MPI Tutorial

Dr. Andrew C. Pineda, HPCERC/AHPCC
Dr. Brian Smith, HPCERC/AHPCC
The University of New Mexico
November 17, 1997
Last Revised: September 18, 1998

MPI (Message Passing Interface)

MPI (Message Passing Interface) is a library of function calls (subroutine calls in Fortran) that allow the coordination of a program running as multiple processes in a distributed memory environment. These function calls can be added to a serial program in order to convert it to a parallel program, often with only a few modifications. MPI programs may be compiled run in parallel on multiple nodes of a massively parallel computer such as an IBM SP2 or on a cluster (homogeneous or heterogeneous) of workstations connected over a network. MPI was written with the goal of creating a widely used language- and platform-independent standard for the writing of message-passing programs.

The chief advantage of MPI is that it allows a more flexible division of work among processes than does a data parallel programming language such as High Performance Fortran (HPF). This flexibility of MPI allows the user to select or develop his or her own parallel programming paradigm or approach such as master/slave, or single-program multiple data. The price of that advantage is that the user becomes responsible for setting up the communications among the processes and ensuring that each process has enough work to do between communications calls. Converting a serial code to a parallel MPI code involves 4 basic steps:

- inserting the MPI include file and additional declarations for auxiliary variables in the specification sections of the parts of the program making MPI calls;
- adding the library calls to initialize the MPI communications environment;
- doing the calculation and performing message passing calls; and
- adding the library calls to terminate the MPI environment.

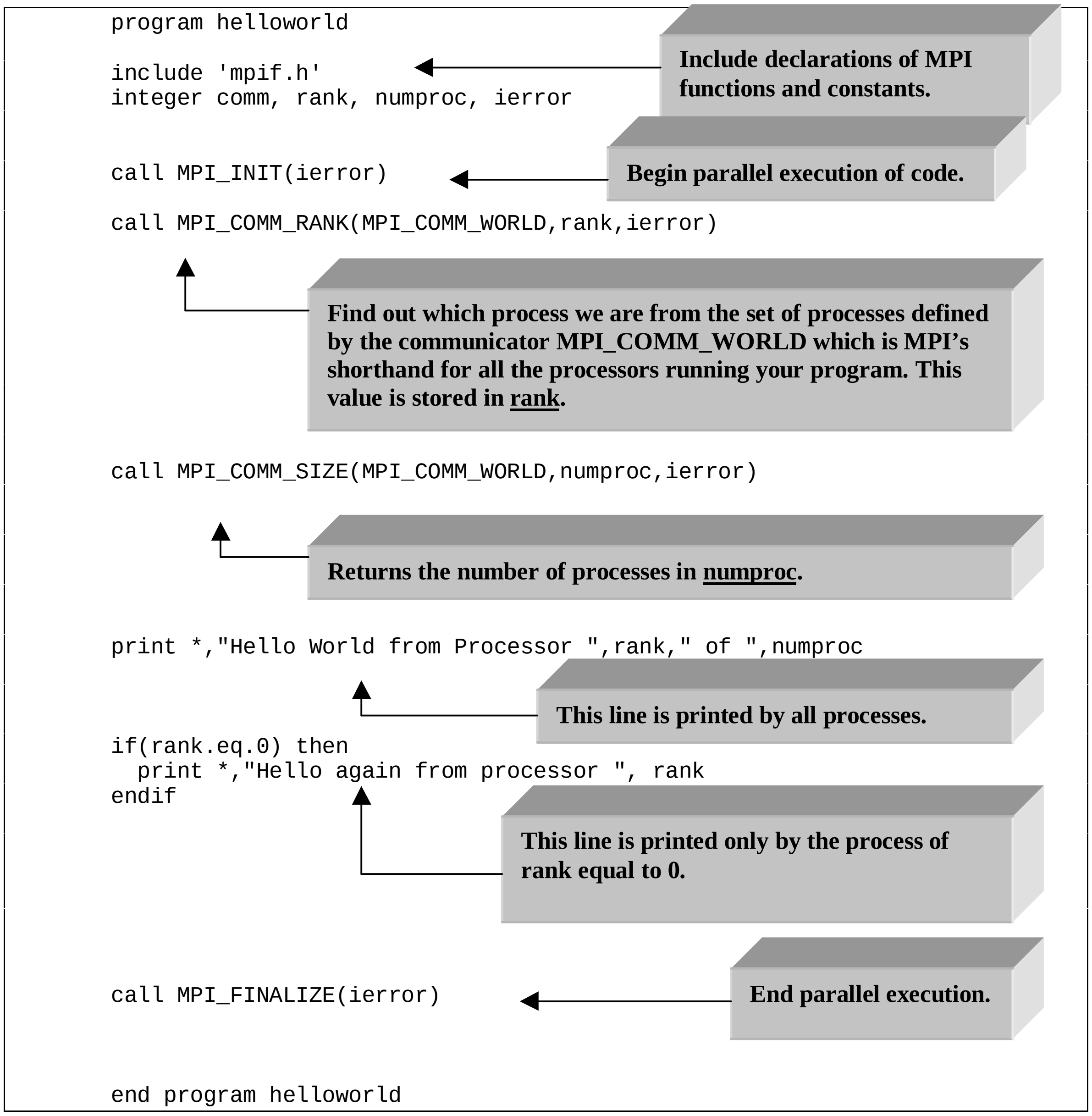
The MPI library can be called from Fortran 77, Fortran 90, C, and C++ programs although function bindings exist only for Fortran 77 and C.[*] That is, the programmer is calling a Fortran 77 or C library when using Fortran 90 or C++. This leads to some serious problems for Fortran 90, because the MPI library lacks the bindings necessary for correctly passing objects (typically arrays) of more than one type to a particular MPI function. This is an issue having to do with the strong type checking of arguments to function calls required by Fortran 90 (and provided via interface blocks) that is not handled consistently by Fortran 90 compilers. For example, the IBM *xlf90* (*mpxlf90)* compiler basically allows the programmer to sidestep this problem with a compiler switch, but the NAG F90 compiler does not. C++ provides an explicit method for calling a C function from a C++ program via the extern "C" declaration (which have been inserted in the MPI header files via an **#ifdef**), so there is in principle no problem with using the C MPI library in C++.[†]

