_buf_p` on process 0 is stored in the first block in `recv_buf_p`, the data stored in the memory referred to by `send_buf_p` on process 1 is stored in the second block referred to by `recv_buf_p`, and so on. So, if we're using a block distribution, we can implement our distributed vector print function, as shown in Program 3.10. Note that `recv_count` is the number of data items received from *each* process, not the total number of data items received.

```
void Print_vector(
      double     local_b[]   /* in */,
      int        local_n     /* in */,
      int        n           /* in */,
      char       title[]     /* in */,
      int        my_rank     /* in */,
      MPI_Comm   comm        /* in */) {

   double* b = NULL;
   int i;

   if (my_rank == 0) {
      b = malloc(n*sizeof(double));
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
            MPI_DOUBLE, 0, comm);
      printf("%s\n", title);
      for (i = 0; i < n; i++)
         printf("%f ", b[i]);
      printf("\n");
      free(b);
   } else {
      MPI_Gather(local_b, local_n, MPI_DOUBLE, b, local_n,
            MPI_DOUBLE, 0, comm);
   }
}  /* Print_vector */
```

Program 3.10: A function for printing a distributed vector.

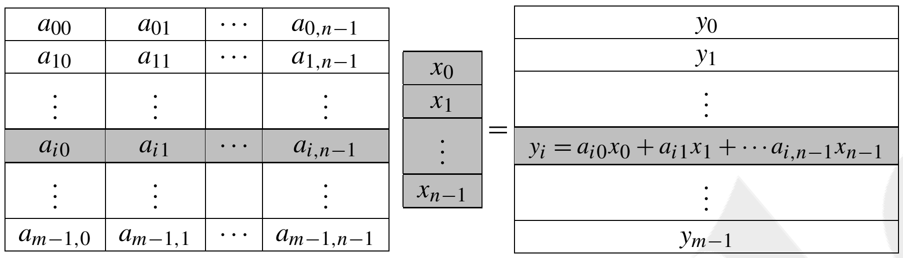
a_{00} a_{01} $\cdots$ $a_{0,n-1}$
a_{10} a_{11} $\cdots$ $a_{1,n-1}$
a_{i0} a_{i1} $\cdots$ $a_{i,n-1}$
$a_{m-1,0}$ $a_{m-1,1}$ $\cdots$ $a_{m-1,n-1}$
x_0 x_1 x_{n-1}
$=$
y_0
y_1
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
y_{m-1}

FIGURE 3.11

Matrix-vector multiplication.

The restrictions on the use of `MPI_Gather` are similar to those on the use of `MPI_Scatter`: our print function will only work correctly with vectors using a block distribution in which each block has the same size.

3.4.9 Allgather

As a final example, let's look at how we might write an MPI function that multiplies a matrix by a vector. Recall that if $A = (a_{ij})$ is an $m \times n$ matrix and $\mathbf{x}$ is a vector with n components, then $\mathbf{y} = A\mathbf{x}$ is a vector with m components, and we can find the ith component of y by forming the dot product of the ith row of A with $\mathbf{x}$:

$$y_i = a_{i0}x_0 + a_{i1}x_1 + a_{i2}x_2 + \cdots a_{i,n-1}x_{n-1}.$$

(See Fig. 3.11.)

So we might write pseudocode for serial matrix multiplication as follows:

```
/* For each row of A */
for (i = 0; i < m; i++) {
   /* Form dot product of ith row with x */
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

In fact, this could be actual C code. However, there are some peculiarities in the way that C programs deal with two-dimensional arrays (see Exercise 3.14). So C programmers frequently use one-dimensional arrays to "simulate" two-dimensional arrays. The most common way to do this is to list the rows one after another. For example, the two-dimensional array

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{pmatrix}$$

would be stored as the one-dimensional array

$$0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11.$$

In this example, if we start counting rows and columns from 0, then the element stored in row 2 and column 1—the 9—in the two-dimensional array is located in position $2 \times 4 + 1 = 9$ in the one-dimensional array. More generally, if our array has n columns, when we use this scheme, we see that the element stored in row i and column j is located in position $i \times n + j$ in the one-dimensional array.

Using this one-dimensional scheme, we get the C function shown in Program 3.11.

```
void Mat_vect_mult(
      double  A[]  /* in   */,
      double  x[]  /* in   */,
      double  y[]  /* out  */,
      int     m    /* in   */,
      int     n    /* in   */) {
   int i, j;

   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i*n+j]*x[j];
   }
}  /* Mat_vect_mult */
```

Program 3.11: Serial matrix-vector multiplication.

Now let's see how we might parallelize this function. An individual task can be the multiplication of an element of A by a component of $\mathbf{x}$ and the addition of this product into a component of $\mathbf{y}$. That is, each execution of the statement

```
y[i] += A[i*n+j]*x[j];
```

is a task, so we see that if `y[i]` is assigned to process q, then it would be convenient to also assign row i of `A` to process q. This suggests that we partition `A` by *rows*. We could partition the rows using a block distribution, a cyclic distribution, or a block-cyclic distribution. In MPI it's easiest to use a block distribution. So let's use a block distribution of the rows of `A`, and, as usual, assume that `comm_sz` evenly divides m, the number of rows.

We are distributing `A` by rows so that the computation of `y[i]` will have all of the needed elements of `A`, so we should distribute `y` by blocks. That is, if the ith row of A is assigned to process q, then the ith component of `y` should also be assigned to process q.

Now the computation of y[i] involves all the elements in the ith row of A and *all* the components of x. So we could minimize the amount of communication by simply assigning all of x to each process. However, in actual applications—especially when the matrix is square—it's often the case that a program using matrix-vector multiplication will execute the multiplication many times, and the result vector **y** from one multiplication will be the input vector **x** for the next iteration. In practice, then, we usually assume that the distribution for x is the same as the distribution for y.

So if x has a block distribution, how can we arrange that each process has access to *all* the components of x before we execute the following loop?

```
for (j = 0; j < n; j++)
   y[i] += A[i*n+j]*x[j];
```

Using the collective communications we're already familiar with, we could execute a call to MPI_Gather, followed by a call to MPI_Bcast. This would, in all likelihood, involve two tree-structured communications, and we may be able to do better by using a butterfly. So, once again, MPI provides a single function:

```
int MPI_Allgather(
      void*          send_buf_p   /* in  */,
      int            send_count   /* in  */,
      MPI_Datatype   send_type    /* in  */,
      void*          recv_buf_p   /* out */,
      int            recv_count   /* in  */,
      MPI_Datatype   recv_type    /* in  */,
      MPI_Comm       comm         /* in  */);
```

This function concatenates the contents of each process's send_buf_p and stores this in each process's recv_buf_p. As usual, recv_count is the amount of data being received from *each* process. So in most cases, recv_count will be the same as send_count.

We can now implement our parallel matrix-vector multiplication function as shown in Program 3.12. If this function is called many times, we can improve performance by allocating x once in the calling function and passing it as an additional argument.

3.5 MPI-derived datatypes

In virtually all distributed-memory systems, communication can be *much* more expensive than local computation. For example, sending a **double** from one node to another will take far longer than adding two **double**s stored in the local memory of a node. Furthermore, the cost of sending a fixed amount of data in multiple messages is usually much greater than the cost of sending a single message with the same amount of data. For example, we would expect the following pair of **for** loops to be much slower than the single send/receive pair:

```
void Mat_vect_mult(
      double    local_A[]  /* in  */,
      double    local_x[]  /* in  */,
      double    local_y[]  /* out */,
      int       local_m    /* in  */,
      int       n          /* in  */,
      int       local_n    /* in  */,
      MPI_Comm  comm       /* in  */) {
   double* x;
   int local_i, j;
   int local_ok = 1;

   x = malloc(n*sizeof(double));
   MPI_Allgather(local_x, local_n, MPI_DOUBLE,
         x, local_n, MPI_DOUBLE, comm);

   for (local_i = 0; local_i < local_m; local_i++) {
      local_y[local_i] = 0.0;
      for (j = 0; j < n; j++)
         local_y[local_i] += local_A[local_i*n+j]*x[j];
   }
   free(x);
}  /* Mat_vect_mult */
```

Program 3.12: An MPI matrix-vector multiplication function.

