4 Intermediate MPI

In the preceding chapter we considered a number of straightforward parallel algorithms. We wrote parallel programs for these algorithms using straightforward MPI subroutine calls, and we verified with timing and program visualization tools that our programs behaved as we expected.

In this chapter we introduce several important issues that require more subtlety in our analysis and more precision in our tools. The mathematical problem we will address here (the Poisson problem) will be only a little more complicated than the problems of the preceding chapter, but the parallel algorithm, particularly the communication pattern, will admit more options.

We introduce several new MPI routines. For example, our mathematical problem takes place on a finite-difference computational grid, and so we introduce the MPI notion of *virtual topology*, which makes the allocation of processes to particular parts of a grid convenient to manage. We also describe many of the variations on the basic *send* and *receive* operations supported by MPI; indeed, the communication patterns needed here by our parallel algorithm motivated some of the more esoteric MPI features.

Our first goal in this chapter is thus to show how MPI enables efficient programs to be written concisely. A secondary goal is to explain some of the issues that arise in analysis of communication patterns in grid-based problems.

We approach both of these goals by examining a number of programs for the 2-D and 3-D Poisson problem, a model partial differential equation. Because Fortran provides a much more convenient syntax for manipulating multidimensional arrays than does C, the bulk of the examples in this chapter are written in Fortran 77.

We also use the Poisson problem as a means to introduce the different mechanisms by which an MPI program can send data from one process to another, particularly with respect to both how data is buffered in the message passing system and how nonblocking communications can be used. By examining the different approaches in the context of a single application, we make clear the distinctions between these approaches. We begin by explaining the mathematical problem and an approach to solving it computationally. Then we describe MPI's virtual topology feature, which allows us to manage a grid of processes. As we progress, we introduce several new MPI functions while considering various ways of organizing the communications. Finally, to help in understanding the reasons for the different implementation choices, we will make another brief foray into scalability analysis.

This chapter may also be viewed as a discussion of the sparse matrix-vector product, because that is really the fundamental operation at work in these algorithms. While we will not discuss it in this book, the message-passing operations discussed in this chapter are the same as are used in implementing a parallel sparse matrix-vector product. The Jacobi method was chosen for its simplicity in the computational part of the program, allowing us to present a complete application.

4.1 The Poisson Problem

The Poisson problem is a simple partial differential equation (PDE) that is at the core of many applications. More elaborate problems and algorithms often have the same communication structure that we will use here to solve this problem. Thus, by studying how MPI can be used here, we are providing fundamentals on how communication patterns appear in more complex PDE problems. At the same time, we can demonstrate a wide variety of message-passing techniques and how MPI may be used to express them.

We emphasize that while the Poisson problem is a useful example for describing the features of MPI that can be used in solving partial differential equations and other problems that involve decomposition across many processes, the numerical techniques in this section are not the last word in solving PDEs and give poor performance relative to more recent and sophisticated methods. For information on more sophisticated, freely available parallel solvers for PDE's that use MPI, see [7]. For more details about the mathematical terminology used in this chapter, consult [10], among other sources.

The Poisson problem is expressed by the equations

$$\nabla^2 u = f(x, y) \text{ in the interior} \tag{4.1.1}$$

$$u(x, y) = g(x, y) \text{ on the boundary} \tag{4.1.2}$$

To simplify the discussion, we use the unit square as the domain.

To find an approximate solution to this problem, we define a square *mesh* (also called a *grid*) consisting of the points (x_i, y_j), given by

$$x_i = \frac{i}{n+1}, i = 0, \ldots, n+1,$$

$$y_j = \frac{j}{n+1}, j = 0, \ldots, n+1,$$

where there are $n+2$ points along each edge of the mesh (see Figure 4.1). We will find an approximation to $u(x, y)$ only at the points (x_i, y_j). We use the shorthand $u_{i,j}$ to refer to the approximation to u at (x_i, y_j). The value $1/(n+1)$ is used frequently; we will denote it by h (following common practice). We can approximate (4.1.1) at each of these points with the formula [10]

$$\frac{u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + u_{i+1,j} - 4u_{i,j}}{h^2} = f_{i,j}. \tag{4.1.3}$$

We wish to solve (4.1.3) for $u_{i,j}$ everywhere on the mesh. Since the formula involves u at five points, we must find some way to solve for u everywhere. One approach is to rewrite (4.1.3) as

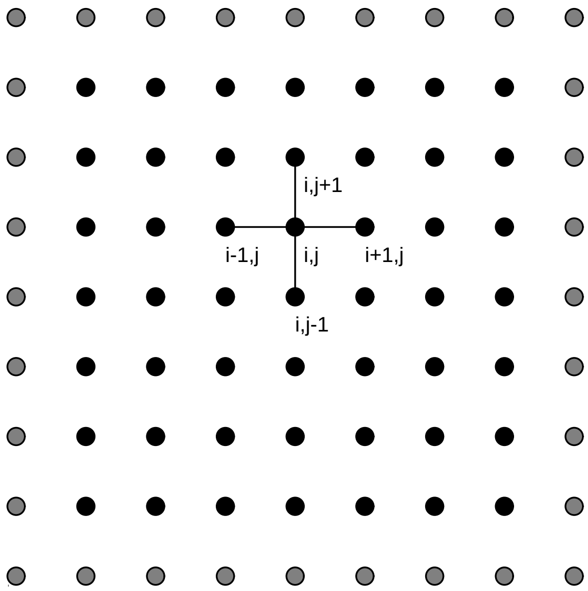


Figure 4.1
Five-point stencil approximation for 2-D Poisson problem, with $n = 7$. The boundaries of the domain are shown in gray.

$$u_{i,j} = \frac{1}{4}\left(u_{i-1,j} + u_{i,j+1} + u_{i,j-1} + u_{i+1,j} - h^2 f_{i,j}\right),$$

iterate by choosing values for all mesh points $u_{i,j}$, and then replace them by using[1]

$$u^{k+1}_{i,j} = \frac{1}{4}\left(u^k_{i-1,j} + u^k_{i,j+1} + u^k_{i,j-1} + u^k_{i+1,j} - h^2 f_{i,j}\right).$$

This process, known as *Jacobi iteration*, is repeated until the solution is reached. Fortran code for this is shown in Figure 4.2.

```
      integer i, j, n
      double precision u(0:n+1,0:n+1), unew(0:n+1,0:n+1)
      do 10 j=1, n
         do 10 i=1, n
            unew(i,j) = &
                0.25*(u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j)) - &
                h * h * f(i,j)
 10   continue
```

Figure 4.2
Jacobi iteration

To parallelize this algorithm, we need to parallelize the loops in this code. To do this, we must distribute the data, in this case the arrays `u`, `unew`, and `f`, across the processes. Several approaches are possible.

One of the simplest decompositions is shown in Figure 4.3. In this decomposition, the physical domain is sliced into slabs, with the computations on each slab being handled by a different process.

It is easy to describe this decomposition in Fortran. On each process, the arrays are dimensioned as

```
double precision u(0:n+1,s:e)
```

where `s:e` indicates the values of j that this process is responsible for. This way of declaring `u` changes the code for the algorithm to that shown in Figure 4.4.

Unfortunately, a problem arises. The loop will require elements such as `u(i,s-1)`, that is, data from a different process. The rest of this chapter will discuss how to identify which process the data is from and how to get that data.

But first, let us fix our routine. Since the data is needed, we must expand our arrays to hold the data. In this case, a dimension of

[1] The ways in which arrays and matrices correspond to one another and are laid out in memory by Fortran and C compilers are often a source of confusion. We discuss this topic in excruciating detail in Appendix **??**.