[*] In this tutorial, we document the syntax of MPI calls in Fortran. There are only two differences between the C and Fortran syntax having to do with the spelling (case) of the MPI call and with how error values are returned. In C, the MPI calls are functions named MPI_Abcdef which return an integer valued error value. In Fortran, the MPI functions are subroutines named MPI_ABCDEF which return the integer valued error as an additional argument which appears last in the argument list.
[†] For those of you not acquainted with the minor differences between C and C++, there is a difference in the order in which C and C++ compilers push function arguments onto the program stack. In C, function arguments are pushed onto the stack proceeding from left to right through the argument list during a call. In C++, they are pushed onto the stack in right to left order. In order to be able to call modules compiled in C, C++ provides a mechanism for reversing this order via the extern "C" declaration.

Our intent in this tutorial is to teach MPI by example, so we will examine several MPI programs that illustrate the use of the MPI subroutines. The entire MPI library consists of over one hundred MPI calls and therefore we do not provide a complete description of the use of all MPI calls. However, a great deal of programming in MPI can be done with less than two dozen calls. Hence, we will focus our attention on the most useful MPI calls and refer the reader to the MPI reference, “MPI: The Complete Reference”, for the more advanced calls.

A Basic MPI Program

As is frequently done when studying a new programming language, we begin our study of MPI with a parallel version of the ubiquitous “Hello, World” program. This example illustrates the basic structure of an MPI program without any communications between processes. It is shown in **Figure 1**.



```
program helloworld

include 'mpif.h'
integer comm, rank, numproc, ierror

call MPI_INIT(ierror)

call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierror)

call MPI_COMM_SIZE(MPI_COMM_WORLD,numproc,ierror)

print *,"Hello World from Processor ",rank," of ",numproc

if(rank.eq.0) then
  print *,"Hello again from processor ", rank
endif

call MPI_FINALIZE(ierror)

end program helloworld
```

Include declarations of MPI functions and constants.

Begin parallel execution of code.

Find out which process we are from the set of processes defined by the communicator MPI_COMM_WORLD which is MPI’s shorthand for all the processors running your program. This value is stored in <u>rank</u>.

