CHAPTER 5

Collective Communication

THERE ARE PROBABLY A FEW THINGS in the trapezoidal rule program that you think we can improve on. For example, there is the I/O issue. There are also a couple of problems we haven't discussed yet. Let's think about what happens when the program is run with eight processes.

All the processes begin executing the program (more or less) simultaneously. However, after carrying out the basic setup tasks (calls to `MPI_Init`, `MPI_Comm_size`, and `MPI_Comm_rank`), processes 1–7 are idle while process 0 collects the input data. We don't want to have idle processes, but in view of our restrictions on which processes can read input, there isn't much we can do about this. However, after process 0 has collected the input data, the higher rank processes must continue to wait while 0 sends the input data to the lower rank processes. This isn't just an I/O issue. Notice that there is a similar inefficiency at the end of the program, when process 0 does all the work of collecting and adding. Of course, this is highly undesirable: the main point of parallel computing is to get multiple processes to collaborate on solving a problem. If one of the processes is doing most of the work, we might as well use a conventional, single-processor machine.

5.1 Tree-Structured Communication

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

During the first stage of the data distribution, 0 sends the data to, say, 1. During the next stage, 0 sends the data to 2, while 1 sends it to 3. During the

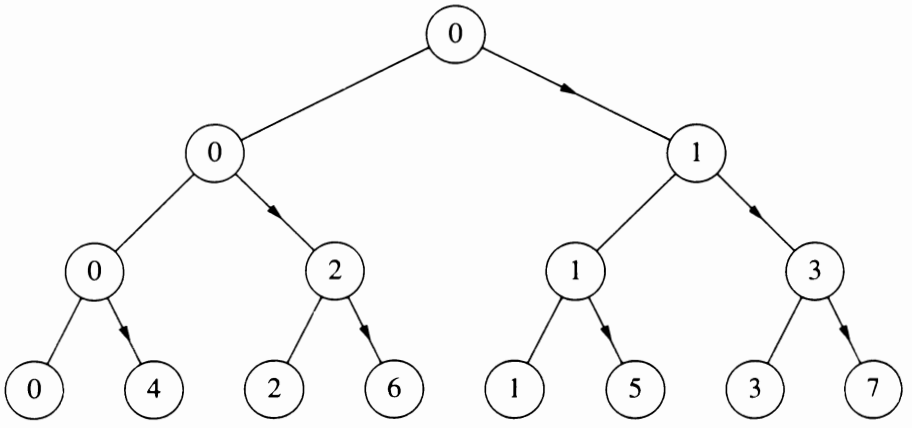

Figure 5.1 Processes configured as a tree

last stage, 0 sends to 4, while 1 sends to 5, 2 sends to 6, and 3 sends to 7. (See Figure 5.1.) So we have reduced our input distribution loop from seven stages to three stages. More generally, if we have p processes, this procedure allows us to distribute the input data in $\lceil \log_2(p) \rceil$ stages,[1] rather than $p - 1$ stages. This can be a huge savings. For example, $\log_2(1024) = 10$. That is, if we use this tree-structured scheme with 1024 processes, the time required for our program to complete the data distribution will be reduced by a factor of 100!

Let's modify our `Get_data` function to use a tree-structured distribution scheme. Essentially, the distribution is a loop:

```
for (stage = first; stage <= last; stage++)
    if (I_receive(stage, my_rank, &source))
        Receive(data, source);
    else if (I_send(stage, my_rank, p, &dest))
        Send(data, dest);
```

The `I_receive` function returns 1 if, during the current stage, the calling process receives data. Otherwise it returns 0. If the calling process receives data, the parameter `source` is used to return the rank of the sender. The `I_send` function performs a similar function, testing whether a process sends during the current stage.

In order to implement this code, we need to calculate

- whether a process receives and, if so, the source, and
- whether a process sends and, if so, the destination.

As you can probably guess, these calculations can be a bit complicated, es-

1 The **ceiling** of a real number x is the smallest whole number greater than or equal to x. It is denoted by $\lceil x \rceil$.

pecially since there is no canonical choice of ordering. In our example, we chose

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

We might also have chosen (for example)

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

Indeed, unless we know something about the topology of our system, we can't really decide which scheme is better. However, it's fairly easy to see how to calculate the required information using the first scheme, so let's use it. Let's also use the C convention that counting begins at 0. So the first stage is stage 0, the second, stage 1, etc. If