```
   double x[1000];
   . . .
   if (my_rank == 0)
      for (i = 0; i < 1000; i++)
         MPI_Send(&x[i], 1, MPI_DOUBLE, 1, 0, comm);
   else /* my_rank == 1 */
      for (i = 0; i < 1000; i++)
         MPI_Recv(&x[i], 1, MPI_DOUBLE, 0, 0, comm, &status);

   if (my_rank == 0)
      MPI_Send(x, 1000, MPI_DOUBLE, 1, 0, comm);
   else  /* my_rank == 1 */
      MPI_Recv(x, 1000, MPI_DOUBLE, 0, 0, comm, &status);
```

In fact, on one of our systems, the code with the loops of sends and receives takes nearly 50 times longer. On another system, the code with the loops takes more than 100 times longer. Thus if we can reduce the total number of messages we send, we're likely to improve the performance of our programs.

MPI provides three basic approaches to consolidating data that might otherwise require multiple messages: the `count` argument to the various communication functions, derived datatypes, and MPI_Pack/Unpack. We've already seen the count argument—it can be used to group contiguous array elements into a single message. In this section, we'll discuss one method for building derived datatypes. In the Exercises, we'll take a look at some other methods for building derived datatypes and MPI_Pack/Unpack.

In MPI, a **derived datatype** can be used to represent any collection of data items in memory by storing both the types of the items and their relative locations in memory. The idea here is that if a function that sends data knows the types and the relative locations in memory of a collection of data items, it can collect the items from memory before they are sent. Similarly, a function that receives data can distribute the items into their correct destinations in memory when they're received. As an example, in our trapezoidal rule program, we needed to call MPI_Bcast three times: once for the left endpoint a, once for the right endpoint b, and once for the number of trapezoids n. As an alternative, we could build a single derived datatype that consists of two **double**s and one **int**. If we do this, we'll only need one call to MPI_Bcast. On process 0, a, b, and n will be sent with the one call, while on the other processes, the values will be received with the call.

Formally, a derived datatype consists of a sequence of basic MPI datatypes together with a *displacement* for each of the datatypes. In our trapezoidal rule example, suppose that on process 0 the variables a, b, and n are stored in memory locations with the following addresses:

Variable	Address
a	24
b	40
n	48

Then the following derived datatype could represent these data items:

$$\{(\texttt{MPI\_DOUBLE}, 0), (\texttt{MPI\_DOUBLE}, 16), (\texttt{MPI\_INT}, 24)\}.$$

The first element of each pair corresponds to the type of the data, and the second element of each pair is the displacement of the data element from the *beginning* of the type. We've assumed that the type begins with a, so it has displacement 0, and the other elements have displacements measured in bytes, from a: b is $40 - 24 = 16$ bytes beyond the start of a, and n is $48 - 24 = 24$ bytes beyond the start of a.

We can use MPI_Type_create_struct to build a derived datatype that consists of individual elements that have different basic types:

```
int MPI_Type_create_struct(
      int            count                      /* in    */,
      int            array_of_blocklengths[]    /* in    */,
      MPI_Aint       array_of_displacements[]   /* in    */,
      MPI_Datatype   array_of_types[]           /* in    */,
      MPI_Datatype*  new_type_p                 /* out   */);
```

The argument count is the number of elements in the datatype, so for our example, it should be three. Each of the array arguments should have count elements. The first array, array_of_block_lengths, allows for the possibility that the individual data items might be arrays or subarrays. If, for example, the first element were an array containing five elements, we would have

```
array_of_blocklengths[0] = 5;
```

However, in our case, none of the elements is an array, so we can simply define

```
int array_of_blocklengths[3] = {1, 1, 1};
```

The third argument to MPI_Type_create_struct, array_of_displacements specifies the displacements in bytes, from the start of the message. So we want

```
array_of_displacements[] = {0, 16, 24};
```

To find these values, we can use the function MPI_Get_address:

```
int MPI_Get_address(
      void*      location_p  /* in  */,
      MPI_Aint*  address_p   /* out */);
```

It returns the address of the memory location referenced by location_p. The special type MPI_Aint is an integer type that is big enough to store an address on the system. Thus to get the values in array_of_displacements, we can use the following code:

```
MPI_Aint a_addr, b_addr, n_addr;

MPI_Get_address(&a, &a_addr);
array_of_displacements[0] = 0;
MPI_Get_address(&b, &b_addr);
array_of_displacements[1] = b_addr - a_addr;
MPI_Get_address(&n, &n_addr);
array_of_displacements[2] = n_addr - a_addr;
```

The array_of_datatypes should store the MPI datatypes of the elements. So we can just define

```
MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE,
      MPI_INT};
```

With these initializations, we can build the new datatype with the call

```
MPI_Datatype input_mpi_t;
. . .
MPI_Type_create_struct(3, array_of_blocklengths,
      array_of_displacements, array_of_types,
      &input_mpi_t);
```

Before we can use input_mpi_t in a communication function, we must first **commit** it with a call to

```
int MPI_Type_commit(
       MPI_Datatype*  new_mpi_t_p  /* in/out */);
```

This allows the MPI implementation to optimize its internal representation of the datatype for use in communication functions.

Now, to use input_mpi_t, we make the following call to MPI_Bcast on each process:

```
MPI_Bcast(&a, 1, input_mpi_t, 0, comm);
```

So we can use input_mpi_t, just as we would use one of the basic MPI datatypes.

In constructing the new datatype, it's likely that the MPI implementation had to allocate additional storage internally. So when we're through using the new type, we can free any additional storage used with a call to

```
int MPI_Type_free(MPI_Datatype*  old_mpi_t_p  /* in/out */);
```

We used the steps outlined here to define a Build_mpi_type function that our Get_input function can call. The new function and the updated Get_input function are shown in Program 3.13.

3.6 Performance evaluation of MPI programs

Let's take a look at the performance of the matrix-vector multiplication program. For the most part, we write parallel programs because we expect that they'll be faster than a serial program that solves the same problem. How can we verify this? We spent some time discussing this in Section 2.6, so we'll start by recalling some of the material we learned there.

3.6.1 Taking timings

We're usually not interested in the time taken from the start of program execution to the end of program execution. For example, in the matrix-vector multiplication, we're not interested in the time it takes to type in the matrix or print out the product. We're only interested in the time it takes to do the actual multiplication, so we need to modify our source code by adding in calls to a function that will tell us the amount of time that elapses from the beginning to the end of the actual matrix-vector multiplication. MPI provides a function, MPI_Wtime, that returns the number of seconds that have elapsed since some time in the past:

```
double MPI_Wtime(void);
```

We can time a block of MPI code as follows:

```
double start, finish;
. . .
start = MPI_Wtime();
/* Code to be timed */
. . .
```