Returns the number of processes in <u>numproc</u>.

This line is printed by all processes.

This line is printed only by the process of rank equal to 0.

End parallel execution.

Figure 1. A parallel version of Hello, World.

In the declaration section, the MPI include file is inserted and additional MPI variables are declared. In Fortran, the included file is called "mpif.h". In Fortran 90, this file is called "mpif90.h". In C and C++, the include files are called "mpi.h" and "mpi++.h", respectively. In the executable section of the code, communication between processes is started by the MPI_INIT function. Next the processes must obtain information about their identity and the number of processes so that they can communicate and allocate work. Processes determine their identity via the MPI_COMM_RANK call. The first argument to MPI_COMM_RANK is the MPI communicator MPI_COMM_WORLD. In MPI, communicators are used to specify the processes constituting a communications group. The MPI_COMM_WORLD communicator is provided by MPI as a way to refer to all of the processes. MPI also provides functions for creating your own communicators for subgroups of the processes. This allows you to create a processor topology that maps onto the problem you are trying to solve. In the example above, the identity of the process is returned in the **rank** variable. This **rank** is a zero-based integer value. The number of processes in a communications group is found using the MPI_COMM_SIZE call.

After the initialization calls are complete, the processes all begin work on their own computations. In the example in Figure 1, each process executes the first print statement in which they print their rank and the number of processes. They then test to see if they are the process with rank equal to zero, and if so execute the second print statement. Finally, with their work completed each process calls MPI_FINALIZE to close down their communications with the other processes and then stops execution.

Exercise 1

Compile and run the "Hello, World" example code above on 3 or more processors. The source code may be found in the file ***hellompi.f.*** *Directions for compiling and running the program under MPI have been provided in "Appendix A. Compiling and running a parallel program." Does it do what you expect? Did anything unusual or surprising happen?*

As you might have noticed, the above example, and indeed all of the examples in this tutorial, follows the single-program multiple-data paradigm. That is, one program is being written and compiled, but the executable for it is loaded and run on all the processors being used. Each processor is running this program asynchronously and therefore executes the statements in the same order but at different times. IF statements such as *if (rank==0)* in **hellompi.f** are executed by each processor, but only the processor that has rank (id) equal to 0 executes the body of the IF construct, in this case, the PRINT statement. Note that *all* processors execute the MPI_COMM_RANK subroutine, but each receives a *different* value of **rank.**

An MPI Program with Point-to-Point Communications

As a next step in building a useful program, we add MPI communications calls to handle communications between processes. As all communications between processes are ultimately built out of communications calls that operate between pairs of processes, i.e., the point-to-point calls, we discuss them first. Point-to-point communications calls consist of two basic operations, sends and receives, which each come in a number of flavors depending upon the amount of control the user wants or needs to have over how the communication operation is performed. The generic versions of these calls are MPI_SEND and MPI_RECV, which respectively send and receive messages between pairs of processes. The calling syntax for these two functions is described in detail below in "Appendix B. Common MPI Library Calls". Other versions of these calls, such as MPI_ISEND, MPI_BSEND to name but a few, are described in the MPI reference.

The most important consideration for the MPI programmer is whether an MPI communications call is **blocking** or **non-blocking**. In a blocking call, a process that is communicating data must wait until the message data has been safely stored away so that the sender can access and overwrite the send buffer. Until then, a blocking call cannot return and as a result the process making the blocking call cannot do any more useful work. In MPI, the basic send and receive operations, MPI_SEND and MPI_RECV, are both blocking calls. However, blocking calls may be implemented as buffered or unbuffered operations. In a buffered operation, the data being sent is copied to an intermediate buffer after which the sending process is free to continue. The particular MPI implementation is free to implement the blocking send as a buffered or an unbuffered operation. The consequence of this is that the blocking send may behave like a non-blocking send operation depending upon the implementation. Variants of MPI_SEND exist to allow the user to force buffered operation, etc.

