Using MPI

Portable Parallel Programming with the Message-Passing Interface

William Gropp
Ewing Lusk
Anthony Skjellum

The MIT Press
Cambridge, Massachusetts
London, England

3 Using MPI in Simple Programs

In this chapter we introduce the most basic MPI calls and use them to write some simple parallel programs. Simplicity of a parallel algorithm does not limit its usefulness, however: even a small number of basic routines are enough to implement a major application. We also demonstrate in this chapter a few of the tools that we use throughout this book to study the behavior of parallel programs.

3.1 A First MPI Program

For our first parallel program, we choose a "perfect" parallel program: it can be expressed with a minimum of communication, load balancing is automatic, and we can verify the answer. Specifically, we compute the value of π by numerical integration. Since

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

we will integrate the function $f(x) = 4/(1+x^2)$. To do this integration numerically, we divide the interval from 0 to 1 into some number n of subintervals and add up the areas of the rectangles as shown in Figure 3.1 for $n = 5$. Larger values of the parameter n will give us more accurate approximations of π. This is not, in fact, a very good way to compute π, but it makes a good example.

To see the relationship between n and the error in the approximation, we write an interactive program in which the user supplies n and the program first computes an approximation (the parallel part of the program) and then compares it with a known, highly accurate approximation to π.

The parallel part of the algorithm occurs as each process computes and adds up the areas for a different subset of the rectangles. At the end of the computation, all of the local sums are combined into a global sum representing the value of π. Communication requirements are consequently simple. One of the processes (we'll call it the master) is responsible for communication with the user. It obtains a value for n from the user and broadcasts it to all of the other processes. Each process is able to compute which rectangles it is responsible for from n, the total number of processes, and its own rank. After reporting a value for π and the error in the approximation, the program asks the user for a new value for n.

The complete program is shown in Figure 3.2. In most of this book we will show only the "interesting" parts of programs and refer the reader to other sources for the complete, runnable version of the code. For our first few programs, however, we include the entire code and describe it more or less line by line. In the directory

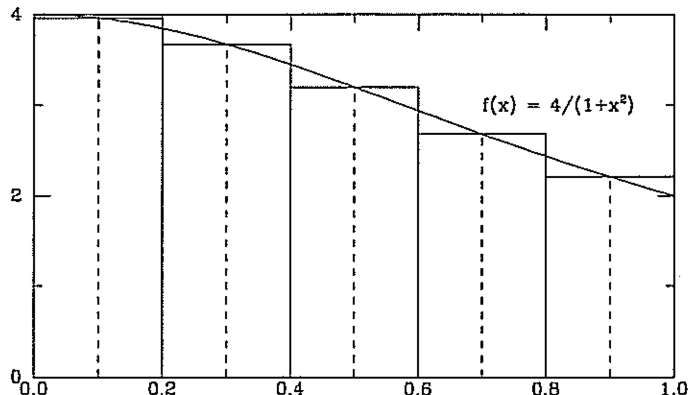

Figure 3.1
Integrating to find the value of π

of programs that accompanies this book, the pi program is available as 'simplempi/pi.f'. See Appendix D for details of how to obtain this code, other examples, and a model implementation of MPI. Instructions for running the model implementation of MPI are given in Appendix B.

Our program starts like any other, with the program main statement. The

```
include "mpif.h"
```

is necessary in every MPI Fortran program and subprogram to define various constants and variables. For Fortran compilers that do not support the include directive, the contents of this file must be inserted by hand into each function and subroutine that uses MPI calls.

After a few lines of variable definitions, we get to three lines that will probably be found near the beginning of every Fortran MPI program:

```
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
```

```
      program main
      include "mpif.h"
      double precision  PI25DT
      parameter         (PI25DT = 3.141592653589793238462643d0)
      double precision  mypi, pi, h, sum, x, f, a
      integer n, myid, numprocs, i, ierr
c                                 function to integrate
      f(a) = 4.d0 / (1.d0 + a*a)

      call MPI_INIT(ierr)
      call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)

 10   if ( myid .eq. 0 ) then
         print *, 'Enter the number of intervals: (0 quits) '
         read(*,*) n
      endif
c                                 broadcast n
      call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
c                                 check for quit signal
      if ( n .le. 0 ) goto 30
c                                 calculate the interval size
      h = 1.0d0/n
      sum  = 0.0d0
      do 20 i = myid+1, n, numprocs
         x = h * (dble(i) - 0.5d0)
         sum = sum + f(x)
 20   continue
      mypi = h * sum
c                                 collect all the partial sums
      call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
     $     MPI_COMM_WORLD,ierr)
c                                 node 0 prints the answer.
      if (myid .eq. 0) then
         print *, 'pi is ', pi, ' Error is', abs(pi - PI25DT)
      endif
      goto 10
 30   call MPI_FINALIZE(ierr)
      stop
      end
```

Figure 3.2
Fortran program for calculating π

The call to MPI_INIT is required in every MPI program and must be the first MPI call.[1] It establishes the MPI "environment." Only one invocation of MPI_INIT can occur in each program execution. Its only argument is an error code. Every Fortran MPI subroutine returns an error code in its last argument, which is either MPI_SUCCESS or an implementation-defined error code. In this example (and in many of our examples) we will be sloppy and not test the return codes from our MPI routines, assuming that they will always be MPI_SUCCESS. This approach will improve readability of the code at the expense of possible debugging time. We will discuss later (in Section 7.7) how to check, handle, and report errors.