$$2^{\texttt{stage}} \leq \texttt{my\_rank} < 2^{\texttt{stage}+1},$$

then I receive from

$$\texttt{my\_rank} - 2^{\texttt{stage}}.$$

If

$$\texttt{my\_rank} < 2^{\texttt{stage}},$$

then I send to

$$\texttt{my\_rank} + 2^{\texttt{stage}}.$$

To summarize, our second version of the `Get_data` function, together with the functions it calls, should look something like this:

```
/* Ceiling of log_2(x) is just the number of
 * times x-1 can be divided by 2 until the quotient
 * is 0.  Dividing by 2 is the same as right shift.
 */
int Ceiling_log2(int  x  /* in */) {
    /* Use unsigned so that right shift will fill
     * leftmost bit with 0
     */
    unsigned temp = (unsigned) x - 1;
    int result = 0;

    while (temp != 0) {
         temp = temp >> 1;
         result = result + 1 ;
    }
```

```
    return result;
} /* Ceiling_log2 */

int I_receive(
        int   stage       /* in  */,
        int   my_rank     /* in  */,
        int*  source_ptr  /* out */) {

    int   power_2_stage;

    /* 2^stage = 1 << stage */
    power_2_stage = 1 << stage;
    if ((power_2_stage <= my_rank) &&
            (my_rank < 2*power_2_stage)){
        *source_ptr = my_rank - power_2_stage;
        return 1;
    } else return 0;
} /* I_receive */

int I_send(
        int   stage    /* in  */,
        int   my_rank  /* in  */,
        int   p        /* in  */,
        int*  dest_ptr /* out */) {
    int power_2_stage;

    /* 2^stage = 1 << stage */
    power_2_stage = 1 << stage;
    if (my_rank < power_2_stage){
        *dest_ptr = my_rank + power_2_stage;
        if (*dest_ptr >= p) return 0;
        else return 1;
    } else return 0;
} /* I_send */

void Send(
        float  a     /* in */,
        float  b     /* in */,
        int    n     /* in */,
        int    dest  /* in */) {

    MPI_Send(&a, 1, MPI_FLOAT, dest, 0, MPI_COMM_WORLD);
    MPI_Send(&b, 1, MPI_FLOAT, dest, 1, MPI_COMM_WORLD);
    MPI_Send(&n, 1, MPI_INT, dest, 2, MPI_COMM_WORLD);
} /* Send */

void Receive(
        float*  a_ptr  /* out */,
        float*  b_ptr  /* out */,
```

```
        int*    n_ptr  /* out */,
        int     source /* in  */) {

    MPI_Status status;

    MPI_Recv(a_ptr, 1, MPI_FLOAT, source, 0,
        MPI_COMM_WORLD, &status);
    MPI_Recv(b_ptr, 1, MPI_FLOAT, source, 1,
        MPI_COMM_WORLD, &status);
    MPI_Recv(n_ptr, 1, MPI_INT, source, 2,
        MPI_COMM_WORLD, &status);
} /* Receive */

void Get_data1(
        float*  a_ptr    /* out */,
        float*  b_ptr    /* out */,
        int*    n_ptr    /* out */,
        int     my_rank  /* in  */,
        int     p        /* in  */) {

    int source;
    int dest;
    int stage;

    if (my_rank == 0){
        printf("Enter a, b, and n\n");
        scanf("%f %f %d", a_ptr, b_ptr, n_ptr);
    }
    for (stage = 0; stage < Ceiling_log2(p); stage++)
        if (I_receive(stage, my_rank, &source))
            Receive(a_ptr, b_ptr, n_ptr, source);
        else if (I_send(stage, my_rank, p, &dest))
            Send(*a_ptr, *b_ptr, *n_ptr, dest);
} /* Get_data1*/
```

5.2 Broadcast

A communication pattern that involves all the processes in a communicator is a **collective communication.** As a consequence, a collective communication usually involves more than two processes. A **broadcast** is a collective communication in which a single process sends the same data to every process in the communicator. As we saw in section 5.1, a tree-structured broadcast can be much more efficient than a broadcast based on a sequence of sends all originating from the **root** of the broadcast. However, we also saw that the details involved in writing the code to carry out this tree-structured broadcast were fairly complicated.

Furthermore, without knowing the details of the topology of the system we're using, we can't be sure that our hand-coded broadcast is the most efficient broadcast possible. So we would like to have someone with detailed knowledge of our system write an optimized broadcast that we can use. Not surprisingly, MPI has allowed for this: each system that runs MPI has a broadcast function, `MPI_Bcast`. Since MPI only specifies the syntax of the function call and the result of calling it, it's possible for implementors to write a highly optimized function.

The syntax of `MPI_Bcast` is

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

It simply sends a copy of the data in `message` on the process with rank `root` to each process in the communicator `comm`. It should be called by *all* the processes in the communicator with the same arguments for `root` and `comm`. Hence a broadcast message cannot be received with `MPI_Recv`. The parameters `count` and `datatype` have the same function that they have in `MPI_Send` and `MPI_Recv`: they specify how much memory is needed for the message. However, unlike the point-to-point functions, in most cases `count` and `datatype` should be the same on all the processes in the communicator. The reason for this is that in some collective operations (see below), a single process will receive data from many other processes, and in order for a program to determine how much data has been received, it would need an array of return statuses. Since there is no tag in collective communication functions, making `count` and `datatype` the same on all processes removes the need for a status parameter.

Note that in the syntax specification, `message` is identified as an `in/out` parameter. In conventional serial programs, `in/out` parameters have values that are both used and modified by the function. In parallel programs, another interpretation can be attached to `in/out` parameters of collective communication functions: on some processes the parameter may be `in`, while on others it may be `out`. In the case of `MPI_Bcast`, `message` is `in` on the process with rank `root`, while it is `out` on the other processes.

Just to make sure that we understand how to use `MPI_Bcast`, let's rewrite `Get_data`.

```
void Get_data2(
        float*  a_ptr    /* out */,
        float*  b_ptr    /* out */,
        int*    n_ptr    /* out */,
        int     my_rank  /* in  */) {
```

Table 5.1 Broadcast times (times are in milliseconds; version 1 uses a linear loop of sends from process 0, version 2 uses `MPI_Bcast`; all systems running `mpich`)

	nCUBE2		*Paragon*		*SP2*	
Processes	*Version 1*	*Version 2*	*Version 1*	*Version 2*	*Version 1*	*Version 2*
2	0.59	0.69	0.21	0.43	0.15	0.16
8	4.7	1.9	0.84	0.93	0.55	0.35
32	19.0	3.0	3.2	1.3	2.0	0.57

Table 5.2 Send/receive sequence of events

Time	*Process A*	*Process B*
1	`MPI_Send` to *B*, tag = 0	Local work
2	`MPI_Send` to *B*, tag = 1	Local work
3	Local work	`MPI_Recv` from *A*, tag = 1
4	Local work	`MPI_Recv` from *A*, tag = 0

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

Certainly this version of `Get_data` is much more compact and readily comprehensible than both the original and our hand-coded tree-structured broadcast. Furthermore, it's probably a good deal faster. Table 5.1 shows the runtimes of the data distribution phase of our first and third versions (`Get_data` and `Get_data2`).