The program in **Figure 2** below illustrates the use of the basic MPI send and receive calls. In this simple program, process 0 sends a message, consisting of a character string, to process 1, which then appends information to it and sends it back to process 1.   The MPI_SEND call takes as arguments a buffer containing the data, an integer value for the size of the buffer, and an integer value describing the type of the data being sent. The MPI include file contains pre-defined values for the standard data types in Fortran and C. In this example, a character string (array) is being sent, so the MPI type in Fortran is MPI_CHARACTER. In "Appendix B. Common MPI Library Calls", the remaining pre-defined types in Fortran are listed. The pre-defined types for C may be found in the MPI reference. Later in this tutorial, we briefly discuss how one can send messages containing arbitrary types. In addition to these arguments, MPI_SEND takes as arguments the rank of the destination process, a message tag to label the messages, and the communicator for the process group involved. The matching receive call, MPI_RECV, takes similar arguments.

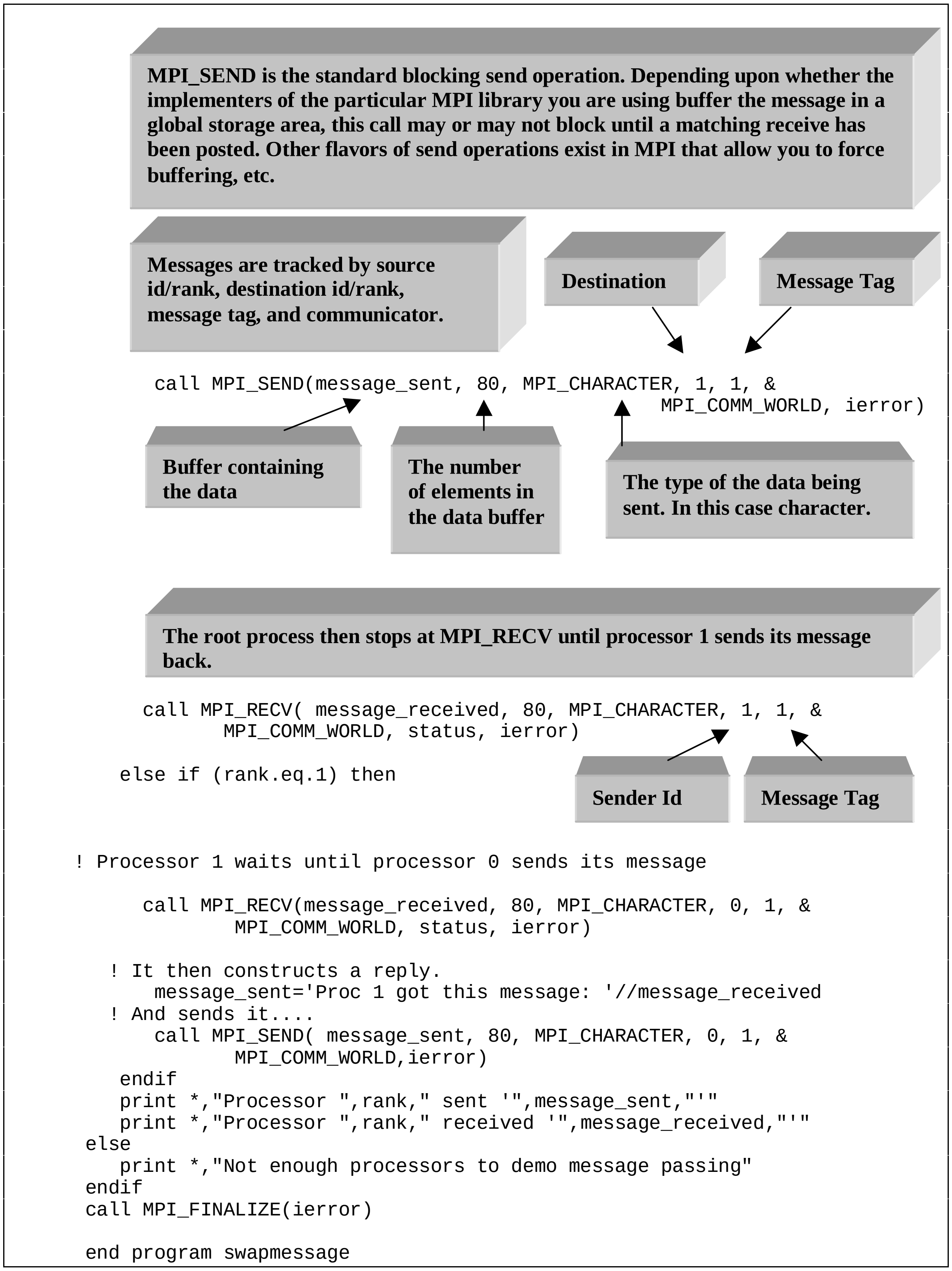
```
      program swapmessage
      include 'mpif.h'
      integer comm, rank, numproc, ierror, root
      integer status(MPI_STATUS_SIZE)
      character(80) message_sent, message_received
      ! Setup default messages.
      message_sent='No message sent'
      message_received='No message received'
      root=0
      ! Start up MPI environment
      call MPI_INIT(ierror)
      call MPI_COMM_RANK(MPI_COMM_WORLD,rank,ierror)
      call MPI_COMM_SIZE(MPI_COMM_WORLD,numproc,ierror)

      if(numproc.gt.1) then

       ! Swap messages only if we have more than 1 processor.  The root
       ! process sends a message to processor 1 and then waits for a
       ! reply.  Processor 1 waits for a message from the root process,
       ! adds to it, and then sends it back.

         if(rank.eq.root) then

            message_sent='Hello from processor 0'
```



