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 that 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 Figure 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 you think about it, the original scheme required \(\text{comm.sz} - 1\) = 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 a property by which 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, that is, three receives and three additions. We've thus reduced the overall time by more than 50%. Furthermore, if we use more processes, we can do even better. For example, if \(\text{comm.sz} = 1024\), 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!

![Figure 3.6](image)

**Figure 3.6**
A tree-structured global sum
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 Figure 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 its performance? If we do, is it possible that one method works best for "small" 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. MPI generalized the global-sum function so that any one of these possibilities can be implemented with a single function:

```c
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-1 Standard [39].

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

```c
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. The following code could thus be used to

<table>
<thead>
<tr>
<th>Operation Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI.MAX</td>
<td>Maximum</td>
</tr>
<tr>
<td>MPI.MIN</td>
<td>Minimum</td>
</tr>
<tr>
<td>MPI.SUM</td>
<td>Sum</td>
</tr>
<tr>
<td>MPI.PROD</td>
<td>Product</td>
</tr>
<tr>
<td>MPI.LAND</td>
<td>Logical and</td>
</tr>
<tr>
<td>MPI.BAND</td>
<td>Bitwise and</td>
</tr>
<tr>
<td>MPI.LOR</td>
<td>Logical or</td>
</tr>
<tr>
<td>MPI.BOR</td>
<td>Bitwise or</td>
</tr>
<tr>
<td>MPI.LXOR</td>
<td>Logical exclusive or</td>
</tr>
<tr>
<td>MPI.BXOR</td>
<td>Bitwise exclusive or</td>
</tr>
<tr>
<td>MPI.MAXLOC</td>
<td>Maximum and location of maximum</td>
</tr>
<tr>
<td>MPI.MINLOC</td>
<td>Minimum and location of minimum</td>
</tr>
</tbody>
</table>

Table 3.2 Predefined Reduction Operators in MPI
add a collection of $N$-dimensional vectors, one per process:

```c
double local.x[N], sum[N];
... MPI.Reduce(local.x, sum, N, MPI.DOUBLE, MPI.SUM, 0,
MPI.COMM.WORLD);
```

### 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 three, and the value of `d` will be six. 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

```c
MPI.Reduce(&x, &x, 1, MPI.DOUBLE, MPI.SUM, 0, comm);
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$a = 1; c = 2$</td>
<td>$a = 1; c = 2$</td>
<td>$a = 1; c = 2$</td>
</tr>
<tr>
<td>1</td>
<td><code>MPI.Reduce(&amp;a, &amp;b, ...)</code></td>
<td><code>MPI.Reduce(&amp;c, &amp;d, ...)</code></td>
<td><code>MPI.Reduce(&amp;a, &amp;b, ...)</code></td>
</tr>
<tr>
<td>2</td>
<td><code>MPI.Reduce(&amp;c, &amp;d, ...)</code></td>
<td><code>MPI.Reduce(&amp;a, &amp;b, ...)</code></td>
<td><code>MPI.Reduce(&amp;c, &amp;d, ...)</code></td>
</tr>
</tbody>
</table>
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 Section 6.1.9 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 in order 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 Figure 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 Figure 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:

```c
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 Figure 3.6, we obtain the tree-structured communication shown in Figure 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
3.4 Collective Communication

FIGURE 3.8
A global sum followed by distribution of the result

FIGURE 3.9
A butterfly-structured global sum
A tree-structured broadcast

a broadcast function:

```c
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`. Program 3.6 shows how

```c
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("%f %f %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`
to modify the Get_input function shown in Program 3.5 so that it uses MPI_Bcast instead of MPI_Send and MPI_Recv.