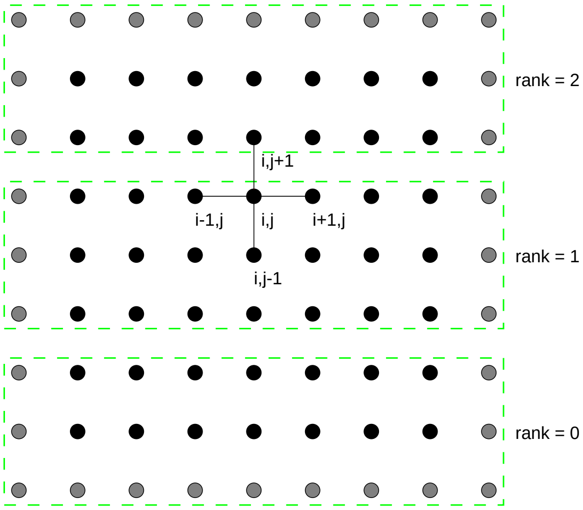


Figure 4.3
1-D decomposition of the domain

```
      integer i, j, n
      double precision u(0:n+1,s:e), unew(0:n+1,s:e)
      do 10 j=s, e
         do 10 i=1, n
            unew(i,j) = &
                  0.25*(u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j)) - &
                     h * h * f(i,j)
 10   continue
```

Figure 4.4
Jacobi iteration for a slice of the domain

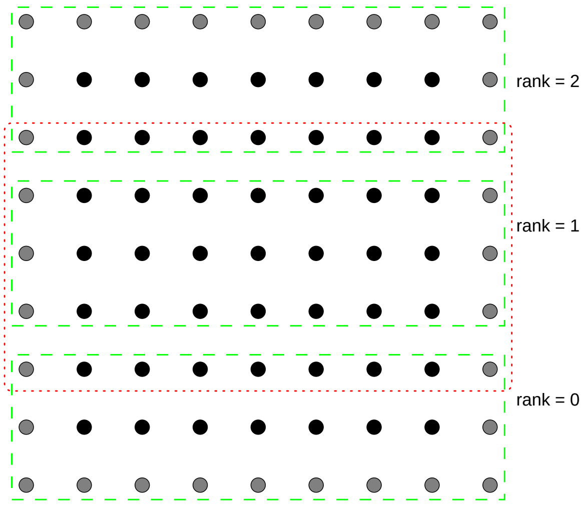


Figure 4.5
The computational domain, with ghost points, for one of the processes

```
double precision u(0:n+1,s-1:e+1)
```

is sufficient (see Figure 4.5). The elements of the array that are used to hold data from other processes are called *ghost points*. We will show how to get the data for these ghost points in Section 4.3.

4.2 Topologies

Our next task is deciding how to assign processes to each part of the decomposed domain. An extensive literature on this subject (e.g., [3, 6, 11]) exists. Handling this assignment of processes to regions is one of the services that MPI provides to the programmer, exactly because the best (or even a good) choice of decomposition depends on the details of the underlying hardware.

The description of how the processes in a parallel computer are connected to one another is often called the *topology* of the computer (or more precisely, of the interconnection network). In most parallel programs, each process communicates with only a few other processes; the pattern of communication is called an *application topology*. The relationships between the topology of the parallel computer's hardware and the application can be made in many ways; some are better than others.

For example, it may seem that simply assigning processes in increasing rank from the bottom is the best approach. On some parallel computers, however, this ordering can lead to performance degradation (see [8, 9] for more details). It is hard for anyone but the vendor to know the best way for application topologies to be fitted onto the physical topology of the parallel machine. MPI allows the vendor to help optimize this aspect of the program through his implementation of the MPI topology functions.

MPI allows the user to define a particular application, or virtual topology. An important virtual topology is the *Cartesian* topology. This is simply a decomposition in the natural coordinate (e.g., x, y) directions. A two-dimensional Cartesian decomposition is shown in Figure 4.6. Each element of the decomposition (rectangles in the figure) is labeled by a coordinate tuple indicating the position of the element in each of the coordinate directions. For example, the second process from the left and the third from the bottom is labeled `(1,2)`. (The indices start from zero, following the practice of C, rather than starting at one, which may be more natural for Fortran users.) MPI provides a collection of routines for defining, examining, and manipulating Cartesian topologies.

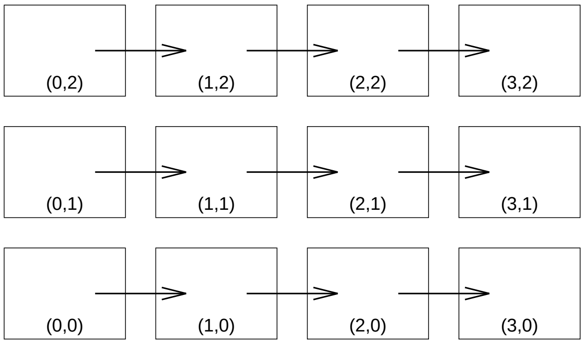


Figure 4.6
A two-dimensional Cartesian decomposition of a domain, also showing a shift by one in the first dimension. Tuples give the coordinates as would be returned by `MPI_Get_coords`.

The routine `MPI_Cart_create` creates a Cartesian decomposition of the processes, with the number of dimensions given by the `ndim` argument. The user can specify the number of processes in any direction by giving a positive value to the corresponding element of `dims`. For example, to form the decomposition shown in Figure 4.6, one can use the following code:

```
dims(1)    = 4
dims(2)    = 3
periods(1) = .false.
periods(2) = .false.
reorder    = .true.
```

```
ndim       = 2
call MPI_CART_CREATE( MPI_COMM_WORLD, ndim, dims, periods, &
                      reorder, comm2d, ierr )
```

This creates a new communicator in the sixth argument from the communicator in the first argument. The new communicator has the Cartesian topology defined by the second through fifth arguments. The `periods` argument indicates whether the processes at the "ends" are connected together (for example, is the right neighbor of the process at the right end the *leftmost* process in that row?). This is useful for "periodic" domains. For example, in simulating the weather on the Earth within the temperate latitudes using a three-dimensional grid with the dimensions referring to east-west, north-south, and up-down, the first of these is periodic and the other two are not.

Note that we have not specified which process is assigned to each of the elements of the decomposition. By setting the argument `reorder` to `.true.`, we have allowed MPI to find a good way to assign the process to the elements of the decomposition.

In one dimension, we can simply use the rank in the new communicator, plus or minus one, to find our neighbors (and not use `MPI_Cart_create`). Even in this case, this may not be the best choice, since neighbors defined in this way may not be neighbors in the actual hardware. In more than one dimension, however, it is more difficult to determine the neighboring processes. The `reorder` argument, when true, lets MPI know that it may reorder the processes for better performance.

Fortunately, MPI provides a simple way to find the neighbors of a Cartesian mesh. The most direct way is to use the routine `MPI_Cart_get`. This routine returns both values of the `dims` and `periods` argument used in `MPI_Cart_create` as well as an array `coords` that contains the Cartesian coordinates of the calling process. For example, the code

```
call MPI_CART_GET( comm1d, 2, dims, periods, coords, ierr )
print *, '(', coords(1), ',', coords(2), ')'
```

will print the coordinates of the calling process in the communicator `comm1d`. Another way is to use `MPI_Cart_coords`; this routine, given a rank in a communicator, returns the coordinates of the process with that rank. For example, to get the coordinates of the calling process, one can use

```
call MPI_COMM_RANK( comm1d, myrank, ierr )
call MPI_CART_COORDS( comm1d, myrank, 2, coords, ierr )
```

However, there is another way that is more closely related to what we are trying to accomplish. Each process needs to send and receive data from its neighbors. In the 1-D decomposition, these are the neighbors above and below. There are many ways to do this, but a simple one is illustrated in Figure 4.7. This represents a copy of the top row from one process to the bottom ghost-point row of the process above it, followed by a copy of the bottom row to the top ghost-point row of the process below. If we look at the first of these operations, we see that each process is both sending and receiving data. In fact, one way to look at this is that data is being shifted up from one process to another. This is a common operation, and MPI provides the routine `MPI_Cart_shift` that may be used to find the neighbors.

Figure 4.6 shows a (nonperiodic) shift by one in the first dimension in a two-dimensional Cartesian grid. `MPI_Cart_shift` may be used to find the destination and source of a shift for each process. For example, the process at Cartesian coordinates $(1, 1)$ has destination at $(2, 1)$ and source at $(0, 1)$. This gives the neighbors to the left (the rank of the process at coordinates $(0, 1)$) and to the right (the rank of the process at coordinates $(2, 1)$).

What happens for a right shift at the right edge? For example, what is the right neighbor of $(3, 0)$ in Figure 4.6? If the grid were periodic, the right neighbor would be $(0, 0)$. In our application, however, the grid is not periodic, and thus there is no neighbor. This is indicated by the value `MPI_PROC_NULL`. This value is a valid source and a valid destination for all the MPI send and receive routines. The behavior of an `MPI_Send` or `MPI_Recv` with `MPI_PROC_NULL` as a source or destination is identical to code of this form:

```
if (source .ne. MPI_PROC_NULL) then
    call MPI_SEND( ..., source, ... )
endif
```

We will exploit `MPI_PROC_NULL` when we write the code to move data between the processes.