MPI_SEND is the standard blocking send operation. Depending upon whether the implementers of the particular MPI library you are using buffer the message in a global storage area, this call may or may not block until a matching receive has been posted. Other flavors of send operations exist in MPI that allow you to force buffering, etc.

Messages are tracked by source id/rank, destination id/rank, message tag, and communicator.

```
       call MPI_SEND(message_sent, 80, MPI_CHARACTER, 1, 1, &
                                            MPI_COMM_WORLD, ierror)
```

Buffer containing the data
The number of elements in the data buffer

The type of the data being sent. In this case character.

The root process then stops at MPI_RECV until processor 1 sends its message back.

```
      call MPI_RECV( message_received, 80, MPI_CHARACTER, 1, 1, &
             MPI_COMM_WORLD, status, ierror)

    else if (rank.eq.1) then
```

```
! Processor 1 waits until processor 0 sends its message

      call MPI_RECV(message_received, 80, MPI_CHARACTER, 0, 1, &
              MPI_COMM_WORLD, status, ierror)

   ! It then constructs a reply.
       message_sent='Proc 1 got this message: '//message_received
   ! And sends it....
       call MPI_SEND( message_sent, 80, MPI_CHARACTER, 0, 1, &
              MPI_COMM_WORLD,ierror)
    endif
    print *,"Processor ",rank," sent '",message_sent,"'"
    print *,"Processor ",rank," received '",message_received,"'"
 else
    print *,"Not enough processors to demo message passing"
 endif
 call MPI_FINALIZE(ierror)

 end program swapmessage
```

Figure 2. A simple exchange of messages.

Exercise 2

*Compile and run the program **swapmsg.f**, which contains the program listed in **Figure 2** to run on two or more processes. What messages are printed by the processes? Add a third process to the communication.*

Exercise 3

With blocking calls, it is possible to arrange the calls in such a way that a pair of processes attempting to communicate with each other will deadlock. Can you construct a simple exchange of messages between 2 processes, using MPI_SEND and MPI_RECV calls, in which the processors are guaranteed to deadlock? (Don't try to run this code…)

In this tutorial, we primarily discuss **blocking** communications for simplicity, however non-blocking calls are occasionally needed to avoid situations in which processes can deadlock. The **non-blocking** call most frequently used to resolve such situations is MPI_IRECV which is the **non-blocking** form of the MPI receive call MPI_RECV. Additionally, MPI_IRECV can be used to overlap time spent on computation with time spent on communications, which can frequently result in dramatic improvements in processing speed. The process calling MPI_IRECV basically tells the other processes that it is expecting a message containing some data from another process, and then returns control of execution to the calling process. The calling process is then free to do useful work, provided that it does not touch the buffer that will contain the received message until the receive operation has been completed. This is done at a later time with the blocking MPI_WAIT function or the non-blocking MPI_TEST function. MPI_TEST checks to see if a message has arrived and either receives the message and sets a logical flag to true indicating that the communication is complete, or sets the logical flag to false and returns. An MPI_IRECV immediately followed by an MPI_WAIT is equivalent to an MPI_RECV call. As illustrated below in Figure 3, MPI_IRECV can be used in place of MPI_RECV.

```
      if(rank.eq.root) then

         message_sent='Hello from processor 0'
```