```
void Build_mpi_type(
      double*         a_p            /* in   */,
      double*         b_p            /* in   */,
      int*            n_p            /* in   */,
      MPI_Datatype*   input_mpi_t_p  /* out  */) {

   int array_of_blocklengths[3] = {1, 1, 1};
   MPI_Datatype array_of_types[3] = {MPI_DOUBLE, MPI_DOUBLE,
      MPI_INT};
   MPI_Aint a_addr, b_addr, n_addr;
   MPI_Aint array_of_displacements[3] = {0};

   MPI_Get_address(a_p, &a_addr);
   MPI_Get_address(b_p, &b_addr);
   MPI_Get_address(n_p, &n_addr);
   array_of_displacements[1] = b_addr-a_addr;
   array_of_displacements[2] = n_addr-a_addr;
   MPI_Type_create_struct(3, array_of_blocklengths,
         array_of_displacements, array_of_types,
         input_mpi_t_p);
   MPI_Type_commit(input_mpi_t_p);
}  /* Build_mpi_type */

void Get_input(int my_rank, int comm_sz, double* a_p,
      double* b_p, int* n_p) {
   MPI_Datatype input_mpi_t;

   Build_mpi_type(a_p, b_p, n_p, &input_mpi_t);

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
   }
   MPI_Bcast(a_p, 1, input_mpi_t, 0, MPI_COMM_WORLD);

   MPI_Type_free(&input_mpi_t);
}  /* Get_input */
```

Program 3.13: The `Get_input` function with a derived datatype.

```
      finish = MPI_Wtime();
      printf("Proc %d > Elapsed time = %e seconds\n"
            my_rank, finish-start);
```

To time serial code, it's not necessary to link in the MPI libraries. There is a POSIX library function called `gettimeofday` that returns the number of microseconds that have elapsed since some point in the past. The syntax details aren't too important: there's a C macro GET_TIME defined in the header file timer.h that can be downloaded from the book's website. This macro should be called with a double argument:

```
#include "timer.h"
. . .
double now;
. . .
GET_TIME(now);
```

After executing this macro, now will store the number of seconds since some time in the past. So we can get the elapsed time of serial code with microsecond resolution by executing

```
#include "timer.h"
. . .
double start, finish;
. . .
GET_TIME(start);
/* Code to be timed */
. . .
GET_TIME(finish);
printf("Elapsed time = %e seconds\n", finish-start);
```

One point to stress here: GET_TIME is a macro, so the code that defines it is inserted directly into your source code by the preprocessor. Hence, it can operate directly on its argument, and the argument is a **double**, *not* a pointer to a **double**. A final note in this connection: Since timer.h is not in the system include file directory, it's necessary to tell the compiler where to find it if it's not in the directory where you're compiling. For example, if it's in the directory /home/peter/my_include, the following command can be used to compile a serial program that uses GET_TIME:

```
$ gcc -g -Wall -I/home/peter/my_include -o <executable>
     <source_code.c>
```

Both MPI_Wtime and GET_TIME return *wall clock* time. Recall that a timer, such as the C clock function, returns CPU time: the time spent in user code, library functions, and operating system code. It doesn't include idle time, which can be a significant part of parallel run time. For example, a call to MPI_Recv may spend a significant amount of time waiting for the arrival of a message. Wall clock time, on the other hand, gives total elapsed time. So it includes idle time.

There are still a few remaining issues. First, as we've described it, our parallel program will report comm_sz times: one for each process. We would like to have it report a single time. Ideally, all of the processes would start execution of the matrix-vector multiplication at the same time, and then we would report the time that elapsed when the last process finished. In other words, the parallel execution time would be the time it took the "slowest" process to finish. We can't get exactly this time because we can't ensure that all the processes start at the same instant. However, we can come reasonably close. The MPI collective communication function MPI_Barrier ensures that no process will return from calling it until every process in the communicator has started calling it. Its syntax is

```
int MPI_Barrier(MPI_Comm  comm  /* in */);
```

So the following code can be used to time a block of MPI code and report a single elapsed time:

```
double local_start, local_finish, local_elapsed, elapsed;
. . .
MPI_Barrier(comm);
local_start = MPI_Wtime();
/* Code to be timed */
. . .
local_finish = MPI_Wtime();
local_elapsed = local_finish − local_start;
MPI_Reduce(&local_elapsed, &elapsed, 1, MPI_DOUBLE,
   MPI_MAX, 0, comm);

if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", elapsed);
```

Note that the call to `MPI_Reduce` is using the `MPI_MAX` operator: it finds the largest of the input arguments `local_elapsed`.

As we noted in Chapter 2, we also need to be aware of variability in timings: when we run a program several times, we're likely to see a substantial variation in the times. This will be true, even if for each run we use the same input, the same number of processes, and the same system. This is because the interaction of the program with the rest of the system, especially the operating system, is unpredictable. Since this interaction will almost certainly not make the program run faster than it would run on a "quiet" system, we usually report the *minimum* run-time, rather than the mean or median. (For further discussion of this, see [6].)

Finally, when we run an MPI program on a hybrid system in which the nodes are multicore processors, we'll only run one MPI process on each node. This may reduce contention for the interconnect and result in somewhat better run-times. It may also reduce variability in run-times.

3.6.2 Results

The results of timing the matrix-vector multiplication program are shown in Table 3.5. The input matrices were square. The times shown are in milliseconds, and we've rounded each time to two significant digits. The times for `comm_sz` = 1 are the run-times of the serial program running on a single core of the distributed memory system. Not surprisingly, if we fix `comm_sz`, and increase n, the order of the matrix, the run-times increase. For relatively small numbers of processes, doubling n results in roughly a four-fold increase in the run-time. However, for large numbers of processes, this formula breaks down.

Table 3.5 Run-times of serial and parallel matrix-vector multiplication (times are in milliseconds).

	Order of Matrix				
comm_sz	**1024**	**2048**	**4096**	**8192**	**16,384**
1	4.1	16.0	64.0	270	1100
2	2.3	8.5	33.0	140	560
4	2.0	5.1	18.0	70	280
8	1.7	3.3	9.8	36	140
16	1.7	2.6	5.9	19	71

If we fix n and increase comm_sz, the run-times usually decrease. In fact, for large values of n, doubling the number of processes roughly halves the overall run-time. However, for small n, there is very little benefit in increasing comm_sz. In fact, in going from 8 to 16 processes when $n = 1024$, the overall run time is unchanged.

These timings are fairly typical of parallel run-times—as we increase the problem size, the run-times increase, and this is true regardless of the number of processes. The rate of increase can be fairly constant (e.g., the one process times) or it can vary wildly (e.g., the sixteen process times). As we increase the number of processes, the run-times typically decrease for a while. However, at some point, the run-times can actually start to get worse. The closest we came to this behavior was going from 8 to 16 processes when the matrix had order 1024.

The explanation for this is that there is a fairly common relation between the run-times of serial programs and the run-times of corresponding parallel programs. Recall that we denote the serial run-time by T_{serial}. Since it typically depends on the size of the input, n, we'll frequently denote it as $T_{\text{serial}}(n)$. Also recall that we denote the parallel run-time by T_{parallel}. Since it depends on both the input size, n, and the number of processes, comm_sz $= p$, we'll frequently denote it as $T_{\text{parallel}}(n, p)$. As we noted in Chapter 2, it's often the case that the parallel program will divide the work of the serial program among the processes, and add in some overhead time, which we denoted T_{overhead}:

$$T_{\text{parallel}}(n, p) = T_{\text{serial}}(n)/p + T_{\text{overhead}}.$$

In MPI programs, the parallel overhead typically comes from communication, and it can depend on both the problem size and the number of processes.

It's not to hard to see that this formula applies to our matrix-vector multiplication program. The heart of the serial program is the pair of nested for loops:

```
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i*n+j]*x[j];
}
```

If we only count floating point operations, the inner loop carries out n multiplications and n additions, for a total of $2n$ floating point operations. Since we execute the inner loop m times, the pair of loops executes a total of $2mn$ floating point operations. So when $m = n$,

$$T_{\text{serial}}(n) \approx an^2$$

for some constant a. (The symbol $\approx$ means "is approximately equal to.")

If the serial program multiplies an $n \times n$ matrix by an n-dimensional vector, then each process in the parallel program multiplies an $n/p \times n$ matrix by an n-dimensional vector. The *local* matrix-vector multiplication part of the parallel program therefore executes n^2/p floating point operations. Thus it appears that this *local* matrix-vector multiplication reduces the work per process by a factor of p.

However, the parallel program also needs to complete a call to `MPI_Allgather` before it can carry out the local matrix-vector multiplication. In our example, it appears that

$$T_{\text{parallel}}(n, p) = T_{\text{serial}}(n)/p + T_{\text{allgather}}.$$

Furthermore, in light of our timing data, it appears that for smaller values of p and larger values of n, the dominant term in our formula is $T_{\text{serial}}(n)/p$. To see this, observe first that for small p (e.g., $p = 2, 4$), doubling p roughly halves the overall run time. For example,

$$\begin{aligned}
T_{\text{serial}}(4096) &= 1.9 \times T_{\text{parallel}}(4096, 2) \\
T_{\text{serial}}(8192) &= 1.9 \times T_{\text{parallel}}(8192, 2) \\
T_{\text{parallel}}(8192, 2) &= 2.0 \times T_{\text{parallel}}(8192, 4) \\
T_{\text{serial}}(16{,}384) &= 2.0 \times T_{\text{parallel}}(16{,}384, 2) \\
T_{\text{parallel}}(16{,}384, 2) &= 2.0 \times T_{\text{parallel}}(16{,}384, 4)
\end{aligned}$$

Also, if we fix p at a small value (e.g., $p = 2, 4$), then increasing n seems to have approximately the same effect as increasing n for the serial program. For example,

$$\begin{aligned}
T_{\text{serial}}(4096) &= 4.0 \times T_{\text{serial}}(2048) \\
T_{\text{parallel}}(4096, 2) &= 3.9 \times T_{\text{serial}}(2048, 2) \\
T_{\text{parallel}}(4096, 4) &= 3.5 \times T_{\text{serial}}(2048, 4) \\
T_{\text{serial}}(8192) &= 4.2 \times T_{\text{serial}}(4096) \\
T_{\text{serial}}(8192, 2) &= 4.2 \times T_{\text{parallel}}(4096, 2) \\
T_{\text{parallel}}(8192, 4) &= 3.9 \times T_{\text{parallel}}(8192, 4)
\end{aligned}$$

These observations suggest that the parallel run-times are behaving much as the run-times of the serial program, i.e., $T_{\text{serial}}(n)/p$. In other words, the overhead $T_{\text{allgather}}$ has little effect on the performance.

Table 3.6 Speedups of parallel matrix-vector multiplication.

	Order of Matrix				
comm_sz	1024	2048	4096	8192	16,384
1	1.0	1.0	1.0	1.0	1.0
2	1.8	1.9	1.9	1.9	2.0
4	2.1	3.1	3.6	3.9	3.9
8	2.4	4.8	6.5	7.5	7.9
16	2.4	6.2	10.8	14.2	15.5

On the other hand, for small n and large p, these patterns break down. For example,

$$T_{\text{parallel}}(1024, 8) = 1.0 \times T_{\text{parallel}}(1024, 16)$$
$$T_{\text{parallel}}(2048, 16) = 1.5 \times T_{\text{parallel}}(1024, 16)$$

So it appears that for small n and large p, the dominant term in our formula for T_{parallel} is $T_{\text{allgather}}$.

3.6.3 Speedup and efficiency

Recall that the most widely used measure of the relation between the serial and the parallel run-times is the **speedup**. It's just the ratio of the serial run-time to the parallel run-time:

$$S(n, p) = \frac{T_{\text{serial}}(n)}{T_{\text{parallel}}(n, p)}.$$

The ideal value for $S(n, p)$ is p. If $S(n, p) = p$, then our parallel program with comm_sz $= p$ processes is running p times faster than the serial program. In practice, this speedup, sometimes called **linear speedup**, is not often achieved. Our matrix-vector multiplication program got the speedups shown in Table 3.6. For small p and large n, our program obtained nearly linear speedup. On the other hand, for large p and small n, the speedup was considerably less than p. The worst case being $n = 1024$ and $p = 16$, when we only managed a speedup of 2.4.

Also recall that another widely used measure of parallel performance is parallel **efficiency**. This is "per process" speedup:

$$E(n, p) = \frac{S(n, p)}{p} = \frac{T_{\text{serial}}(n)}{p \times T_{\text{parallel}}(n, p)}.$$

So linear speedup corresponds to a parallel efficiency of $p/p = 1.0$, and, in general, we expect that our efficiencies will usually be less than 1.

Table 3.7 Efficiencies of parallel matrix-vector multiplication.

	Order of Matrix				
comm_sz	**1024**	**2048**	**4096**	**8192**	**16,384**
1	1.00	1.00	1.00	1.00	1.00
2	0.89	0.94	0.97	0.96	0.98
4	0.51	0.78	0.89	0.96	0.98
8	0.30	0.61	0.82	0.94	0.98
16	0.15	0.39	0.68	0.89	0.97

The efficiencies for the matrix-vector multiplication program are shown in Table 3.7. Once again, for small p and large n, our parallel efficiencies are near linear, and for large p and small n, they are very far from linear.

3.6.4 Scalability

Our parallel matrix-vector multiplication program doesn't come close to obtaining linear speedup for small n and large p. Does this mean that it's not a good program? Many computer scientists answer this question by looking at the "scalability" of the program. Recall that very roughly speaking a program is **scalable** if the problem size can be increased at a rate so that the efficiency doesn't decrease as the number of processes increase.

The problem with this definition is the phrase "the problem size can be increased at a rate ...." Consider two parallel programs: program A and program B. Suppose that if $p \geq 2$, the efficiency of program A is 0.75, regardless of problem size. Also suppose that the efficiency of program B is $n/(625p)$, provided $p \geq 2$ and $1000 \leq n \leq 625p$. Then according to our "definition," both programs are scalable. For program A, the rate of increase needed to maintain constant efficiency is 0, while for program B if we increase n at the same rate as we increase p, we'll maintain a constant efficiency. For example, if $n = 1000$ and $p = 2$, the efficiency of B is 0.80. If we then double p to 4 and we leave the problem size at $n = 1000$, the efficiency will drop to 0.4, but if we also double the problem size to $n = 2000$, the efficiency will remain constant at 0.8. Program A is thus *more* scalable than B, but both satisfy our definition of scalability.

Looking at our table of parallel efficiencies (Table 3.7), we see that our matrix-vector multiplication program definitely doesn't have the same scalability as program A: in almost every case when p is increased, the efficiency decreases. On the other hand, the program is somewhat like program B: if $p \geq 2$ and we increase both p and n by a factor of 2, the parallel efficiency, for the most part, actually increases. Furthermore, the only exceptions occur when we increase p from 2 to 4, and when computer scientists discuss scalability, they're usually interested in large values of p. When p is increased from 4 to 8 or from 8 to 16, our efficiency always increases when we increase n by a factor of 2.

Recall that programs that can maintain a constant efficiency without increasing the problem size are sometimes said to be **strongly scalable**. Programs that can maintain a constant efficiency if the problem size increases at the same rate as the number of processes are sometimes said to be **weakly scalable**. Program *A* is strongly scalable, and program *B* is weakly scalable. Furthermore, our matrix-vector multiplication program is also apparently weakly scalable.

3.7 A parallel sorting algorithm

What do we mean by a parallel sorting algorithm in a distributed memory environment? What would its "input" be and what would its "output" be? The answers depend on where the keys are stored. We can start or finish with the keys distributed among the processes or assigned to a single process. In this section, we'll look at an algorithm that starts and finishes with the keys distributed among the processes. In programming assignment 3.8, we'll look at an algorithm that finishes with the keys assigned to a single process.

If we have a total of n keys and $p =$ `comm_sz` processes, our algorithm will start and finish with n/p keys assigned to each process. (As usual, we'll assume n is evenly divisible by p.) At the start, there are no restrictions on which keys are assigned to which processes. However, when the algorithm terminates,

- the keys assigned to each process should be sorted in (say) increasing order, and
- if $0 \leq q < r < p$, then each key assigned to process q should be less than or equal to every key assigned to process r.

So if we lined up the keys according to process rank—keys from process 0 first, then keys from process 1, and so on—then the keys would be sorted in increasing order. For the sake of explicitness, we'll assume our keys are ordinary **ints**.

3.7.1 Some simple serial sorting algorithms

Before starting, let's look at a couple of simple serial sorting algorithms. Perhaps the best-known serial sorting algorithm is bubble sort. (See Program 3.14.) The array `a` stores the unsorted keys when the function is called, and the sorted keys when the function returns. The number of keys in `a` is `n`. The algorithm proceeds by comparing the elements of the list `a` pairwise: `a[0]` is compared to `a[1]`, `a[1]` is compared to `a[2]`, and so on. Whenever a pair is out of order, the entries are swapped, so in the first pass through the outer loop, when `list_length` = `n`, the largest value in the list will be moved into `a[n−1]`. The next pass will ignore this last element, and it will move the next-to-the-largest element into `a[n−2]`. Thus as `list_length` decreases, successively more elements get assigned to their final positions in the sorted list.

There isn't much point in trying to parallelize this algorithm because of the inherently sequential ordering of the comparisons. To see this, suppose that `a[i−1]` = 9, `a[i]` = 5, and `a[i+1]` = 7. The algorithm will first compare 9 and 5 and swap them;