5.3 Tags, Safety, Buffering, and Synchronization

Note that `MPI_Bcast` (and all the other collective communication functions in MPI) do not use tags. In order to understand the reasons for this, let's look at a couple of examples.

Recall our example from Chapter 3 in which we discussed the need for tags in `MPI_Send` and `MPI_Recv`. In that example, process *A* sent several messages to process *B*, and *B* decided how to handle these messages on the basis of the tag. In particular, consider the sequence of events outlined in Table 5.2. Note that this sequence of events requires that the system **buffer** the messages being sent to process *B*. That is, memory must be set aside for

Table 5.3 Broadcast sequence of events

Time	*Process A*	*Process B*	*Process C*
1	`MPI_Bcast x`	Local work	Local work
2	`MPI_Bcast y`	Local work	Local work
3	Local work	`MPI_Bcast y`	`MPI_Bcast x`
4	Local work	`MPI_Bcast x`	`MPI_Bcast y`

storing messages before a receive has been executed. Recall that the message envelope contains

1. The rank of the receiver
2. The rank of the sender
3. A tag
4. A communicator

There is no information specifying where the message should be stored by the receiving process. Thus, until *B* calls `MPI_Recv`, the system doesn't know where the data that *A* is sending should be stored. When *B* calls `MPI_Recv`, the system software that executes the receive can see which (if any) buffered message has an envelope that matches the parameters specified by the receive. If there isn't a message, it waits until one arrives.

If a system has no buffering, then *A* cannot send its data until it knows that *B* is ready to receive the message, and consequently memory is available for the data. When a send cannot complete until the receiver is ready to receive the message, the send is said to use **synchronous mode.**

Programming under the assumption that a system has buffering is very common (most systems automatically provide some buffering), although in MPI parlance, it is **unsafe**. This means that if the program is run on a system that does not provide buffering, the program will **deadlock**. If the system has no buffering, *A*'s first message cannot be received until *B* has signalled that it is ready to receive the data with tag 0. So *A* will hang while it waits for *B* to receive the first send, and *B* will hang while it waits for *A* to execute the second send.

Now consider a somewhat analogous example that uses broadcasts. Suppose we have three processes, *A*, *B*, and *C*, and *A* is broadcasting two floats, `x` and `y`, to *B* and *C*. Also suppose that on process *A*, `x = 5` and `y = 10`. See Table 5.3. When the broadcasts are completed on all three processes, `x = 5` and `y = 10` on processes *A* and *C*. However, on process *B* the values will be reversed: `x = 10` and `y = 5`. Why? When people first started programming parallel processors, broadcasts (and all other collective communication functions) were points of **synchronization**. On a given process the broadcast would not return until every process had received the broadcast data. On current systems this restriction has been relaxed. If the system has adequate

buffering, it's OK for A to complete (from its point of view) two broadcasts before the other processes even begin their calls. However, in terms of the data communicated, the *effect* must be the same as if the processes synchronized. So the first `MPI_Bcast` on B matches the first `MPI_Bcast` on A, and B stores the first value it receives, 5, in `y`.

At first this may seem a little confusing. However, its net effect is that a sequence of collective communications on distinct processes will be matched in the order in which they're executed.

5.4 Reduce

In the trapezoidal rule program after the input phase, every process executes essentially the same commands until the final summation phase. So unless our function $f(x)$ is fairly complicated (i.e., it requires considerably more work to evaluate over certain parts of $[a, b]$), this part of the program distributes the work equally among the processes. As we have already noted, this is not the case with the final summation phase, when, once again, process 0 gets a disproportionate amount of the work. However, you have probably already noticed that by reversing the arrows in Figure 5.1, we can use the same idea we used in section 5.1. That is, we can distribute the work of calculating the sum among the processes as follows:

1. a. 4 sends to 0; 5 sends to 1; 6 sends to 2; 7 sends to 3.
   b. 0 adds its integral to that of 4; 1 adds its integral to that of 5; etc.
2. a. 2 sends to 0; 3 sends to 1.
   b. 0 adds; 1 adds.
3. a. 1 sends to 0.
   b. 0 adds.

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

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

```
int MPI_Reduce(
        void*         operand   /* in  */,
        void*         result    /* out */,
        int           count     /* in  */,
```

Table 5.4 Predefined reduction operators in MPI

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

```
MPI_Datatype  datatype  /* in  */,
MPI_Op        operator  /* in  */,
int           root      /* in  */,
MPI_Comm      comm      /* in  */)
```

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

The parameter operator can take on one of the predefined values listed in Table 5.4. (It is also possible to define additional operations. See the MPI Standard for details [28].)