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:

\[
\begin{align*}
    x + y &= (x_0, x_1, \ldots, x_{n-1}) + (y_0, y_1, \ldots, y_{n-1}) \\
    &= (x_0 + y_0, x_1 + y_1, \ldots, x_{n-1} + y_{n-1}) \\
    &= (z_0, z_1, \ldots, z_{n-1}) \\
    &= z
\end{align*}
\]

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

```c
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

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 \(\text{comm.sz}\) cores or processes, let’s assume that \(n\) evenly divides \(\text{comm.sz}\) and define \(\text{local.n} = n/\text{comm.sz}\). Then we can simply assign blocks of \(\text{local.n}\) consecutive components to each process. The four columns on the left of Table 3.4 show an example when \(n = 12\) and \(\text{comm.sz} = 3\). 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 \(\text{comm.sz} = 3\). Process 0 gets component 0, process 1 gets component 1, process 2 gets component 2, process 0 gets component 3, and so on.
Table 3.4 Different Partitions of a 12-Component Vector among Three Processes

<table>
<thead>
<tr>
<th>Components</th>
<th>Block Cyclic Block-Cyclic Blocksize = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>Block</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
</tbody>
</table>

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 \(\text{comm.sz} = 3\), \(n = 12\), 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 \(\text{local.n}\) components of the vector, and, in order to save on storage, we can just store these on each process as an array of \(\text{local.n}\) 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' portion of the vector, this function is virtually identical to the original serial function.

```c
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];
}
```

Program 3.8: A parallel implementation of vector addition

### 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 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, and so on. 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 that is on process 0 but only sends the needed components to each of the other processes. For the communication MPI provides just such a function:

```c
int MPI_Scatter(
    void* sendbuf, /* in */,
    int sendcount /* in */,
    MPI_Datatype sendtype /* in */,
    void* recvbuf /* out */,
    int recvcount /* in */,
    MPI_Datatype recvtype /* in */,
    int srcproc /* in */,
    MPI_Comm comm /* in */);
```

If the communicator `comm` contains `comm_sz` processes, then `MPI_Scatter` divides the data referenced by `sendbuf` 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 `sendbuf`. 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 `recvbuf` argument and the `recvcount` argument should be `local_n`. Both `sendtype` and `recvtype` should be `MPI.DOUBLE` and `srcproc` should be 0. Perhaps surprisingly, `recvcount` should also be `local_n`—`sendcount` is the amount of data going to each process; it’s not the amount of data in the memory referred to by `sendbuf`. 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.

One point to note here is that `MPI_Scatter` sends the first block of `sendcount` objects to process 0, the next block of `sendcount` 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 a partial solution to dealing with a cyclic or block-cyclic distribution in Exercise 18. For a complete solution, see [23]. We’ll look at dealing with the case in which n is not evenly divisible by `comm_sz` in Exercise 3.13.
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("%f", &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

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
", 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

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 \( x \) is a vector with \( n \) components, then \( y = Ax \) is a vector with \( m \) components and we can find the \( i \)th component of \( y \) by forming the dot product of the \( i \)th row of \( A \) with \( x \):

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

See Figure 3.11.

Thus, we might write pseudo-code 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];
}
FIGURE 3.11
Matrix-vector multiplication

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 in the two-dimensional array (the 9), 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.

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 \(x\) and the addition of this product into a component of \(y\). That is, each execution of the statement

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

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 \(\text{comm.sz}\) 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 \(i\)th row of
void Mat_vec 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_vec mult */
}

Program 3.11: Serial matrix-vector multiplication

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’ send_buf_p and stores this in each process’ recv_buf_p. As usual, recv_count is the amount of data being
void Mat_vec_mult(
  double localA[] /* in */,
  double localX[] /* in */,
  double localY[] /* out */,
  int localM /* in */,
  int n /* in */,
  int localN /* in */,
  MPI_Comm comm /* in */)
{
  double* x;
  int localI, j;
  int localOk = 1;

  x = malloc(n*sizeof(double));
  MPI_Allgather(localX, localN, MPI_DOUBLE,
                x, localN, MPI_DOUBLE, comm);

  for (localI = 0; localI < localM; localI++) {
    localY[localI] = 0.0;
    for (j = 0; j < n; j++)
      localY[localI] += localA[localI*n+j]*x[j];
  }
  free(x);
  /* Mat_vec_mult */
}

Program 3.12: An MPI matrix-vector multiplication function

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 doubles 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:

```c
double x[1000];
if (my_rank == 0)
  for (i = 0; i < 1000; i++)
```