The last routine that we need in defining the decomposition helps us determine the array limits (`s` and `e` in the sample code), given the Cartesian coordinate of the process and the size of the array (`n` in our sample). Because it is a common need, we have provided `MPE_Decomp1d`. To determine the values of `s` and `e`, we use

```
    call MPE_DECOMP1D( n, size, rank, s, e )
```

where `size` is the number of processes in the Cartesian coordinate, `rank` is the Cartesian coordinate of the calling process, and `n` is the size of the array (assumed to run from `1` to `n`). `MPE_Decomp1d` computes values for `s` and `e`. If `n` is evenly divisible by `size`, this routine isn't really needed; in that case we have

$$
\begin{aligned}
s &= 1 + rank * \text{floor}(n/size) \\
e &= s + \text{floor}(n/size) - 1
\end{aligned}
$$

By using this routine, we are ensured that if `size` does not evenly divide `n`, we will still get correct decomposition of the data, with good load balancing. To see

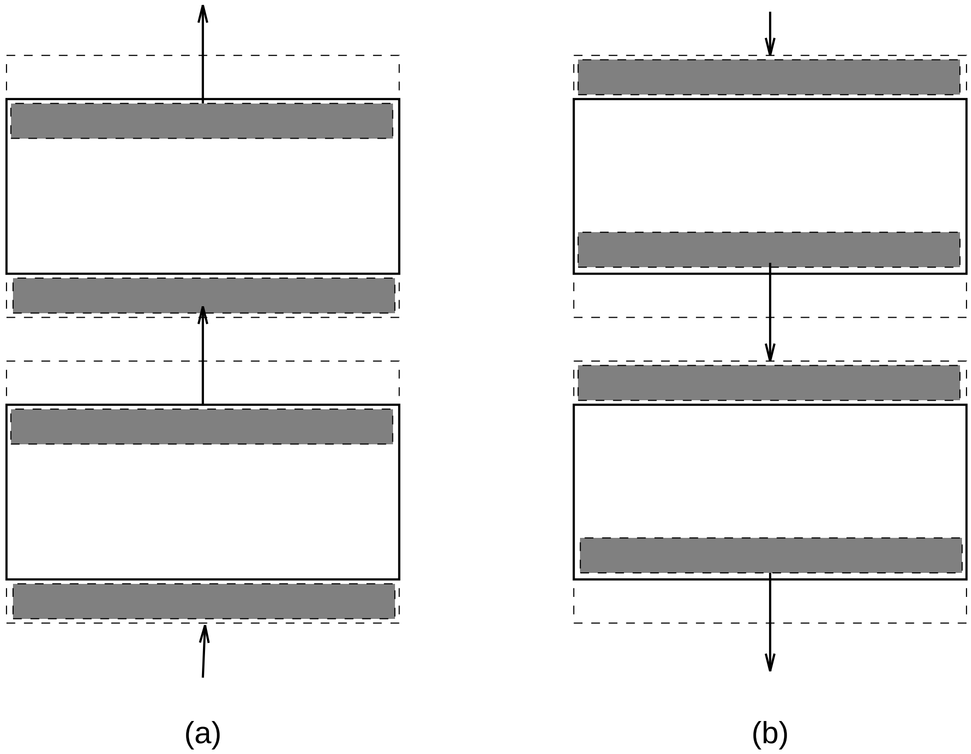


Figure 4.7
Two-step process to transfer data. Ghost point areas are shown in dashed boxes; data to be moved is shaded.

```
MPI_CART_CREATE(commold, ndims, dims, periods, reorder, newcomm,ierror)
        integer commold, ndims, dims(*), newcomm, ierror
        logical periodic(*),reorder

MPI_CART_SHIFT(comm, direction, shift, src, dest, ierror )
        integer comm, direction, shift, src, dest, ierror

MPI_CART_GET(comm, maxdims, dims, periods, coords, ierror)
        integer comm, maxdims, dims(*), coords(*), ierror
        logical periods(*)

MPI_CART_COORDS(comm, rank, maxdims, coords, ierror)
        integer comm, rank, maxdims, coords(*), ierror
```

Table 4.1
Fortran bindings for topology routines

why, consider using these formulas with `size=64` and `n=127`. Every process gets floor$(n/size) = 1$ elements except for the last, which gets $n-63*\text{floor}(n/size) = 64$. `MPE_Decomp1d` in this case gives processes 0 through 62 two elements and gives the last process a single element.

Now that we know how the data is decomposed among the processes, and how the processes are ranked in the decomposition, we can write the routine to get the data that we need. For each process, we must get the ghost-point data for the `s-1` row from the process below and the data for the `e+1` row from the process above. Many methods exist to do even this simple operation, and we will investigate several of them through the course of this chapter.

4.3 A Code for the Poisson Problem

In this section we will assemble the pieces of the code that we have defined, as well as the first version of the MPI routines needed to exchange the ghost points between the processes.

The only piece of this code that we have not yet described is the routine to exchange data between the processes. The rest of this chapter will be concerned with different ways to perform this communication operation, and it will pay particular attention to some subtle issues that are often ignored in discussions of message passing. With the warning that we are about to embark on a long journey, we start with perhaps the simplest approach, shown in Figure 4.8.

In this routine, each process sends data to the process on top and then receives data from the process below it. The order is then reversed, and data is sent to the

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

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

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

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

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

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

MPI_WAITANY(count, array_of_requests, index, status, ierror)
integer count, array_of_requests(*), index,
status(MPI_STATUS_SIZE),ierror

Table 4.2
Fortran bindings for various data exchange routines

int **MPI_Cart_create**(MPI_Comm comm_old, int ndims, int *dims,int *periods,
int reorder, MPI_Comm *new_comm)

int **MPI_Cart_shift**(MPI_Comm comm, int direction, int displ,int *src, int *dest)

int **MPI_Cart_get**(MPI_Comm comm, int maxdims, int *dims, int *periods,
int *coords)

int **MPI_Cart_coords**(MPI_Comm comm, int rank, int maxdims, int *coords)

Table 4.3
C bindings for topology routines

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

int MPI_Isend(void* buf, int count, MPI_Datatype datatype, int dest,int tag,
                 MPI_Comm comm, MPI_Request *request)

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

int MPI_Wait(MPI_Request *request, MPI_Status *status)

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

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

int MPI_Waitany(int count, MPI_Request *array_of_requests,int *index,
                 MPI_Status *status)
```

Table 4.4
C bindings for various data exchange routines

```
        subroutine exchng1( a, nx, s, e, comm1d, nbrbottom, nbrtop )
        use mpi
        integer nx, s, e
        double precision a(0:nx+1,s-1:e+1)
        integer comm1d, nbrbottom, nbrtop
        integer status(MPI_STATUS_SIZE), ierr
!
        call MPI_SEND( a(1,e), nx, MPI_DOUBLE_PRECISION, &
                       nbrtop, 0, comm1d, ierr )
        call MPI_RECV( a(1,s-1), nx, MPI_DOUBLE_PRECISION, &
                       nbrbottom, 0, comm1d, status, ierr )
        call MPI_SEND( a(1,s), nx, MPI_DOUBLE_PRECISION, &
                      nbrbottom, 1, comm1d, ierr )
        call MPI_RECV( a(1,e+1), nx, MPI_DOUBLE_PRECISION, &
                      nbrtop, 1, comm1d, status, ierr )
        return
        end
```

Figure 4.8
Code to exchange data for ghost points using blocking sends and receives

process below and received from the process above. We will see below that, while this strategy is simple, it is not necessarily the best way to implement the exchange of ghost points.

We now have all of the pieces needed to put together our first version of the Poisson solver program. This program uses `MPI_Cart_create` to create the decomposition of processes and the routine `MPE_Decomp1d` to determine the decomposition of the arrays. The routine `onedinit` simply initializes the elements of the arrays `a`, `b`, and `f`. The solution is computed alternately in the array `a` and then `b`; this is why there are two calls to `exchng1` and `sweep1d` in the loop. The iteration is terminated when the difference between two successive approximations to the solution is less than 1×10^{-5}. The difference between the local parts of `a` and `b` is computed with the routine `diff`; the routine `MPI_Allreduce` is used to ensure that all processes compute the same value for the difference in all of the elements. The program prints both the progress of the iterations and the final iteration count. A do-loop with a maximum iteration count ensures that the program terminates even if the iteration is not converging. The computational part of the program is shown in Figure 4.9.

Let's run this program, using the MPI profiling interface and the graphical tools to understand the behavior. Using the "automatic" profiling method described in Chapter **??**, we use the MPE profiling files `mpe_proff.o` and `mpe_prof.o` to generate a logfile of the communication. The '`Makefile`' in