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

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

/* Print the result */
        .
        .
        .
```

Note that each processor calls MPI_Reduce with the same arguments. In particular, even though total only has significance on process 0, each process must supply an argument. This makes sense; otherwise, the syntax would be

different on the root process, and this would create the impression that there were distinct function calls, rather than a single collective call.

It may be tempting to pass the same argument to both `operand` and `result` on the process with rank root. For example, we might try to call the function with

```
/* Attempt to store the result in the same
 * location as the operand.  Illegal call.
 */
MPI_Reduce(&integral, &integral, 1, MPI_FLOAT,
    MPI_SUM, 0, MPI_COMM_WORLD);
```

This is called **aliasing** of arguments, and it is illegal to alias `out` or `in/out` arguments in *any* MPI function. The main reason is that without the extra argument, an implementation may be forced to provide large temporary buffers. For an exploration of this problem, take a look at programming assignment 2.

5.5 Dot Product

As another example of the application of `MPI_Reduce`, let's write a function that calculates the dot product of two vectors.

Recall that if $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})^T$ and $\mathbf{y} = (y_0, y_1, \ldots, y_{n-1})^T$ are n-dimensional vectors of real numbers,[2] then their **dot product** is just

$$\mathbf{x} \cdot \mathbf{y} = x_0 y_0 + x_1 y_1 + \cdots + x_{n-1} y_{n-1}.$$

So on a conventional system, we can form the dot product as follows:

```
float Serial_dot(
          float  x[]  /* in */,
          float  y[]  /* in */,
          int    n    /* in */) {

    int    i;
    float  sum = 0.0;

    for (i = 0; i < n; i++)
        sum = sum + x[i]*y[i];
    return sum;
} /* Serial_dot */
```

If we have p processes, and n is divisible by p, it's natural to divide the vectors among the processes so that each process has $\bar{n} = n/p$ components

2 The transpose of $\mathbf{x}$ is denoted $\mathbf{x}^T$. We adhere to the convention that vectors are *column* vectors. We also use the C convention that arrays are indexed from 0 to $n-1$, rather than the more common 1 to n.

Table 5.5 Block mapping of the vector $\mathbf{x}$ to the processes

Process	*Components*
0	$x_0, x_1, \ldots, x_{\bar{n}-1}$
1	$x_{\bar{n}}, x_{\bar{n}+1}, \ldots, x_{2\bar{n}-1}$
⋮	⋮
k	$x_{k\bar{n}}, x_{k\bar{n}+1}, \ldots, x_{(k+1)\bar{n}-1}$
⋮	⋮
$p-1$	$x_{(p-1)\bar{n}}, x_{(p-1)\bar{n}+1}, \ldots, x_{n-1}$

of each of the vectors. For the sake of explicitness, let's suppose that we use a **block** distribution of the data. Recall (from section 2.2.5) that this means that the vector $\mathbf{x}$ is distributed as indicated in Table 5.5. A similar distribution would be used for $\mathbf{y}$.

With this distribution of the data, we can calculate a "local" dot product, by just calling `Serial_dot`, and then calling `MPI_Reduce` to get the "global" dot product.

```
float Parallel_dot(
          float  local_x[]  /* in */,
          float  local_y[]  /* in */,
          int    n_bar      /* in */) {

    float  local_dot;
    float  dot = 0.0;
    float  Serial_dot(float x[], float y[], int m);

    local_dot = Serial_dot(local_x, local_y, n_bar);
    MPI_Reduce(&local_dot, &dot, 1, MPI_FLOAT,
        MPI_SUM, 0, MPI_COMM_WORLD);
    return dot;
} /* Parallel_dot */
```

5.6 Allreduce

Note that in our `Parallel_dot` function only process 0 will return the dot product. The other processes will return 0. If we only wish, for example, to print the result, this won't be a problem. However, if we want to use the result in subsequent calculations, we may want each process to return the correct dot product. An obvious approach to doing this is to follow the call to `MPI_Reduce` with a call to `MPI_Bcast`. However, it may be possible to use a more efficient implementation.

Consider the following approach to calculating the sum that will store the

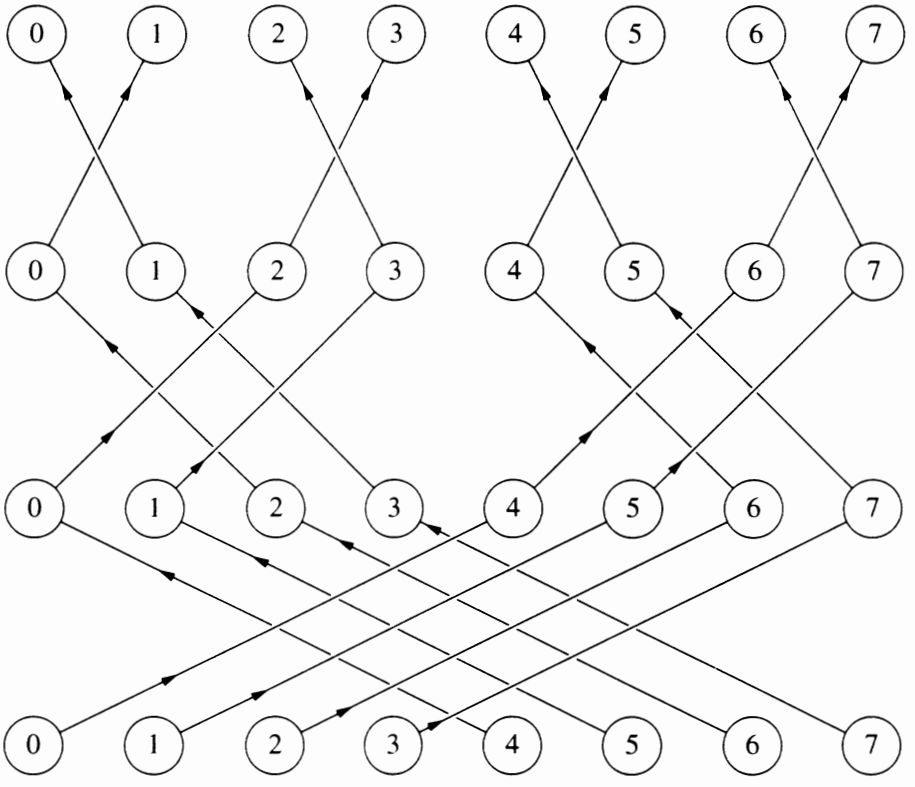

Figure 5.2 A butterfly

result on all the processes. Once again, for the sake of explicitness, let's assume that we have eight processes.

1. a. Processes 0 and 4 exchange their local results, processes 1 and 5 exchange theirs, processes 2 and 6 exchange, and processes 3 and 7 exchange.
   b. Each process adds its result to the result just received.
2. a. Processes 0 and 2 exchange their intermediate results, processes 1 and 3 exchange theirs, processes 4 and 6 exchange, and processes 5 and 7 exchange.
   b. Each process adds.
3. a. Processes 0 and 1 exchange, processes 2 and 3 exchange, processes 4 and 5 exchange, and processes 6 and 7 exchange.
   b. Each process adds.

What's happened? Essentially we've done a tree-structured reduce rooted at all the processes simultaneously! Figure 5.2 illustrates the process. This communication structure is frequently called a **butterfly**. If you add vertical lines joining processes of the same rank in adjacent rows, you'll see that it contains a tree rooted at each process.

Of course, we don't want to have to write this communication scheme ourselves, so let's let MPI take care of it.