As described in Chapter 2, all MPI communication is associated with a *communicator* that describes the communication context and an associated group of processes. In this program we will be using only the default communicator, predefined and named MPI_COMM_WORLD, that defines one context and the set of all processes. MPI_COMM_WORLD is one of the items defined in 'mpif.h'.

The call MPI_COMM_SIZE returns (in numprocs) the number of processes that the user has started for this program. Precisely how the user caused these processes to be started depends on the implementation, but any program can find out this number with this call. The value numprocs is actually the size of the group associated with the default communicator MPI_COMM_WORLD. We think of the processes in any group as being numbered with consecutive integers beginning with 0, called *ranks*. Each process finds out its rank in the group associated with a communicator by calling MPI_COMM_RANK. Thus although each process in this program will get the same number in numprocs, each will have a different number for myid.

Next, the master process (which can identify itself by using myid) gets a value for n, the number of rectangles, from the user. The line

```
call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
```

sends the value of n to all other processes. The MPI_BCAST results in every process (in the groups associated with the communicator given in the fifth argument) ending up with a copy of n. The data to be communicated is described by the address (n), the datatype (MPI_INTEGER), and the number of items (1). The process with the original copy is specified by the fourth argument (0 in this case, the master process, which just reads it from the user). (MPI assigns a type to every data item. MPI datatypes are described in full in Section 5.1.)

Thus, after the call to MPI_BCAST, all processes have n and their own identifiers, which is enough information for each one to compute its contribution, mypi. Each

[1] An exception is the MPI_Initialized routine, which a library can call to determine whether MPI_Init has been called or not. See Section 7.8.2.

process computes the area of every numprocs'th rectangle, starting with myid+1. Next, all of the values of mypi held by the individual processes need to be added up. MPI provides a rich set of such operations, using the MPI_REDUCE routine, with an argument specifying which arithmetic or logical operation is being requested. In our case the call is

```
      call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,
     $     MPI_COMM_WORLD,ierr)
```

The first two arguments identify the source and result addresses, respectively. The data being collected consists of 1 (third argument) item of type MPI_DOUBLE_PRECISION (fourth argument). The operation is addition (MPI_SUM, the next argument), and the result of the operation is to be placed in pi on the process with rank 0 (fifth argument). The last two arguments are the communicator and error return code, as usual. The first two arguments of MPI_Reduce must not overlap (i.e., must be different variables or sections of an array). A full list of the operations is presented in Section 7.3.2; user-defined operations are discussed in Section 7.3.3.

All processes then return to the top of the loop (the master prints the answer first). The MPI_BCAST causes all the processes except the master to wait for the next value of n.

When the user types a zero in response to the request for a number of rectangles, the loop terminates and all processes execute

```
      call MPI_FINALIZE(ierr)
```

This call must be made by every process in an MPI computation. It terminates the MPI "environment"; no MPI calls may be made by a process after its call to MPI_FINALIZE. (MPI_INIT cannot be called again.)

The Fortran bindings for the MPI routines used in this section are summarized in Table 3.1. In the tables of Fortran bindings, the expression <type> stands for any Fortran datatype, such as INTEGER or DOUBLE PRECISION.

3.2 Running Your First MPI Program

The way in which MPI programs are "launched" on a particular machine or network is not itself part of the MPI standard. Therefore it may vary from one machine to another. On (most of) the supported machines, one would use

```
mpirun -np 4 pi
```

MPI_INIT(ierror)   integer ierror
MPI_COMM_SIZE(comm, size, ierror)   integer comm, size, ierror
MPI_COMM_RANK(comm, rank, ierror)   integer comm, rank, ierror
MPI_BCAST(buffer, count, datatype, root, comm, ierror)   <type> buffer(*)   integer count, datatype, root, comm, ierror
MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)   <type> sendbuf(*), recvbuf(*)   integer count, datatype, op, root, comm, ierror
MPI_FINALIZE(ierror)   integer ierror

Table 3.1
Fortran bindings for routines used in the pi program

to run the program with four processes when using the portable, model implementation of MPI described in Appendix B.

3.3 A First MPI Program in C

In this section we repeat the program for computing the value of π in C rather than Fortran. In general, every effort has been made to keep the Fortran and C bindings similar. The primary difference is that error codes are returned as the value of C functions instead of in a separate argument. In addition, the arguments to most functions are more strongly typed than they are in Fortran, having specific C types such as `MPI_Comm` and `MPI_Datatype` where Fortran has integers. The included file is, of course, different: '`mpi.h`' instead of '`mpif.h`'. Finally, the arguments to `MPI_Init` are different, so that a C program can take advantage of command-line arguments. An MPI implementation is expected to remove from the `argv` array any command-line arguments that should be processed by the implementation before returning control to the user program and to decrement `argc` accordingly. Note