```
! Get a new communicator for a decomposition of the domain
! and my position in it
      call MPI_CART_CREATE( MPI_COMM_WORLD, 1, numprocs, .false., &
                            .true., comm1d, ierr )
      call MPI_COMM_RANK( comm1d, myid, ierr )
      call MPI_CART_SHIFT( comm1d, 0,  1, nbrbottom, nbrtop, ierr )
! Compute the actual decomposition
      call MPE_DECOMP1D( ny, numprocs, myid, s, e )
! Initialize the right-hand-side (f) and the initial solution guess (a)
      call ONEDINIT( a, b, f, nx, s, e )
!
! Actually do the computation.  Note the use of a collective operation to
! check for convergence, and a do-loop to bound the number of iterations.
!
      do 10 it=1, maxit
!                    get ghost points
        call EXCHNG1( a, nx, s, e, comm1d, nbrbottom, nbrtop )
!                    perform one Jacobi "sweep"
        call SWEEP1D( a, f, nx, s, e, b )
!                    repeat to get a solution back into array a
        call EXCHNG1( b, nx, s, e, comm1d, nbrbottom, nbrtop )
        call SWEEP1D( b, f, nx, s, e, a )
!                    check for convergence
        diffw = DIFF( a, b, nx, s, e )
        call MPI_ALLREDUCE( diffw, diffnorm, 1, MPI_DOUBLE_PRECISION, &
                            MPI_SUM, comm1d, ierr )
        if (diffnorm .lt. 1.0e-5) goto 20
        if (myid .eq. 0) print *, 2*it, ' Difference is ', diffnorm
10     continue
      if (myid .eq. 0) print *, 'Failed to converge'
20    continue
      if (myid .eq. 0) then
          print *, 'Converged after ', 2*it, ' Iterations'
      endif
```

Figure 4.9
Implementation of the Jacobi iteration

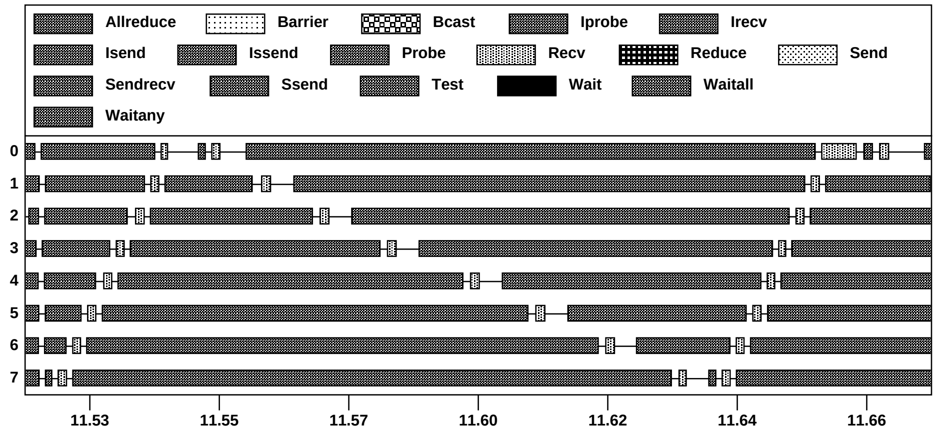


Figure 4.10
Communication in a single iteration for send followed by receive

'`intermediate`' contains a target for '`oned`'; this is the first version of our Poisson solver. To run it, we give the command

```
mpirun -np 8 oned
```

(This assumes that we are using the model MPI implementation.)

Using the MPE logging tools described in Section ??, we can get a graphical display of the communication. One output from `upshot` is shown in Figure 4.10. Note that the communication is entirely sequential! What went wrong?

Although this is a frequently used communication pattern, it is not a safe one, particularly for large values of `nx` (long messages). The reason is that the amount of parallelism depends in a subtle way on the amount of buffering provided by the message passing system, which is not explicitly specified by MPI and may be difficult to determine. Let us suppose that we run this on a system with a small amount of system buffer space or with a large message size. Then we will get the behavior displayed in Figure 4.10.

Looking at the `upshot` output gives us a clue to what has happened. The sends do not complete until the matching receives are issued on the destination process. Since one process (the "top" process) does not send to anyone in the first step, it can receive from the process below it, thus allowing that process to receive from below it, etc. This produces a staircase pattern of sends and receives. We illustrate this in Figure 4.11.

Before we go any further, we need to understand in more detail what happens when we ask MPI (or any message-passing system) to send a message.

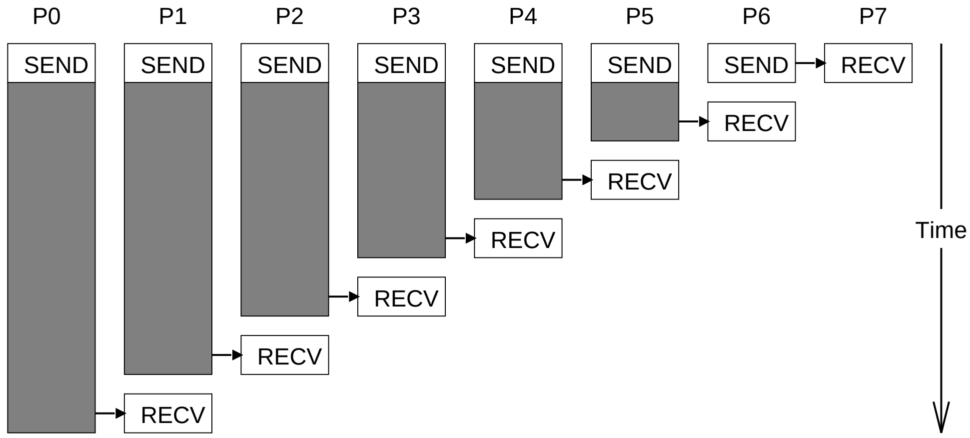


Figure 4.11
Sequentialization caused by sends blocking until the matching receive is posted. The shaded area indicates the time a process is idle, waiting for the send to be allowed to transfer data to the neighboring process

Consider the following code:

```
if (rank .eq. 0) then
    call MPI_Send( sbuffer, ..., 1, ... )
else
    call MPI_Recv( rbuffer, ..., 0, ... )
endif
```

What happens on the process with rank zero?

The easy answer is that the message in `sbuffer` is sent to process one. But what if process one is not ready to receive it? Perhaps process one is still computing a previous result. What can process zero do? There are three possibilities: process zero can stop and wait until process one is ready to receive the message, it can copy the message out of `sbuffer` into some internal buffer (which may be located on process zero, process one, or somewhere else) and return from the `MPI_Send` call, or it can fail.

There are good arguments for the first two of these. The argument for the second case is the easiest: as long as there is space available to hold a copy of the message, the message-passing system should provide this service to the programmer rather than forcing the process to stop dead in its tracks until the matching receive is called.

The argument for the first case is, in part, a rebuttal to this. What if there isn't enough space available? We don't want the computation to fail just because the matching receive has not yet been made; perhaps that `MPI_Recv` is about to be called. Since we cannot guarantee that there will be enough space to store a copy of an arbitrary message, why not simply say that we will never copy the message into internal storage?

The MPIF had long and impassioned discussions about these choices; in the end, both of these interpretations were allowed. That is, an MPI implementation is permitted to copy the message to be sent into internal storage in order to permit the `MPI_Send` to return, but it is not required to do so. If the MPI implementation does copy the send buffer into internal storage, we say that it *buffers* the data. Different buffering strategies provide differing levels of convenience and performance.

The third case is also interesting because it allows for certain performance improvements. The use of this kind of send operation is described in Section **??**.

For large applications that are already using a large amount of memory, even requiring the message-passing system to use all "available" memory may not provide enough memory to make this code work. For example, consider a value of `nx` in the above example that represents more memory space than is free on the process. There is no place to store this message on the sending process, and, until the receiver begins the matching receive, there is no place to store the message on the receiving process. For large applications that are already using large amounts of memory, the value of `nx` that triggers this situation may be quite small.

The performance problem shown here is even more dangerous. As we saw, the code runs (it does not deadlock) but it does not execute in parallel. All of these issues suggest that programmers should be aware of the pitfalls in assuming that the system will provide adequate buffering. In the next few sections, we will describe ways in which the MPI programmer can ensure that the correct parallel execution of a program does not depend on the amount of buffering, if any, provided by the message-passing system.

Ordered send and receive. One of the easiest ways to correct for a dependence on buffering is to order the sends and receives so that they are paired up. That is, the sends and receives are ordered so that if one process is sending to another, the destination will do a receive that matches that send before doing a send of its own. The code for this approach is shown in Figure 4.12. In this code, the even processes (in Cartesian coordinates) send first, and the odd processes receive first.

Figure 4.13 shows the communication pattern in a single iteration of the Jacobi code when using this approach.

Combined send and receive. The approach of pairing the sends and receives is effective but can be difficult to implement when the arrangement of processes is complex (for example, with an irregular grid). An alternative is to use the MPI routine `MPI_Sendrecv`. This routine allows you to send and receive data without worrying about deadlock from a lack of buffering. Each process then sends to the