```
int MPI_Allreduce(
        void*          operand   /* in  */,
        void*          result    /* out */,
        int            count     /* in  */,
        MPI_Datatype   datatype  /* in  */,
        MPI_Op         operator  /* in  */,
        MPI_Comm       comm      /* in  */)
```

It is used in exactly the same way as `MPI_Reduce`. The only difference is that the result of the reduction is returned in `result` on *all* the processes. Hence there is no `root` parameter.

In order to use it in `Parallel_dot`, we can simply replace the call to `MPI_Reduce` with a call to `MPI_Allreduce`.

5.7 Gather and Scatter

Let's take a look at how we might implement another linear algebra operation: matrix-vector product. Recall that if $A = (a_{ij})$ is an $m \times n$ matrix, and $\mathbf{x} = (x_0, \ldots, x_{n-1})^T$ is an n-dimensional vector, then we can form the matrix-vector product $\mathbf{y} = A\mathbf{x}$ by taking the dot product of each row of A with $\mathbf{x}$. If A has m rows, then we will form m dot products. So the product vector $\mathbf{y}$ will consist of m entries: $\mathbf{y} = (y_0, y_1, \ldots, y_{m-1})$ and

$$y_k = a_{k0}x_0 + a_{k1}x_1 + \cdots + a_{k,n-1}x_{n-1}.$$

Thus, a serial function for forming a matrix-vector product might look something like this:

```
/* MATRIX_T is a two-dimensional array of floats */
void Serial_matrix_vector_prod(
        MATRIX_T  A    /* in  */,
        int       m    /* in  */,
        int       n    /* in  */,
        float     x[]  /* in  */,
        float     y[]  /* out */) {

    int k, j;

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

In order to parallelize this, we must decide how the matrices and vectors are to be distributed among the processes. One of the simplest matrix distributions is a **block-row** or **panel** distribution. In this distribution, we partition the

Table 5.6 Block-row distribution

Process	*Elements of A*			
0	a_{00}	a_{01}	a_{02}	a_{03}
	a_{10}	a_{11}	a_{12}	a_{13}
1	a_{20}	a_{21}	a_{22}	a_{23}
	a_{30}	a_{31}	a_{32}	a_{33}
2	a_{40}	a_{41}	a_{42}	a_{43}
	a_{50}	a_{51}	a_{52}	a_{53}
3	a_{60}	a_{61}	a_{62}	a_{63}
	a_{70}	a_{71}	a_{72}	a_{73}

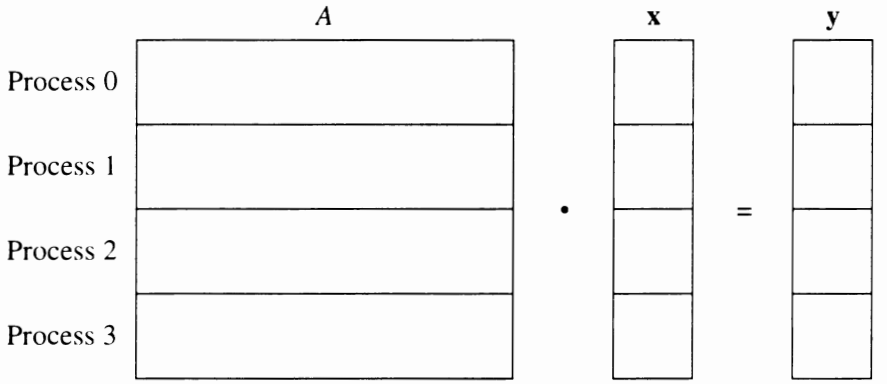

Figure 5.3 Mappings of A, $\mathbf{x}$, and $\mathbf{y}$ for matrix-vector product

matrix into blocks of consecutive rows or panels, and assign a panel to each process. For example, if $m = 8, n = 4$, and $p = 4$, the assignment of matrix elements to the processes is illustrated in Table 5.6. Let's use our familiar block distribution for the vectors (see Table 5.5). A schematic illustration of all the mappings (with four processes) is contained in Figure 5.3.

Now in order to form the dot product of each row of A with $\mathbf{x}$, we need to either **gather** all of $\mathbf{x}$ onto each process or **scatter** each row of A across the processes. For example, suppose, $m = n = p = 4$. Then, before we form the dot products, $a_{00}, a_{01}, a_{02}, a_{03}$, and x_0 are assigned to process 0, while x_1 is assigned to process 1, x_2 is assigned to process 2, and x_3 is assigned to process 3. So in order to form the dot product of the first row of A with $\mathbf{x}$, we can either send x_1, x_2, and x_3 to process 0, or we can send a_{01} to process 1, a_{02} to process 2, and a_{03} to process 3. The first collection of sends is called a gather. The second is called a scatter. See Figures 5.4 and 5.5.