```
void Bubble_sort(
      int  a[]  /* in/out */,
      int  n    /* in     */) {
   int list_length, i, temp;

   for (list_length = n; list_length >= 2; list_length--)
      for (i = 0; i < list_length-1; i++)
         if (a[i] > a[i+1]) {
            temp = a[i];
            a[i] = a[i+1];
            a[i+1] = temp;
         }

}  /* Bubble_sort */
```

Program 3.14: Serial bubble sort.

it will then compare 9 and 7 and swap them; and we'll have the sequence 5, 7, 9. If we try to do the comparisons out of order, that is, if we compare the 5 and 7 first and then compare the 9 and 5, we'll wind up with the sequence 5, 9, 7. Therefore the order in which the "compare-swaps" take place is essential to the correctness of the algorithm.

A variant of bubble sort, known as *odd-even transposition sort*, has considerably more opportunities for parallelism. The key idea is to "decouple" the compare-swaps. The algorithm consists of a sequence of *phases*, of two different types. During *even* phases, compare-swaps are executed on the pairs

$$(a[0], a[1]), (a[2], a[3]), (a[4], a[5]), \ldots,$$

and during *odd* phases, compare-swaps are executed on the pairs

$$(a[1], a[2]), (a[3], a[4]), (a[5], a[6]), \ldots.$$

Here's a small example:

Start: 5, 9, 4, 3
Even phase: Compare-swap (5, 9) and (4, 3), getting the list 5, 9, 3, 4.
Odd phase: Compare-swap (9, 3), getting the list 5, 3, 9, 4.
Even phase: Compare-swap (5, 3), and (9, 4) getting the list 3, 5, 4, 9.
Odd phase: Compare-swap (5, 4) getting the list 3, 4, 5, 9.

This example required four phases to sort a four-element list. In general, it may require fewer phases, but the following theorem guarantees that we can sort a list of n elements in at most n phases.

Theorem. *Suppose A is a list with n keys, and A is the input to the odd-even transposition sort algorithm. Then after n phases, A will be sorted.*

Program 3.15 shows code for a serial odd-even transposition sort function.

```
void Odd_even_sort(
      int  a[]  /* in/out */,
      int  n    /* in     */) {
   int phase, i, temp;

   for (phase = 0; phase < n; phase++)
      if (phase % 2 == 0) { /* Even phase */
         for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) {
               temp = a[i];
               a[i] = a[i-1];
               a[i-1] = temp;
            }
      } else { /* Odd phase */
         for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) {
               temp = a[i];
               a[i] = a[i+1];
               a[i+1] = temp;
            }
      }
}  /* Odd_even_sort */
```

Program 3.15: Serial odd-even transposition sort.

3.7.2 Parallel odd-even transposition sort

It should be clear that odd-even transposition sort has considerably more opportunities for parallelism than bubble sort, because all of the compare-swaps in a single phase can happen simultaneously. Let's try to exploit this.

There are a number of possible ways to apply Foster's methodology. Here's one:

- *Tasks*: Determine the value of a[i] at the end of phase j.
- *Communications*: The task that's determining the value of a[i] needs to communicate with either the task determining the value of a[i−1] or a[i+1]. Also the value of a[i] at the end of phase j needs to be available for determining the value of a[i] at the end of phase $j+1$.

This is illustrated in Fig. 3.12, where we've labeled the task determining the value of a[k] with a[k].

Now recall that when our sorting algorithm starts and finishes execution, each process is assigned n/p keys. In this case, our aggregation and mapping are at least partially specified by the description of the problem. Let's look at two cases.

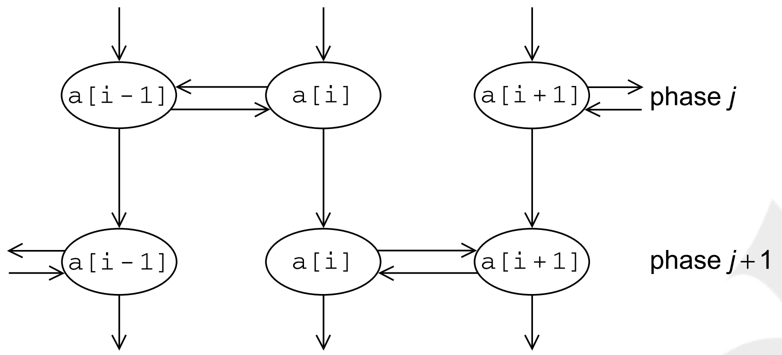

FIGURE 3.12

Communications among tasks in odd-even sort. Tasks determining `a[k]` are labeled with `a[k]`.

When $n = p$, Fig. 3.12 makes it fairly clear how the algorithm should proceed. Depending on the phase, process i can send its current value, `a[i]`, either to process $i - 1$ or process $i + 1$. At the same time, it should receive the value stored on process $i - 1$ or process $i + 1$, respectively, and then decide which of the two values it should store as `a[i]` for the next phase.

However, it's unlikely that we'll actually want to apply the algorithm when $n = p$, since we're unlikely to have more than a few hundred or a few thousand processors at our disposal, and sorting a few thousand values is usually a fairly trivial matter for a single processor. Furthermore, even if we do have access to thousands or even millions of processors, the added cost of sending and receiving a message for each compare-exchange will slow the program down so much that it will be useless. Remember that the cost of communication is usually much greater than the cost of "local" computation—for example, a compare-swap.

How should this be modified when each process is storing $n/p > 1$ elements? (Recall that we're assuming that n is evenly divisible by p.) Let's look at an example. Suppose we have $p = 4$ processes and $n = 16$ keys, as shown in Table 3.8. In the first place, we can apply a fast serial sorting algorithm to the keys assigned to each process. For example, we can use the C library function `qsort` on each process to sort the local keys. Now if we had one element per process, 0 and 1 would exchange elements, and 2 and 3 would exchange. So let's try this: Let's have 0 and 1 exchange *all* their elements, and 2 and 3 exchange all of theirs. Then it would seem natural for 0 to keep the four smaller elements and 1 to keep the larger. Similarly, 2 should keep the smaller and 3 the larger. This gives us the situation shown in the third row of the table. Once again, looking at the one element per process case, in phase 1, processes 1 and 2 exchange their elements and processes 0 and 4 are idle. If process 1 keeps the smaller and 2 the larger elements, we get the distribution shown in the fourth row. Continuing this process for two more phases results in a sorted list. That is, each

Table 3.8 Parallel odd-even transposition sort.

	Process			
Time	**0**	**1**	**2**	**3**
Start	15, 11, 9, 16	3, 14, 8, 7	4, 6, 12, 10	5, 2, 13, 1
After Local Sort	9, 11, 15, 16	3, 7, 8, 14	4, 6, 10, 12	1, 2, 5, 13
After Phase 0	3, 7, 8, 9	11, 14, 15, 16	1, 2, 4, 5	6, 10, 12, 13
After Phase 1	3, 7, 8, 9	1, 2, 4, 5	11, 14, 15, 16	6, 10, 12, 13
After Phase 2	1, 2, 3, 4	5, 7, 8, 9	6, 10, 11, 12	13, 14, 15, 16
After Phase 3	1, 2, 3, 4	5, 6, 7, 8	9, 10, 11, 12	13, 14, 15, 16

process's keys are stored in increasing order, and if $q < r$, then the keys assigned to process q are less than or equal to the keys assigned to process r.

In fact, our example illustrates the worst-case performance of this algorithm:

Theorem. *If parallel odd-even transposition sort is run with p processes, then after p phases, the input list will be sorted.*

The parallel algorithm is clear to a human computer:

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

However, there are some details that we need to clear up before we can convert the algorithm into an MPI program.

First, how do we compute the partner rank? And what is the partner rank when a process is idle? If the phase is even, then odd-ranked partners exchange with `my_rank` − 1 and even-ranked partners exchange with `my_rank`+1. In odd phases, the calculations are reversed. However, these calculations can return some invalid ranks: if `my_rank` = 0 or `my_rank` = `comm_sz`−1, the partner rank can be -1 or `comm_sz`. But when either `partner` = −1 or `partner` = `comm_sz`, the process should be idle. So we can use the rank computed by `Compute_partner` to determine whether a process is idle:

```
if (phase % 2 == 0)        /* Even phase */
   if (my_rank % 2 != 0)      /* Odd rank */
      partner = my_rank − 1;
   else                       /* Even rank */
      partner = my_rank + 1;
      else                          /* Odd phase */
         if (my_rank % 2 != 0)      /* Odd rank */
            partner = my_rank + 1;
         else                       /* Even rank */
            partner = my_rank − 1;
      if (partner == −1 || partner == comm_sz)
         partner = MPI_PROC_NULL;
```

MPI_PROC_NULL is a constant defined by MPI. When it's used as the source or destination rank in a point-to-point communication, no communication will take place, and the call to the communication will simply return.

3.7.3 Safety in MPI programs

If a process is not idle, we might try to implement the communication with a call to MPI_Send and a call to MPI_Recv:

```
   MPI_Send(my_keys, n/comm_sz, MPI_INT, partner, 0, comm);
   MPI_Recv(temp_keys, n/comm_sz, MPI_INT, partner, 0, comm,
         MPI_STATUS_IGNORE);
```

This, however, might result in the program's hanging or crashing. Recall that the MPI standard allows MPI_Send to behave in two different ways: it can simply copy the message into an MPI-managed buffer and return, or it can block until the matching call to MPI_Recv starts. Furthermore, many implementations of MPI set a threshold at which the system switches from buffering to blocking. That is, messages that are relatively small will be buffered by MPI_Send, but for larger messages, it will block. If the MPI_Send executed by each process blocks, no process will be able to start executing a call to MPI_Recv, and the program will hang or **deadlock**: that is, each process is blocked waiting for an event that will never happen.

A program that relies on MPI-provided buffering is said to be **unsafe**. Such a program may run without problems for various sets of input, but it may hang or crash with other sets. If we use MPI_Send and MPI_Recv in this way, our program will be unsafe, and it's likely that for small values of *n* the program will run without problems, while for larger values of *n*, it's likely that it will hang or crash.

There are a couple of questions that arise here:

1. In general, how can we tell if a program is safe?
2. How can we modify the communication in the parallel odd-even sort program so that it is safe?

To answer the first question, we can use an alternative to MPI_Send defined by the MPI standard. It's called MPI_Ssend. The extra "s" stands for *synchronous*, and MPI_Ssend is guaranteed to block until the matching receive starts. So, we can check whether a program is safe by replacing the calls to MPI_Send with calls to MPI_Ssend. If the program doesn't hang or crash when it's run with appropriate input and comm_sz, then the original program was safe. The arguments to MPI_Ssend are the same as the arguments to MPI_Send:

```
int MPI_Ssend(
      void*          msg_buf_p      /* in */,
      int            msg_size       /* in */,
      MPI_Datatype   msg_type       /* in */,
      int            dest           /* in */,
      int            tag            /* in */,
      MPI_Comm       communicator   /* in */);
```

The answer to the second question is that the communication must be restructured. The most common cause of an unsafe program is multiple processes simultaneously first sending to each other and then receiving. Our exchanges with partners is one example. Another example is a "ring pass," in which each process q sends to the process with rank $q + 1$, except that process comm_sz $- 1$ sends to 0:

```
MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0,
      comm);
MPI_Recv(new_msg, size, MPI_INT,
      (my_rank+comm_sz-1) % comm_sz, 0, comm,
      MPI_STATUS_IGNORE);
```

In both settings, we need to restructure the communications so that some of the processes receive before sending. For example, the preceding communications could be restructured as follows:

```
if (my_rank % 2 == 0) {
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0,
         comm);
   MPI_Recv(new_msg, size, MPI_INT,
         (my_rank+comm_sz-1) % comm_sz, 0, comm,
         MPI_STATUS_IGNORE);
} else {
   MPI_Recv(new_msg, size, MPI_INT,
         (my_rank+comm_sz-1) % comm_sz, 0, comm,
         MPI_STATUS_IGNORE);
   MPI_Send(msg, size, MPI_INT, (my_rank+1) % comm_sz, 0,
         comm);
}
```

It's fairly clear that this will work if comm_sz is even. If, say, comm_sz $= 4$, then processes 0 and 2 will first send to 1 and 3, respectively, while processes 1 and 3 will receive from 0 and 2, respectively. The roles are reversed for the next send-receive pairs: processes 1 and 3 will send to 2 and 0, respectively, while 2 and 0 will receive from 1 and 3.

However, it may not be clear that this scheme is also safe if comm_sz is odd (and greater than 1). Suppose, for example, that comm_sz $= 5$. Then Fig. 3.13 shows a possible sequence of events. The solid arrows show a completed communication, and the dashed arrows show a communication waiting to complete.

MPI provides an alternative to scheduling the communications ourselves—we can call the function MPI_Sendrecv:

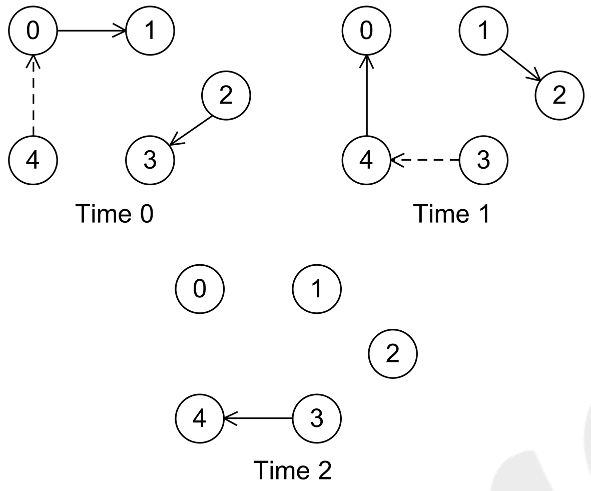

FIGURE 3.13

Safe communication with five processes.