```
      subroutine exchng1( a, nx, s, e, comm1d, nbrbottom, nbrtop )
      use mpi
      integer nx, s, e
      double precision a(0:nx+1,s-1:e+1)
      integer comm1d, nbrbottom, nbrtop, rank, coord
      integer status(MPI_STATUS_SIZE), ierr
!
      call MPI_COMM_RANK( comm1d, rank, ierr )
      call MPI_CART_COORDS( comm1d, rank, 1, coord, ierr )
      if (mod( coord, 2 ) .eq. 0) then
        call MPI_SEND( a(1,e), nx, MPI_DOUBLE_PRECISION, &
                       nbrtop, 0, comm1d, ierr )
        call MPI_RECV( a(1,s-1), nx, MPI_DOUBLE_PRECISION, &
                       nbrbottom, 0, comm1d, status, ierr )
        call MPI_SEND( a(1,s), nx, MPI_DOUBLE_PRECISION, &
                       nbrbottom, 1, comm1d, ierr )
        call MPI_RECV( a(1,e+1), nx, MPI_DOUBLE_PRECISION, &
                       nbrtop, 1, comm1d, status, ierr )
      else
        call MPI_RECV( a(1,s-1), nx, MPI_DOUBLE_PRECISION, &
                       nbrbottom, 0, comm1d, status, ierr )
        call MPI_SEND( a(1,e), nx, MPI_DOUBLE_PRECISION, &
                       nbrtop, 0, comm1d, ierr )
        call MPI_RECV( a(1,e+1), nx, MPI_DOUBLE_PRECISION, &
                       nbrtop, 1, comm1d, status, ierr )
        call MPI_SEND( a(1,s), nx, MPI_DOUBLE_PRECISION, &
                       nbrbottom, 1, comm1d, ierr )
      endif
      return
      end
```

Figure 4.12
Exchange routine with paired sends and receives

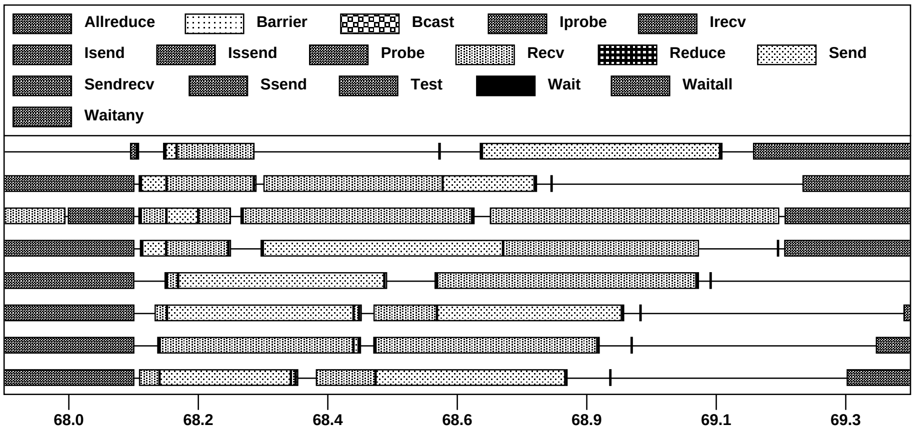


Figure 4.13
Communication in a single iteration with paired sends and receives

process below it and receives from the process above it. The code fragment for this sequence of operations is shown in Figure 4.14.

Buffered sends. Instead of requiring the programmer to determine a safe ordering of the send and receive operations, MPI allows the programmer to provide a buffer into which data can be placed until it is delivered (or at least left the buffer). The change to the exchange routine is simple; one just replaces the `MPI_SEND` calls with `MPI_BSEND`. The resulting routine is shown in Figure 4.15.

In addition to the change to the exchange routine, MPI requires that the programmer provide the storage into which the message may be placed with the routine `MPI_BUFFER_ATTACH`. This buffer should be large enough to hold all of the messages that must be sent before the matching receives are called. In our case, we need a buffer of `2*nx` double precision values. We can provide this with code like this:

```
double precision buffer(2*MAXNX+2*MPI_BSEND_OVERHEAD)
...
call MPI_BUFFER_ATTACH( buffer, 2*MAXNX*8+2*MPI_BSEND_OVERHEAD*8, ierr )
```

The “8” here is the number of bytes in a double-precision value; this value could also be computed with `MPI_Type_extent` as described in Chapter **??** or with the MPI-2 function `MPI_SIZEOF` (for Fortran only).

Note that extra space is allocated in the buffer. For each message that is sent with `MPI_Bsend`, an extra amount of space, of `MPI_BSEND_OVERHEAD` bytes, must be allocated. An MPI implementation uses this within the `MPI_Bsend` routine to

```
      subroutine exchng1( a, nx, s, e, comm1d, nbrbottom, nbrtop )
      use mpi
      integer nx, s, e
      double precision a(0:nx+1,s-1:e+1)
      integer comm1d, nbrbottom, nbrtop
      integer status(MPI_STATUS_SIZE), ierr
!
      call MPI_SENDRECV( &
               a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0, &
               a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, &
               comm1d, status, ierr )
      call MPI_SENDRECV( &
               a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1, &
               a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, &
               comm1d, status, ierr )
      return
      end
```

Figure 4.14
Exchange routine with send-receive

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

MPI_BUFFER_ATTACH(buffer, size, ierror)
<type> buffer(*)
integer size, ierror

MPI_BUFFER_DETACH(buffer, size, ierror)
<type> buffer(*)
integer size, ierror

Table 4.5
Fortran bindings for buffer operations

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

int **MPI_Buffer_attach**(void* buffer, int size)

int **MPI_Buffer_detach**(void** buffer, int* size)

Table 4.6
C bindings for buffering routines

```
      subroutine exchng1( a, nx, s, e, comm1d, nbrbottom, nbrtop )
      use mpi
      integer nx, s, e
      double precision a(0:nx+1,s-1:e+1)
      integer comm1d, nbrbottom, nbrtop
      integer status(MPI_STATUS_SIZE), ierr
!
      call MPI_BSEND( a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0, &
                      comm1d, ierr )
      call MPI_RECV( a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, &
                     comm1d, status, ierr )
      call MPI_BSEND( a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1, &
                      comm1d, ierr )
      call MPI_RECV( a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, &
                     comm1d, status, ierr )
      return
      end
```

Figure 4.15
Exchange routine with buffered sends

manage the buffer space and the communication (e.g., it may allocate an `MPI_Request` within the buffer area). The code here actually allocates eight times as much space for this overhead than is needed; this is done to simplify the declaration, which would otherwise have to be something like

```
double precision buffer(2*MAXNX+2*((MPI_BSEND_OVERHEAD+7)/8))
```

Once a program no longer needs to use a buffer (or wishes to reclaim it for other use), the routine `MPI_BUFFER_DETACH` should be called. The bindings for these routines are shown in Tables 4.5 and 4.6. (The reason for the `buffer` and `size` arguments in `MPI_Buffer_detach` is that C programmers can use these to find out the location and size of an existing buffer.) It is important to remember that buffering may incur a performance penalty (extra effort copying data to and from the buffer), and may expose the program to failure if the amount of needed buffering is not calculated correctly.

Another approach involves using communications operations that do not block but permit the program to continue to execute instead of waiting for communications to complete. This also allows the program to compute while waiting for the communications to complete. However, the ability to overlap computation with communication is just one reason to consider the nonblocking operations in MPI; the ability to prevent one communication operation from preventing others from finishing is just as important.

4.4 Using Nonblocking Communications

On most parallel computers, moving data from one process to another takes more time than moving or manipulating data within a single process. For example, on one modern parallel computer, each process can compute up to 125 million floating-point results per second, but can move fewer than one million words per second between processes.[2] To keep a program from being slowed down (also described as "starved for data"), many parallel computers allow users to start sending (and receiving) several messages and to proceed with other operations. Programmers who have used "asynchronous I/O" will recognize this approach as a way of compensating for the relatively slow speed of access to external information (disks in the case of I/O, another process in the case of message passing). MPI supports this approach by providing *nonblocking* sends and receives.

The routine `MPI_Isend` begins the nonblocking send operation. The arguments are the same as for `MPI_Send` with the addition of a *handle* as the next to last

[2] This apparent mismatch of capabilities reflects underlying engineering and physical realities and is a major reason that the message-passing approach, which keeps the programmer reminded of the cost of accessing remote data, has been a successful way to program parallel computers.

argument (the last argument in C). The two routines also behave similarly except that, for MPI_Isend, the buffer containing the message to be sent must not be modified until the message has been delivered (more precisely, until the operation is complete, as indicated by one of the MPI_Wait or MPI_Test routines). In C, the type of this handle is MPI_Request.

The *handle* argument is used to determine whether a message has been delivered. The easiest way to test is with MPI_Test:

```
    call MPI_ISEND( buffer, count, datatype, dest, tag, &
                    comm, request, ierr )
     < do other work >
10  call MPI_TEST( request, flag, status, ierr )
    if (.not. flag) goto 10
```

Often, one wishes to wait until the send completes. Rather than writing the loop in the previous example, one can use MPI_Wait instead:

```
    call MPI_WAIT( request, status, ierr )
```

The routine MPI_Irecv begins the nonblocking receive operation. It has one additional argument, the handle, just as MPI_Isend does. It also has one *less* argument: the status argument, which is used to return information on the completed receive, is deleted from the argument list. Just as for MPI_Isend, MPI_Test may be used to test for the completion of a receive started with MPI_Irecv, and MPI_Wait may be used to wait for the completion of such a receive. The status arguments of these two routines return the information on the completed receive in the same form as MPI_Recv does for a blocking receive.

In many cases, one wishes to test or wait for many nonblocking operations. Although one can simply loop through the operations, this approach is inefficient, since this forces the user's program to be constantly executing rather than waiting (possibly without consuming CPU time) for the "next" message. MPI provides a way to wait for all or any of a collection of nonblocking operations (with MPI_Waitall and MPI_Waitany) and to test all or any of a collection of nonblocking operations (with MPI_Testall and MPI_Testany). For example, to start two nonblocking receives and then wait on them, one can use

```
call MPI_IRECV( ..., requests(1), ierr )
call MPI_IRECV( ..., requests(2), ierr )
...
call MPI_WAITALL( 2, requests, status, ierr )
```

Here, status must be an array of two MPI_status objects; it can be declared with

```
integer status(MPI_STATUS_SIZE,2)
```

With these routines, we can rewrite the exchange routine `exchng1` using nonblocking operations, as shown in Figure 4.16. This approach allows for both sends and receives to take place at the same time. In principle, this approach can be almost twice as fast as the version in Figure 4.12, though few existing systems support this (don't forget that MPI was designed to support current and future message-passing systems). In the next section, we try them out and see what happens.

```
        subroutine exchng1( a, nx, s, e, comm1d, &
                            nbrbottom, nbrtop )
        use mpi
        integer nx, s, e
        double precision a(0:nx+1,s-1:e+1)
        integer comm1d, nbrbottom, nbrtop
        integer status_array(MPI_STATUS_SIZE,4), ierr, req(4)
!
        call MPI_IRECV ( &
             a(1,s-1), nx, MPI_DOUBLE_PRECISION, nbrbottom, 0, &
             comm1d, req(1), ierr )
        call MPI_IRECV ( &
             a(1,e+1), nx, MPI_DOUBLE_PRECISION, nbrtop, 1, &
             comm1d, req(2), ierr )
        call MPI_ISEND ( &
             a(1,e), nx, MPI_DOUBLE_PRECISION, nbrtop, 0, &
             comm1d, req(3), ierr )
        call MPI_ISEND ( &
             a(1,s), nx, MPI_DOUBLE_PRECISION, nbrbottom, 1, &
             comm1d, req(4), ierr )
!
        call MPI_WAITALL ( 4, req, status_array, ierr )
        return
        end
```

Figure 4.16
Row-exchange routine using nonblocking operations

Note that in order to overlap communication and computation, we must make further changes to our program. In particular, we need to change the `sweep` program to allow us to do some of the work while we wait for data to arrive. We will come back to nonblocking operations in Section 4.9, where we discuss overlapping communication with computation.

P	Blocking Send	Ordered Send	Sendrecv	Buffered Bsend	Noblock Isend
1	5.38	5.54	5.54	5.38	5.40
2	2.77	2.88	2.91	2.75	2.77
4	1.58	1.56	1.57	1.50	1.51
8	1.15	0.947	0.931	0.854	0.849
16	1.18	0.574	0.534	0.521	0.545
32	1.94	0.443	0.451	0.452	0.397
64	3.73	0.447	0.391	0.362	0.391

Table 4.7
Timings for variants of the 1-D decomposition of the Poisson problem

4.5 Synchronous Sends and "Safe" Programs

What can we do to ensure that a program does not depend on buffering? In general, this is equivalent to the Turing halting-problem and is hence unanswerable, but in many special cases it is possible to show that if the program runs successfully with no buffering, it will run with any amount of buffering. MPI provides a way to send a message in such a way that the send does not return until the destination begins to receive the message. The routine is `MPI_Ssend`. The arguments to this send are identical to `MPI_Send`. Note that it is permissible for an MPI operation to implement `MPI_Send` with `MPI_Ssend`; thus, for maximum portability, one should ensure that any use of `MPI_Send` can be replaced with `MPI_Ssend`. Programs that do not require buffering (or, with `MPI_Bsend`, only the amount of buffering made available) for their correct operation are sometimes called *safe*.

4.6 More on Scalability

Because we have encapsulated the routines to exchange data between the processes, it is a simple matter to link the program with different methods and to compare them. Table 4.7 shows the results for one particular parallel computer and MPI implementation.

Before going on, we note that each iteration of the loop calls `MPI_Allreduce`. This routine can take a significant amount of time; on networks of workstations, it can take many milliseconds. In many applications, the effective cost of the `MPI_Allreduce` is reduced by taking a number of iterations without computing the difference between successive iterates; this approach works because the Jacobi method converges very slowly.

A quick examination of Table 4.7 reveals a number of interesting features. First, with the exception of the blocking sends, the performance of the other methods is roughly the same. The blocking sends case shows the lack of parallelism in the communication; with thirty two processes, the computation actually takes longer than with four processes. More serious is the poor performance of the other methods. At 64 processes, the program is running only about fourteen times as fast as for a single process, for an efficiency of about 20%.

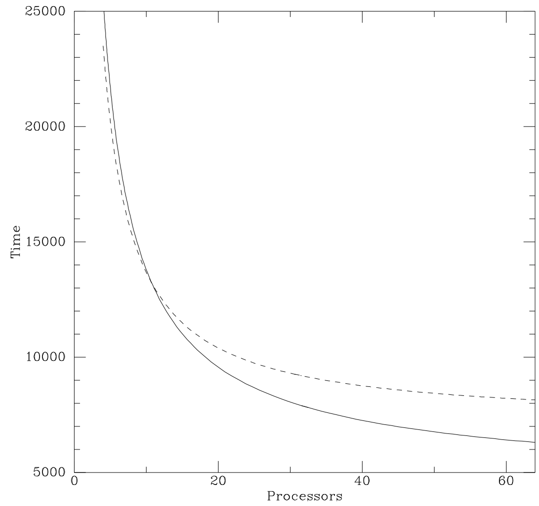


Figure 4.17
Predicted time for a 1-D (dashed) and 2-D (solid) decomposition of the 2-D Poisson problem

To understand the performance of these methods, we perform a simple scalability analysis similar to the one in Chapter ??. We will need a slightly more sophisticated model of communication cost than used in Chapter ??. There, we used T_{comm} as the time to send a word. We will replace this with $s + rn$ as the time to send n bytes; for n large, we have $T_{comm} \approx r * 8$. The term s is the latency or startup time; it can be thought of as the time to send a message containing no data beyond the message tag and source. The term r is the inverse rate; it is the time to send a single byte and is given by one over the bandwidth. For example, if the bandwidth of a connection is 10 MB/sec, $r = 1/(10\text{MB/sec}) = 10^{-7}$ sec/byte. Using this formula, we can easily see that the `exchng1d` routine in Figure 4.12 takes roughly $2(s + rn)$ time, where $n = 8$ `nx` (assuming 8-byte double-precision data).

In a two-dimensional decomposition, let there be p_x processes in the x direction and p_y processes in the y direction, with each process having the same number of mesh points. Then, with the exception of the processes on any edge of the domain, the amount of communication T_c is

$$T_c = 2\left(s + r\frac{n}{p_x}\right) + 2\left(s + r\frac{n}{p_y}\right).$$

If $p_x = p_y = \sqrt{p}$, this takes a particularly simple form:

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

Figure 4.17 shows the expected performance for both 1-D and 2-D decompositions of the domain, based on these estimates. The situation is even more extreme in 3-D, as shown in Figure 4.18. Note that, for small numbers of processes, this model suggests that a 1-D decomposition is actually better than a 2-D decomposition. Care must be exercised here, as our analysis is good in the 2-D case only for $p \geq 9$, since for $p < 9$, no process has four neighbors. More details on analyzing the communication in parallel codes for PDEs may be found in [4, 5]; an example of using these techniques to analyze a large application can be found in [1].

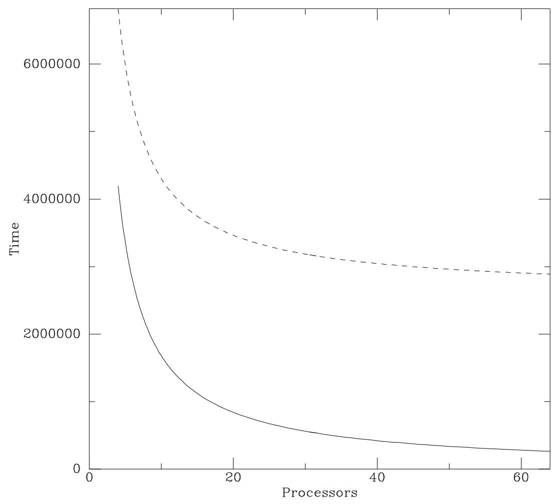


Figure 4.18
Predicted time for a 1-D (dashed) and 3-D (solid) decomposition of the 3-D Poisson problem

4.7 Jacobi with a 2-D Decomposition

Figures 4.17 and 4.18 and this scalability analysis suggest that we need to rewrite our program to use these higher-dimensional decompositions. Fortunately, MPI makes this task relatively easy. We will show how a few modifications to the program in Figure 4.9 changes it from a one-dimensional to two-dimensional decomposition.

First, we let MPI compute the decomposition of the domain for us with `MPI_Cart_create`:

```
      periods(1) = .false.
      periods(2) = .false.
      reorder    = .true.
      call MPI_CART_CREATE( MPI_COMM_WORLD, 2, dims, periods, &
                            reorder, comm2d, ierr )
```

Compare this with the 1-D code in Section 4.2.

Next, we get left and right neighbors as well as the top and bottom neighbors:

```
      call MPI_CART_SHIFT( comm2d, 0,  1, nbrleft, nbrright, ierr )
      call MPI_CART_SHIFT( comm2d, 1,  1, nbrbottom, nbrtop, ierr )
```

We change the body of the sweep routine to

```
      integer i, j, n
      double precision u(sx-1:ex+1,sy-1:ey+1), &
                       unew(sx-1:ex+1,sy-1:ey+1)
      do 10 j=sy, ey
         do 10 i=sx, ex
            unew(i,j) = &
                  0.25*(u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j)) - &
                     h * h * f(i,j)
 10   continue
```

The last routine that we need to change is the data exchange routine (`exchng1d` in the 1-D examples). This is a little more difficult because, while the data sent to the top and bottom processes is stored contiguously in memory, the data sent to the left and right processes is not.

4.8 An MPI Derived Datatype

One of MPI's novel features is its use of a datatype associated with every message. Specifying the length of a message as a given count of occurrences of a given datatype is more portable than using length in bytes, since lengths of given types may vary from one machine to another. It also allows MPI to provide translations between machine formats.

So far all of our messages have consisted of contiguous areas in memory, so the basic datatypes such as `MPI_INTEGER` and `MPI_DOUBLE_PRECISION`, accompanied by a count, have been sufficient to describe our messages. In this section we introduce MPI's *derived datatypes*, which allow us to specify *noncontiguous* areas in memory, such as a row of an array stored columnwise (or, in our case, a column of an array stored rowwise).

This is a common situation, and MPI provides a mechanism for describing this kind of data layout. We begin by defining a new datatype that describes a group of elements that are separated by a constant amount in memory (a constant *stride*). We do this with the `MPI_Type_vector`:

```
call MPI_TYPE_VECTOR( ey + 1 - (sy - 1), 1, ex + 1 - (sx - 1), &
                      MPI_DOUBLE_PRECISION, stridetype, ierr )
call MPI_TYPE_COMMIT( stridetype, ierr )
```

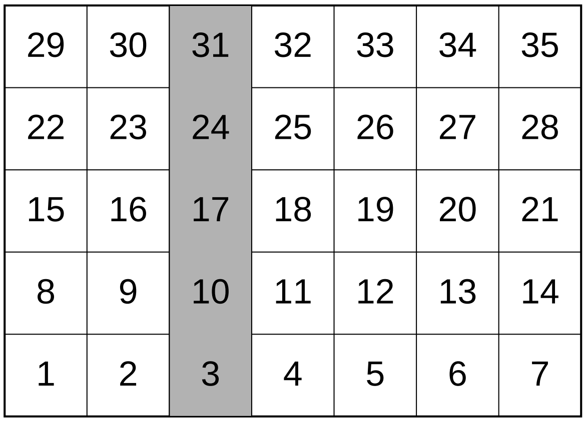
The arguments to `MPI_Type_vector` describe a *block*, which consists of a number of (contiguous) copies of the input datatype given by the second argument. The first argument is the number of blocks; the second is the number of elements of the old datatype in each block (this is often one). The old datatype is the fourth argument. The third argument is the *stride*; this is the distance in terms of the extent of the input datatype between successive elements. The fifth argument is the created derived datatype. In this example, there is one double precision item per block; the double-precision values are `ex + 1 - (sx - 1)` apart, and there are `ey + 1 - (sy - 1)` of them. Figure 4.19 illustrates an MPI vector datatype.

Note that after the new datatype is created with the `MPI_Type_vector` command, we *commit* it to the system with `MPI_Type_commit`. This routine takes the newly constructed datatype and gives the system the opportunity to perform any performance optimizations that it may wish to. All user-constructed data types must be committed before they can be used.

With this new datatype definition, the MPI code for sending a row differs from the code for sending a column only in the datatype argument. The final version of `exchng2d` is shown in Figure 4.20.

When a datatype is no longer needed, it should be freed with `MPI_Type_free`. The datatype variable (the first argument) is set to `MPI_TYPE_NULL` by `MPI_Type_free`. Bindings for these routines are shown in Tables 4.8 and 4.9.

An alternative definition of the strided type is described in Section **??** that allows the programmer to send any number of elements with the same datatype.




```
MPI_TYPE_VECTOR( 5, 1, 7, MPI_DOUBLE_PRECISION, newtype, ierr )
```

Figure 4.19
A strided data item (shaded) and its MPI definition. Numbers indicate consecutive memory locations.

MPI_TYPE_VECTOR(count, blocklength, stride, oldtype, newtype, ierror)
    integer count, blocklength, stride, oldtype, newtype, ierror

MPI_TYPE_COMMIT(datatype, ierror)
    integer datatype, ierror

MPI_TYPE_FREE(datatype, ierror)
    integer datatype, ierror

Table 4.8
Fortran bindings for elementary MPI datatype routines

int **MPI_Type_vector**(int count, int blocklength, int stride, MPI_Datatype oldtype,
    MPI_Datatype *newtype)

int **MPI_Type_commit**(MPI_Datatype *datatype)

int **MPI_Type_free**(MPI_Datatype *datatype)

Table 4.9
C bindings for MPI elementary datatype routines

```
       subroutine exchng2( a, sx, ex, sy, ey, &
                           comm2d, stridetype, &
                           nbrleft, nbrright, nbrtop, nbrbottom  )
       use mpi
       integer sx, ex, sy, ey, stridetype
       double precision a(sx-1:ex+1, sy-1:ey+1)
       integer nbrleft, nbrright, nbrtop, nbrbottom, comm2d
       integer status(MPI_STATUS_SIZE), ierr, nx
!
       nx = ex - sx + 1
!  These are just like the 1-d versions, except for less data
        call MPI_SENDRECV( a(sx,ey),  nx, MPI_DOUBLE_PRECISION, &
                          nbrtop, 0, &
                          a(sx,sy-1), nx, MPI_DOUBLE_PRECISION, &
                          nbrbottom, 0, comm2d, status, ierr )
        call MPI_SENDRECV( a(sx,sy),  nx, MPI_DOUBLE_PRECISION, &
                          nbrbottom, 1, &
                          a(sx,ey+1), nx, MPI_DOUBLE_PRECISION, &
                          nbrtop, 1, comm2d, status, ierr )
!
! This uses the "strided" datatype
        call MPI_SENDRECV( a(ex,sy),  1, stridetype, nbrright, 0, &
                           a(sx-1,sy), 1, stridetype, nbrleft, 0, &
                           comm2d, status, ierr )
        call MPI_SENDRECV( a(sx,sy),  1, stridetype, nbrleft,   1, &
                           a(ex+1,sy), 1, stridetype, nbrright, 1, &
                           comm2d, status, ierr )
        return
        end
```

Figure 4.20
Two-dimensional exchange with sendrecv

4.9 Overlapping Communication and Computation

Because of the time it can take to move data from one process to another, it is often advantageous to arrange the program so that some work can be done while the messages are "in transit." So far, we have used nonblocking operations to avoid deadlock in the communications. Here we describe some of the details in arranging a program so that computation and communication can take place simultaneously.

In the Jacobi method, the values of `unew` at points of the mesh that are interior to the domain on each process may be computed without needing any data from any other process; this is shown in Figure 4.21.

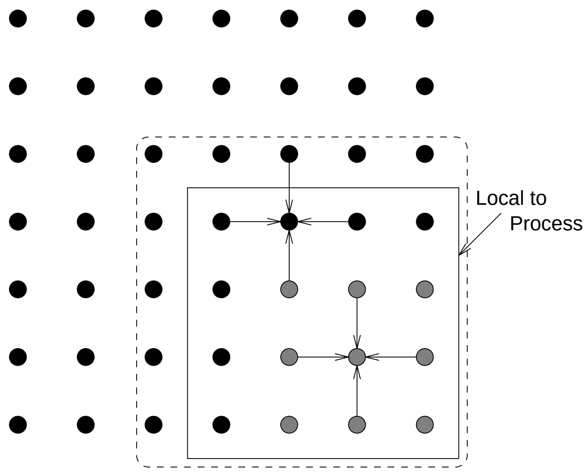


Figure 4.21
The shaded points show those mesh points whose computation does not depend on any data from other processes. The local domain is given by the solid outline; the domain with ghost points is given by the dashed box.

We can arrange our computational task as follows: (1), indicate where to receive data from other processes; (2), begin sending data to the other processes; (3), compute with the local data; and (4), receive data from the other processes and finish computing with it. Separating the code into these four steps does increase the amount of code, but all of it is easily derived from our existing code. For example, Step 3 is given by the code in Figure 4.22.

All that we have done here is to select the part of the domain that is one away from the edges of the local domain, ensuring that all of the data is available.

The communication routines change in similar ways; in addition, the calls are split into two sections of code, separated by the main computational section. The first part begins both the nonblocking sends and nonblocking receives:

```
      integer i, j, n
      double precision u(sx-1:ex+1,sy-1:ey+1), &
                       unew(sx-1:ex+1,sy-1:ey+1)
      do 10 j=sy+1, ey-1
         do 10 i=sx+1, ex-1
            unew(i,j) = &
                  0.25*(u(i-1,j)+u(i,j+1)+u(i,j-1)+u(i+1,j)) - &
                     h * h * f(i,j)
 10   continue
```

Figure 4.22
Code to compute a Jacobi iteration with only local data

```
    call MPI_IRECV( ..., requests(1), ierr )
    ...
    call MPI_ISEND( ..., requests(5), ierr )
```

These are followed by[3]

```
    do 100 k=1,8
        call MPI_WAITANY( 8, requests, idx, status, ierr )
!       Use tag to determine which edge
        goto (1,2,3,4,100,100,100,100), status(MPI_TAG,idx)
1       do 11 j=sy,ey
11          unew(si,j) = ...
        goto 100
2       do 21 j=sy,ey
21          unew(ei,j) = ...
        goto 100
3       do 31, i=sx,ex
31          unew(i,ej) = ...
        goto 100
4       do 41 i=sx,ex
41          unew(i,sj) = ...
        goto 100
100  continue
```

Here, `requests(1)` through `requests(4)` are receive handles, and `requests(5)` through `requests(8)` are send handles.

[3]For C programmers: the construction `goto (...), variable` is similar to the C `switch (variable)` except that the values are labels and the variable is in the range 1 to the number of labels.

MPI_BARRIER(comm,ierr) integer comm,ierr

Table 4.10
Fortran bindings for barrier routine

int **MPI_Barrier**(MPI_Comm comm)

Table 4.11
C bindings for barrier routine

4.10 More on Timing Programs

Our simple code can also be used to solve time-dependent PDEs. Many discretizations for time-dependent PDEs require only the data that our Jacobi iteration needs. The only difference is that the `sweep` routines change and there is no longer any need for the `MPI_Allreduce` to check for convergence. However, this does raise an issue when timing this program: how do we know that everyone is done when we call `MPI_Wtime`? We can use the MPI routine `MPI_Barrier` to ensure that everyone has completed the computation. A barrier is a special collective operation that does not let the process continue until all processes in the communicator have called `MPI_Barrier`.

```
call MPI_Barrier( MPI_COMM_WORLD, ierr )
t1 = MPI_Wtime()
<do work>
call MPI_Barrier( MPI_COMM_WORLD, ierr )
total_time = MPI_Wtime() - t1
```

The barriers ensure that all processes have reached the same point in the code and are ready to proceed. Many of the collective operations (e.g., `MPI_Allreduce`) have the same property; that is, no process can exit the operation until all processes have entered. Note that this is not true for operations like `MPI_Reduce`, where only the root process must wait for all other processes to enter the `MPI_Reduce` call. The bindings for `MPI_Barrier` are shown in Tables 4.10 and 4.11.

4.11 Three Dimensions

So far, we have restricted ourselves to the 2-D problem. Now, we want to introduce the 3-D case, and also show the problem in a C implementation. Even a relatively

MPI_DIMS_CREATE(nnodes,ndims,dims,ierr) integer nnodes, ndims, dims(*),ierr

Table 4.12
Fortran binding for `MPI_DIMS_CREATE`

int **MPI_Dims_create**(int nnodes, int ndims, int *dims)

Table 4.13
C binding for `MPI_DIMS_CREATE`

small 3-D problem having 100 grid points on a side involves 10^6 grid points; it is common to encounter one or two orders of magnitude more grid points than this in 3-D simulations. These problems are consequently ideal candidates for parallel computing. The flexibility of MPI makes generalizing our previous 2-D problem solutions to 3-D straightforward.

One complication in developing a 3-D code is the decomposition of the domain among the processes. As we would expect from the scalability analysis in this chapter, we should use a 3-D virtual topology. MPI provides the routine `MPI_Dims_create` to help in generating a Cartesian virtual topology with any number of dimensions. This routine takes as input the total number of processes and the number of dimensions, and returns an array containing the Cartesian dimensions. The bindings are shown in Tables 4.12 and 4.13. Sample code for a 3-D problem is available in the directory '`intermediate/3d`' and may be downloaded according to the directions in Appendix **??**.

4.12 Application: Simulating Vortex Evolution in Superconducting Materials

As an example of an application that uses the techniques described in this chapter, we briefly discuss a model of superconductivity.

The time-dependent Ginzburg-Landau (TDGL) equation can be used for the numerical simulation of vortex dynamics and phase transitions in type-II superconductors. The TDGL equation provides a phenomenological description of the macroscopic properties of high-temperature superconductors and has been remarkably successful in explaining experimental results on phase transitions. The TDGL equation is a partial differential equation for the complex-valued order parameter with stochastic coefficients. This model uses a field description of vortices as described by the time-dependent Ginzburg-Landau equation in three dimensions.

A group at Argonne National Laboratory [2] has parallelized a three-dimensional

TDGL program by partitioning the superconductor data structure among the processes. Each processes' memory contains a "cubelet" of the original global data structure. Each process is then responsible for the time integration over its part of the superconductor. The time integration is done using the forward Euler technique. Finite differences are used to approximate the derivatives using a box (27-point) stencil.

Using this program, researchers have been able to answer several questions about the behavior of high-temperature superconductors. The computations require very high resolution.

The update step for each cell requires values from neighboring cells. Because of the array decomposition, neighbors of some of the cells that a process has require values from cells stored in other processes' memories. To communicate these values, routines similar to the exchange routines described for the Poisson problem are used.

This chapter was processed by LaTeX on April 26, 1999.

References

[1] I. Foster, W. Gropp, and R. Stevens. The parallel scalability of the spectral transform method. *Monthly Weather Review*, 120(5):835–850, 1992.

[2] N. Galbreath, W. Gropp, D. Gunter, G. Leaf, and D. Levine. Parallel solution of the three-dimensional, time-dependent Ginzburg-Landau equation. In *Proceedings of the SIAM Parallel Processing for Scientific Computing Conference*. SIAM, 1993.

[3] John R. Gilbert and Robert Schreiber. Optimal data placement for distributed memory architectures. In *Parallel Processing for Scientific Computing*, pages 462–471. SIAM, March 1991.

[4] W. Gropp and E. Smith. Computational fluid dynamics on parallel processors. *Computers and Fluids*, 18:289–304, 1990.

[5] William D. Gropp. Solving PDEs on loosely-coupled parallel processors. *Par. Comput.*, 5:165–173, 1987.

[6] William D. Gropp and I. C. F. Ipsen. Recursive mesh refinement on hypercubes. *BIT*, pages 186–211, 1989.

[7] William D. Gropp and Barry Smith. Users manual for KSP: Data-structure-neutral codes implementing Krylov space methods. Technical Report ANL-93/30, Argonne National Laboratory, August 1993.

[8] C.-T. Ho and L. Johnsson. On the embedding of arbitrary meshes in Boolean cubes with expansion two dilation two. In *Proc. 1987 ICPP*, pages 188–191, 1987.

[9] L. Johnsson. Communication efficient basic linear algebra computations on hypercube architectures. *J. Par. Dist. Comp.*, 4:133–172, 1987.

[10] Leon Lapidus and George F. Pinder. *Numerical Solution of Partial Differential Equations in Science and Engineering*. Wiley-Interscience, 1982.

[11] Piyush Mehrotra, Joel Saltz, and Robert Voigt, editors. *Unstructured Scientific Computation on Scalable Multiprocessors*. MIT Press, 1992.

Subject Index

Function and Term Index