In view of our previous discussion, it isn't surprising that MPI provides both a gather and a scatter function. For example, we can gather $\mathbf{x}$ onto process 0 with the call

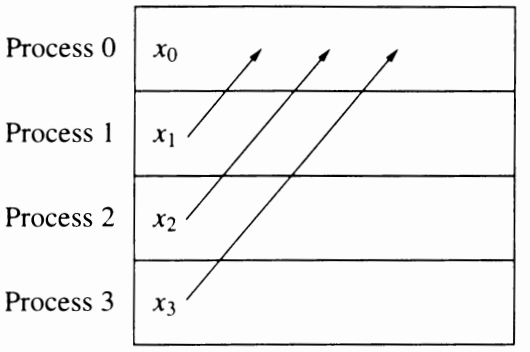

Figure 5.4 A gather

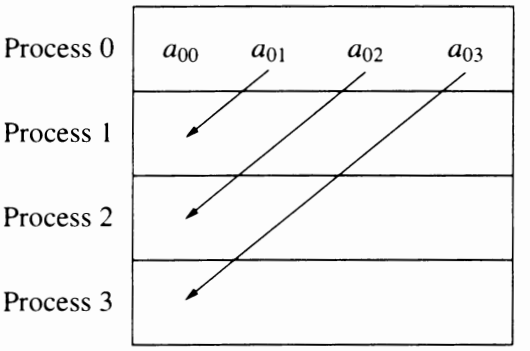

Figure 5.5 A scatter

```
/* Space allocated in calling program     */
float local_x[];  /* local storage for x  */
float global_x[]; /* storage for all of x */

/* Assumes n is divisible by p */
MPI_Gather(local_x, n/p, MPI_FLOAT,
           global_x, n/p, MPI_FLOAT,
           0, MPI_COMM_WORLD);
```

The exact syntax of `MPI_Gather` is

```
int MPI_Gather(
        void*         send_data    /* in  */,
        int           send_count   /* in  */,
        MPI_Datatype  send_type    /* in  */,
        void*         recv_data    /* out */,
        int           recv_count   /* in  */,
        MPI_Datatype  recv_type    /* in  */,
        int           root         /* in  */,
        MPI_Comm      comm         /* in  */)
```

MPI_Gather collects the data referenced by send_data from each process in the communicator comm and stores the data in process rank order on the process with rank root in the memory referenced by recv_data. Thus, the data from process 0 is followed by the data from process 1, which is followed by the data from process 2, etc. The parameters send_count and send_type have their usual interpretation: the data referenced by send_data on each process consists of send_count elements, each of which has type send_type. The parameters recv_count and recv_type will, in most cases, be the same as send_count and send_type, respectively. They specify the number of elements and type of elements received from *each* process. They do not specify the total amount of data received. The recv parameters are only significant on the root process. The parameters root and comm must be identical on all the processes in the communicator comm. In most cases, the parameters send_count and send_type will be the same on all the processes.

We can scatter the first row of A among the processes using MPI_Scatter:

```
/* Both arrays allocated by calling program       */
LOCAL_MATRIX_T local_A; /* A 2-dimensional array  */
float          row_segment[];
                        /* An array containing    */
                        /* storage for n/p floats */

/* Assumes n is divisible by p */
MPI_Scatter(&(local_A[0][0]), n/p, MPI_FLOAT,
            row_segment, n/p, MPI_FLOAT,
            0, MPI_COMM_WORLD);
```

The syntax of MPI_Scatter is

```
int MPI_Scatter(
        void*         send_data    /* in  */,
        int           send_count   /* in  */,
        MPI_Datatype  send_type    /* in  */,
        void*         recv_data    /* out */,
        int           recv_count   /* in  */,
        MPI_Datatype  recv_type    /* in  */,
        int           root         /* in  */,
        MPI_Comm      comm         /* in  */)
```

MPI_Scatter splits the data referenced by send_data on the process with rank root into p segments, each of which consists of send_count elements of type send_type. The first segment is sent to process 0, the second to process 1, etc. The send parameters are significant only on the process with rank root. In most cases send_count will be the same as recv_count and send_type will be the same as recv_type. The parameters root and comm must be the same on all processes.

Table 5.7 Products of elements of first row with elements of **x** if first row is scattered

Process	*Product*
0	$a_{00}x_0$
1	$a_{01}x_1$
2	$a_{02}x_2$
3	$a_{03}x_3$

5.8 Allgather

We still need to decide how to carry out our parallel matrix-vector multiplication! Observe that if we gather **x** onto each process and form the dot product of each local row of A with **x**, no additional communication is needed. The kth element of **y** is assigned to the same process as the kth row of A, and the kth element of **y** is computed by forming the dot product of the kth row of A with x.

On the other hand, suppose we scatter each row of A, and, for the sake of explicitness, suppose $m = n = p = 4$. Then after the scatter of row 0 (for example), process q will compute $a_{0q}x_q$ (see Table 5.7). That is, the individual terms in the dot product will be assigned to different processes. Thus, we'll need to call `MPI_Reduce` in order to finish the dot product.