```
int MPI_Sendrecv(
      void*           send_buf_p       /* in   */,
      int             send_buf_size    /* in   */,
      MPI_Datatype    send_buf_type    /* in   */,
      int             dest             /* in   */,
      int             send_tag         /* in   */,
      void*           recv_buf_p       /* out  */,
      int             recv_buf_size    /* in   */,
      MPI_Datatype    recv_buf_type    /* in   */,
      int             source           /* in   */,
      int             recv_tag         /* in   */,
      MPI_Comm        communicator     /* in   */,
      MPI_Status*     status_p         /* in   */);
```

This function carries out a blocking send and a receive in a single call. The `dest` and the `source` can be the same or different. What makes it especially useful is that the MPI implementation schedules the communications so that the program won't hang or crash. The complex code we used above—the code that checks whether the process rank is odd or even—can be replaced with a single call to `MPI_Sendrecv`. If it happens that the send and the receive buffers should be the same, MPI provides the alternative:

```
int MPI_Sendrecv_replace(
      void*           buf_p            /* in/out */,
      int             buf_size         /* in     */,
      MPI_Datatype    buf_type         /* in     */,
      int             dest             /* in     */,
      int            send_tag       /* in      */,
      int            source         /* in      */,
      int            recv_tag       /* in      */,
      MPI_Comm       communicator   /* in      */,
      MPI_Status*    status_p       /* in      */);
```

3.7.4 Final details of parallel odd-even sort

Recall that we had developed the following parallel odd-even transposition sort algorithm:

```
Sort local keys;
for (phase = 0; phase < comm_sz; phase++) {
   partner = Compute_partner(phase, my_rank);
   if (I'm not idle) {
      Send my keys to partner;
      Receive keys from partner;
      if (my_rank < partner)
         Keep smaller keys;
      else
         Keep larger keys;
   }
}
```

In light of our discussion of safety in MPI, it probably makes sense to implement the send and the receive with a single call to `MPI_Sendrecv`:

```
MPI_Sendrecv(my_keys, n/comm_sz, MPI_INT, partner, 0,
      recv_keys, n/comm_sz, MPI_INT, partner, 0, comm,
      MPI_Status_ignore);
```

So it only remains to identify which keys we keep. Suppose for the moment that we want to keep the smaller keys. Then we want to keep the smallest n/p keys in a collection of $2n/p$ keys. An obvious approach to doing this is to sort (using a serial sorting algorithm) the list of $2n/p$ keys, and keep the first half of the list. However, sorting is a relatively expensive operation, and we can exploit the fact that we already have two sorted lists of n/p keys to reduce the cost by *merging* the two lists into a single list. In fact, we can do even better: we don't need a fully general merge; once we've found the smallest n/p keys, we can quit. See Program 3.16.

To get the largest n/p keys, we simply reverse the order of the merge. That is, we start with `local_n`-1 and work backward through the arrays. A final improvement avoids copying the arrays; it simply swaps pointers. (See Exercise 3.28.)

Run-times for the version of parallel odd-even sort with the "final improvement" are shown in Table 3.9. Note that if parallel odd-even sort is run on a single processor, it will use whatever serial sorting algorithm we use to sort the local keys. So the times for a single process use serial quicksort, not serial odd-even sort, which would be *much* slower. We'll take a closer look at these times in Exercise 3.27.

```
void Merge_low(
      int  my_keys[],      /* in/out    */
      int  recv_keys[],    /* in        */
      int  temp_keys[],    /* scratch   */
      int  local_n         /* = n/p, in */) {
   int m_i, r_i, t_i;

   m_i = r_i = t_i = 0;
   while (t_i < local_n) {
      if (my_keys[m_i] <= recv_keys[r_i]) {
         temp_keys[t_i] = my_keys[m_i];
         t_i++; m_i++;
      } else {
         temp_keys[t_i] = recv_keys[r_i];
         t_i++; r_i++;
      }
   }