Thus, it appears that by scattering the rows of A, we'll have to do considerably more communication, and, as a consequence, we should gather **x** onto each process. The obvious approach to carrying this out would be to have each process call `MPI_Gather` p times: once to gather **x** onto 0, once to gather **x** onto 1, etc.:

```
for (root = 0; root < p; root++)
    MPI_Gather(local_x, n/p, MPI_FLOAT,
        global_x, n/p, MPI_FLOAT,
        root, MPI_COMM_WORLD);
```

However, a little thought will convince you that we can use the butterfly scheme to simultaneously gather all of **x** onto each process. Thus, MPI should provide a more efficient solution to the problem of gathering a distributed array to every process, and, not surprisingly, it does. The function is called `MPI_Allgather`:

```
int MPI_Allgather(
        void*         send_data   /* in  */,
        int           send_count  /* in  */,
        MPI_Datatype  send_type   /* in  */,
        void*         recv_data   /* out */,
        int           recv_count  /* in  */,
        MPI_Datatype  recv_type   /* in  */,
        MPI_Comm      comm        /* in  */)
```

It gathers the contents of each process's `send_data` into each process's `recv_data`.

At last, we can write the parallel matrix-vector product function.

```
/* All arrays are allocated in calling program */
void Parallel_matrix_vector_prod(
         LOCAL_MATRIX_T  local_A     /* in  */,
         int             m           /* in  */,
         int             n           /* in  */,
         float           local_x[]   /* in  */,
         float           global_x[]  /* in  */,
         float           local_y[]   /* out */,
         int             local_m     /* in  */,
         int             local_n     /* in  */) {

   /* local_m = n/p, local_n = n/p */

   int i, j;

   MPI_Allgather(local_x, local_n, MPI_FLOAT,
                 global_x, local_n, MPI_FLOAT,
                 MPI_COMM_WORLD);
   for (i = 0; i < local_m; i++) {
       local_y[i] = 0.0;
       for (j = 0; j < n; j++)
           local_y[i] = local_y[i] +
                        local_A[i][j]*global_x[j];
   }
}  /* Parallel_matrix_vector_prod */
```

5.9 Summary

Collective communication is communication that involves all the processes in a communicator. It usually involves more than two processes. We looked at several different types of collective communication. In a *broadcast*, a single process sends the same data to all the other processes in a communicator. In a *reduction*, each process in a communicator contains an operand, and all of them are combined using a binary operator that is successively applied to each. The reduction operation that we studied was sum: the reduction adds a collection of numbers distributed across the processes. In a *gather*, a distributed data structure is collected onto a single process. In a *scatter*, a data structure that is stored on a single process is distributed across the processes.

We saw that the performance of both a broadcast and a reduce can be greatly improved if the processes are viewed as a tree, and the communication proceeds along the branches of the tree. Constructing an optimal tree depends on the topology of the parallel system, and, as a consequence, the internals of such commands should be system-dependent.

The MPI command for a broadcast is

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

The process in the communicator with rank `root` broadcasts the contents of `message` to all the other processes in the communicator.

The MPI command for a reduce operation is

```
int MPI_Reduce(
        void*          operand   /* in  */,
        void*          result    /* out */,
        int            count     /* in  */,
        MPI_Datatype   datatype  /* in  */,
        MPI_Op         operator  /* in  */,
        int            root      /* in  */,
        MPI_Comm       comm      /* in  */)
```

The values stored in each process's `operand` are combined using `operator`. The result is stored in `result` on the process with rank `root`.

The MPI command for a gather operation is

```
int MPI_Gather(
        void*          send_data    /* in  */,
        int            send_count   /* in  */,
        MPI_Datatype   send_type    /* in  */,
        void*          recv_data    /* out */,
        int            recv_count   /* in  */,
        MPI_Datatype   recv_type    /* in  */,
        int            root         /* in  */,
        MPI_Comm       comm         /* in  */)
```

It gathers the data stored in each process's `send_data` into the memory referenced by `recv_data` on the process with rank `root`.

The MPI command for a scatter operation is

```
int MPI_Scatter(
        void*          send_data    /* in  */,
        int            send_count   /* in  */,
        MPI_Datatype   send_type    /* in  */,
        void*          recv_data    /* out */,
        int            recv_count   /* in  */,
        MPI_Datatype   recv_type    /* in  */,
        int            root         /* in  */,
        MPI_Comm       comm         /* in  */)
```

It distributes the memory referenced by `send_data` across the processes in `comm`.

None of the collective communication operations use tags. This is because they are supposed to behave, in terms of the data transmitted, as if they were *synchronous* operations. A communication operation is synchronous if it cannot complete on any one process until all the other processes involved have begun execution of the operation. The practical effect of the absence of tags is that if the processes in a communicator are executing a sequence of collective communication operations, they must all execute the same operations in the same order.

The use of the same argument for two distinct parameters is called *aliasing*. Aliasing of output or input/output parameters is illegal in MPI. In particular, it is illegal to use the same argument for both `operand` and `result` in `MPI_Reduce`, and it is illegal to use the same argument for `send_data` and `recv_data` in both `MPI_Gather` and `MPI_Scatter`.

Most parallel systems use *buffered* communication. This means that the data in a message that is being sent is put into temporary storage until the receiving process calls a receive function, at which point the data is transferred into regular storage. If a parallel system doesn't provide buffering, then all communication must be synchronous.

In MPI, a program that will run correctly with buffering but will fail without buffering is said to be *unsafe*. The most common reason for a program to fail when run without buffering is *deadlock*. This means that the processes are all waiting or hung, and the program will stop executing new commands.

We saw that it may be useful for the result of a reduce or a gather to be available to all the processes in the communicator. We also saw that implementations of these extended operations may be more efficient if they are not based on `MPI_Reduce` or `MPI_Gather`. For example, we can use a *butterfly structured* communication pattern. MPI allows for this possibility by defining extended reduce and gather operations.

```
int MPI_Allreduce(
        void*          operand   /* in  */,
        void*          result    /* out */,
        int            count     /* in  */,
        MPI_Datatype   datatype  /* in  */,
        MPI_Op         operator  /* in  */,
        MPI_Comm       comm      /* in  */)
```

This function is used in exactly the same way as `MPI_Reduce`. The only difference is that after the function returns, all the processes in the communicator will have the result stored in the memory referenced by `result`.

The extended gather function is

```
int MPI_Allgather(
        void*          send_data   /* in  */,
```

```
int           send_count  /* in  */,
MPI_Datatype  send_type   /* in  */,
void*         recv_data   /* out */,
int           recv_count  /* in  */,
MPI_Datatype  recv_type   /* in  */,
MPI_Comm      comm        /* in  */)
```

This has the effect of gathering the contents of each process's `send_data` into each process's `recv_data`.

5.10 References

Definitions, discussions, and examples of MPI's collective communications are contained in the MPI Standard [28, 29]. Further discussion and examples are contained in [34] and [21]. For background material and detailed discussions of implementations of collective operations, see [26]. For an elementary discussion of parallel implementations of linear algebra operations, see [19].

5.11 Exercises

1. Study the two tree-structured broadcasts on a linear array of eight processors, a 2×4 mesh, and a three-dimensional hypercube. Is there any reason to prefer one of the broadcasts to the other on each topology? Can you devise a better pattern for the broadcast on any of the architectures?
2. Suppose that `MPI_COMM_WORLD` consists of the three processes 0, 1, and 2, and suppose the following code is executed:

```
int x, y, z;

switch (my_rank) {
    case 0: x = 0; y = 1; z = 2;
        MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Send(&y, 1, MPI_INT, 2, 43, MPI_COMM_WORLD);
        MPI_Bcast(&z, 1, MPI_INT, 1, MPI_COMM_WORLD);
        break;
    case 1: x = 3; y = 4; z = 5;
        MPI_Bcast(&x, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Bcast(&y, 1, MPI_INT, 1, MPI_COMM_WORLD);
        break;
    case 2: x = 6; y = 7; z = 8;
        MPI_Bcast(&z, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Recv(&x, 1, MPI_INT, 0, 43, MPI_COMM_WORLD,
            &status);
        MPI_Bcast(&y, 1, MPI_INT, 1, MPI_COMM_WORLD);
        break;
}
```

What are the values of `x`, `y`, and `z` on each process after the code has been executed?

3. Modify the trapezoidal rule program so that it uses `Get_data1`.
4. Modify the trapezoidal rule program so that it uses `Get_data2`.
5. Let's take a look at a simple special case of the scatter-reduce matrix-vector multiplication: $m = n = p = 2$. Illustrate the steps involved in multiplying a 2×2 matrix by a vector if we first scatter each row of the matrix, carry out the multiplication of the matrix entries by the elements of the vector, and then sum the results of the multiplications. Your illustration should show how the contents of each process's memory changes after each step of the process. Use arrows to indicate communication of data. Does this example suggest a distribution of the matrix that might produce a more efficient matrix-vector multiplication?
6. Illustrate the implementation of gather and scatter using a tree and the implementation of allgather using a butterfly. Use eight processes and arrays of order 16. Your diagram should show the memory referenced by `send_data` on each process at the beginning of the communication. It should also show the contents of `recv_data` at each stage of the process.

5.12 Programming Assignments

In each of the assignments, assume that the appropriate parameters are divisible by p. For example, if you are writing a matrix-vector multiply in which you use a block row distribution of the matrix and a block distribution of the vectors, then the number of rows and the number of columns of the matrix should both be divisible by p.

Be sure to write assignment 1 before working on the other assignments.

1. In order to avoid some of the problems that may occur with I/O on your system, you should only input the scalar parameters to your programs. For example, if you are multiplying a matrix by a vector, you should only input the order of the matrix. The program should generate the matrix and vector (e.g., a matrix of all ones and a vector of all ones). Of course, you won't be able to tell whether your program works unless you print the results of your computations. So you should write two output functions: a vector output function and a matrix output function for matrices distributed by block rows.
2. Write a subprogram that performs a "global sum" of a distributed collection of floats. The parameters to the subprogram should consist of a scalar operand provided by each process and a root process. The operand will be an `in/out` parameter on the root. That is, on the root, the result of the global sum will be returned in the operand. We're not aiming for efficiency here. So a simple loop of receives on the root process will suffice. Does

your subprogram have to create a temporary buffer on the root process? Modify your subprogram so that it performs a "global vector sum." Each process passes a vector of floats, the size of the vector, and a root process to the subprogram. The root process returns the result in its input vector. Does your subprogram have to create a temporary buffer on the root process? If so, can you come up with an approach to solving this problem so that the root process doesn't need to create a temporary buffer?

3. Recollect that a matrix-matrix product is formed by taking the dot product of each row of the left factor with each column of the right factor. More explicitly, suppose $A = (a_{ij})$ is an $m \times n$ matrix and $B = (b_{ij})$ is an $n \times r$ matrix. Then the product matrix, $AB = C = (c_{ij})$ is an $m \times r$ matrix, and c_{ij} is the dot product of the ith row of A with the jth column of B, or

$$c_{ij} = a_{i0}b_{0j} + a_{i1}b_{1j} + \cdots + a_{i,n-1}b_{n-1,j}.$$

If A and B have been distributed by block rows, then we can use our parallel matrix-vector product to compute a parallel matrix-matrix product as follows:

```
for each column x of B {
    Compute the parallel matrix-vector product Ax
}
```

Write a subprogram that will compute a parallel matrix-matrix product using this algorithm.