   for (m_i = 0; m_i < local_n; m_i++)
      my_keys[m_i] = temp_keys[m_i];
}  /* Merge_low */
```

Program 3.16: The `Merge_low` function in parallel odd-even transposition sort.

Table 3.9 Run-times of parallel odd-even sort (times are in milliseconds).

	Number of Keys (in thousands)				
Processes	**200**	**400**	**800**	**1600**	**3200**
1	88	190	390	830	1800
2	43	91	190	410	860
4	22	46	96	200	430
8	12	24	51	110	220
16	7.5	14	29	60	130

3.8 Summary

MPI, or the Message-Passing Interface, is a library of functions that can be called from C or Fortran programs. Many systems use `mpicc` to compile MPI programs and `mpiexec` to run them. C MPI programs should include the `mpi.h` header file to get function prototypes, types, macros, and so on defined by MPI.

`MPI_Init` does the setup needed to run MPI. It should be called before other MPI functions are called. When your program doesn't use `argc` and `argv`, `NULL` can be passed for both arguments.

In MPI a **communicator** is a collection of processes that can send messages to each other. After an MPI program is started, MPI always creates a communicator consisting of all the processes. It's called `MPI_COMM_WORLD`.

Many parallel programs use the **single-program multiple data** or **SPMD** approach: running a single program obtains the effect of running multiple different programs by including branches on data, such as the process rank.

When you're done using MPI, you should call `MPI_Finalize`.

To send a message from one MPI process to another, you can use `MPI_Send`. To receive a message, you can use `MPI_Recv`. The arguments to `MPI_Send` describe the contents of the message and its destination. The arguments to `MPI_Recv` describe the storage that the message can be received into, and where the message should be received from. `MPI_Recv` is **blocking**. That is, a call to `MPI_Recv` won't return until the message has been received (or an error has occurred). The behavior of `MPI_Send` is defined by the MPI implementation. It can either block or **buffer** the message. When it blocks, it won't return until the matching receive has started. If the message is buffered, MPI will copy the message into its own private storage, and `MPI_Send` will return as soon as the message is copied.

When you're writing MPI programs, it's important to differentiate between **local** and **global** variables. Local variables have values that are specific to the process on which they're defined, while global variables are the same on all the processes. In the trapezoidal rule program, the total number of trapezoids, n, was a global variable, while the left and right endpoints of each process's subinterval were local variables.

Most serial programs are **deterministic**. This means that if we run the same program with the same input, we'll get the same output. Recall that parallel programs often don't possess this property—if multiple processes are operating more or less independently, the processes may reach various points at different times, depending on events outside the control of the process. Thus parallel programs can be **nondeterministic**. That is, the same input can result in different outputs. If all the processes in an MPI program are printing output, the order in which the output appears may be different each time the program is run. For this reason, it's common in MPI programs to have a single process (e.g., process 0) handle all the output. This rule of thumb is usually ignored during debugging, when we allow each process to print debug information.

Most MPI implementations allow all the processes to print to `stdout` and `stderr`. However, every implementation we've encountered only allows at most one process (usually process 0 in `MPI_COMM_WORLD`) to read from `stdin`.

Collective communications involve all the processes in a communicator, so they're different from `MPI_Send` and `MPI_Recv`, which only involve two processes. To distinguish between the two types of communications, functions such as `MPI_Send` and `MPI_Recv` are often called **point-to-point** communications.

Two of the most commonly used collective communication functions are `MPI_Reduce` and `MPI_Allreduce`. `MPI_Reduce` stores the result of a global operation (e.g., a global sum) on a single designated process, while `MPI_Allreduce` stores the result on all the processes in the communicator.

In MPI functions, such as `MPI_Reduce`, it may be tempting to pass the same actual argument to both the input and output buffers. This is called **argument aliasing**, and MPI explicitly prohibits aliasing an output argument with another argument.

We learned about a number of other important MPI collective communications:

- `MPI_Bcast` sends data from a single process to all the processes in a communicator. This is very useful if, for example, process 0 reads data from `stdin` and the data needs to be sent to all the processes.
- `MPI_Scatter` distributes the elements of an array among the processes. If the array to be distributed contains n elements, and there are p processes, then the first n/p are sent to process 0, the next n/p to process 1, and so on.
- `MPI_Gather` is the "inverse operation" to `MPI_Scatter`. If each process stores a subarray containing m elements, `MPI_Gather` will collect all of the elements on a designated process, putting the elements from process 0 first, then the elements from process 1, and so on.
- `MPI_Allgather` is like `MPI_Gather`, except that it collects all of the elements on *all* the processes.
- `MPI_Barrier` approximately synchronizes the processes; no process can return from a call to `MPI_Barrier` until all the processes in the communicator have started the call.

In distributed-memory systems there is no globally shared memory, so partitioning global data structures among the processes is a key issue in writing MPI programs. For ordinary vectors and arrays, we saw that we could use block partitioning, cyclic partitioning, or block-cyclic partitioning. If the global vector or array has n components and there are p processes, a **block partition** assigns the first n/p to process 0, the next n/p to process 1, and so on. A **cyclic partition** assigns the elements in a "round-robin" fashion: the first element goes to 0, the next to 1, ... the pth to $p-1$. After assigning the first p elements, we return to process 0, so the $(p+1)$st goes to process 0, the $(p+2)$nd to process 1, and so on. A **block-cyclic partition** assigns blocks of elements to the processes in a cyclic fashion.

Compared to operations involving only the CPU and main memory, sending messages is expensive. Furthermore, sending a given volume of data in fewer messages is usually less expensive than sending the same volume in more messages. Thus it often makes sense to reduce the number of messages sent by combining the contents of multiple messages into a single message. MPI provides three methods for doing this: the `count` argument to communication functions, derived datatypes, and `MPI_Pack`/`Unpack`. Derived datatypes describe arbitrary collections of data by specifying the types of the data items and their relative positions in memory. In this chapter, we took a brief look at the use of `MPI_Type_create_struct` to build a derived datatype. In the exercises, we'll explore some other methods, and we'll take a look at `MPI_Pack`/`Unpack`

When we time parallel programs, we're usually interested in elapsed time or "wall clock time," which is the total time taken by a block of code. It includes time in user code, time in library functions, time in operating system functions started by the user code, and idle time. We learned about two methods for finding wall clock time. GET_TIME and MPI_Wtime. GET_TIME is a macro defined in the file timer.h that can be downloaded from the book's website. It can be used in serial code as follows:

```
#include "timer.h"  // From the book's website
. . .
double start, finish, elapsed;
. . .
GET_TIME(start);
/* Code to be timed */
. . .
GET_TIME(finish);
elapsed = finish − start;
printf("Elapsed time = %e seconds\n", elapsed);
```

MPI provides a function, MPI_Wtime, that can be used instead of GET_TIME. In spite of this, timing parallel code is more complex, since—ideally—we'd like to synchronize the processes at the start of the code, and then report the time it took for the "slowest" process to complete the code. MPI_Barrier does a fairly good job of synchronizing the processes. A process that calls it will block until all the processes in the communicator have called it. We can use the following template for finding the run-time of MPI code:

```
double start, finish, loc_elapsed, elapsed;
. . .
MPI_Barrier(comm);
start = MPI_Wtime();
/* Code to be timed */
. . .
finish = MPI_Wtime();
loc_elapsed = finish − start;
MPI_Reduce(&loc_elapsed, &elapsed, 1, MPI_DOUBLE,
      MPI_MAX, 0, comm);
if (my_rank == 0)
   printf("Elapsed time = %e seconds\n", elapsed);
```

A further problem with taking timings lies in the fact that there is ordinarily considerable variation if the same code is timed repeatedly. For example, the operating system may idle one or more of our processes so that other processes can run. Therefore we typically take several timings and report their minimum.

After taking timings, we can use the **speedup** or the **efficiency** to evaluate the program performance. The speedup is the ratio of the serial run-time to the parallel run-time, and the efficiency is the speedup divided by the number of parallel processes. The ideal value for speedup is p, the number of processes, and the ideal value for the efficiency is 1.0. We often fail to achieve these ideals, but it's not uncommon to see programs that get close to these values, especially when p is small and n, the problem size, is large. **Parallel overhead** is the part of the parallel run-time that's due to any additional work that isn't done by the serial program. In MPI programs, parallel overhead will come from communication. When p is large and n is small, it's not unusual for parallel overhead to dominate the total run-time, and speedups and efficiencies can be quite low.

If it's possible to increase the problem size (n) so that the efficiency doesn't decrease as p is increased, a parallel program is said to be **scalable**.

Recall that `MPI_Send` can either block or buffer its input. An MPI program is **unsafe** if its correct behavior depends on the fact that `MPI_Send` is buffering its input. This typically happens when multiple processes first call `MPI_Send` and then call `MPI_Recv`. If the calls to `MPI_Send` don't buffer the messages, then they'll block until the matching calls to `MPI_Recv` have started. However, this will never happen. For example, if process 0 and process 1 are trying to exchange data, and both call `MPI_Send` before calling `MPI_Recv`, then each process will wait forever for the other process to call `MPI_Recv`, since both will block in their calls to `MPI_Send`. That is, the processes will hang or **deadlock**—they'll block forever waiting for events that will never happen.

An MPI program can be checked for safety by replacing each call to `MPI_Send` with a call to `MPI_Ssend`. `MPI_Ssend` takes the same arguments as `MPI_Send`, but it always blocks until the matching receive has started. The extra "s" stands for *synchronous*. If the program completes correctly with `MPI_Ssend` for the desired inputs and communicator sizes, then the program is safe.

An unsafe MPI program can be made safe in several ways. The programmer can schedule the calls to `MPI_Send` and `MPI_Recv` so that some processes (e.g., even-ranked processes) first call `MPI_Send`, while others (e.g., odd-ranked processes) first call `MPI_Recv`. Alternatively, we can use `MPI_Sendrecv` or `MPI_Sendrecv_replace`. These functions execute both a send and a receive, but they're guaranteed to schedule them so that the program won't hang or crash. `MPI_Sendrecv` uses different arguments for the send and the receive buffers, while `MPI_Sendrecv_replace` uses the same buffer for both.