Begin the receive operation by letting the world know we are expecting a message from process 1. We then return immediately.

```
         call MPI_IRECV( message_received, 80, MPI_CHARACTER, 1, 1, &
               MPI_COMM_WORLD, request, ierror)
```

Now send the message as before.

```
         call MPI_SEND(message_sent, 80, MPI_CHARACTER, 1, 1, &
              MPI_COMM_WORLD, ierror)
```

Now wait for the receive operation to complete.

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


      else if (rank.eq.1) then
```

Figure 3. A replacement for the rank.eq.root code in swapmsg.f that uses MPI_IRECV instead of MPI_RECV.

Exercise 4

Fix your deadlocking code from the previous exercise using MPI_IRECV. Compile and run it to see that it works.

MPI Programs with Collective Communications

MPI collective communications calls provide mechanisms for the passing of data from one process to all, all processes to one, or from all processes to all processes in an efficient manner. These calls are used to distribute arrays or other data across multiple processes for parallel computation. Calls for all to all communications are not discussed here for the sake of simplicity and brevity.

Collective function calls may (depending upon the MPI implementation) return control to their calling process as soon as their participation in the collective communication is complete. At this point, the calling process is free to access data in the communications buffer. The four most commonly used types of collective calls are broadcast, scatter, gather, and reduction operations. In a broadcast operation, a block of data is distributed from one process to all the processes in a communication group. In a scatter operation, a block of data is broken up into pieces and the pieces are distributed to all the processes in the communications group. A gather operation is the inverse of the scatter operation; it collects the pieces of data distributed across a group of processes and assembles them as a single block of data on a single processor.

The next example program, listed in Figure 4, illustrates the use of the MPI_BCAST, MPI_SCATTER, and MPI_GATHER routines. The program computes a linear combination (sum) of two vectors of length 80 by distributing the pieces of the vectors across 8 processors in blocks of length 10 using the MPI_SCATTER function. The coefficients multiplying each vector are distributed to all processes using the MPI_BCAST function. Each processor then forms the vector sum on components of the vector from the block assigned to it and returns the completed sum to the root process via the MPI_GATHER function, which reassembles the components of the vector in their proper order. In this case, the data was evenly divisible among the processors. If this were not the case, we would use the more general functions MPI_SCATTERV and MPI_GATHERV to distribute the data. See “Appendix B. Common MPI Library Calls” for details on these calls.

```
program  vecsum

include 'mpif.h'

integer, parameter ::  dim1 = 80, dim2 = 10
integer  ierr, rank, size, root
integer  sec_start, nano_start
integer  sec_curr, nano_curr
integer  sec_startup, nano_startup
integer  sec_comp, nano_comp
integer  sec_cleanup, nano_cleanup

real, dimension(dim1) :: x, y, z
real, dimension(dim2) :: xpart, ypart, zpart
real, dimension(2) ::  coeff

interface
  subroutine posix_timer(job_sec, job_nanosec)
    integer job_sec, job_nanosec
  end subroutine
end interface

root = 0
call  MPI_INIT( ierr )
call  MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
call  MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )

print *, 'START process on processor ', rank

if( rank == root )  then
  call posix_timer(sec_start, nano_start)
  coeff = (/ 1.0, 2.0 /)
  x = 2.0
  y = 3.0
endif
```

This program computes

$$\boxed{\vec{z} = a\vec{x} + b\vec{y}}$$

on 8 processes using MPI calls.

Do timing in the root process only.

MPI_SCATTER distributes blocks of array <u>x</u> from the <u>root</u> process to the array <u>xpart</u> belonging to each process in MPI_COMM_WORLD. Likewise, blocks of the array <u>y</u> are distributed to the array <u>ypart</u>.

Array <u>x</u> and the number of elements of type real to send to each process. Only meaningful to <u>root</u>.

Array <u>xpart</u> and the number of elements of type real to receive.

```
call  MPI_SCATTER( x, dim2, MPI_REAL, xpart, dim2, MPI_REAL, root, &
                   MPI_COMM_WORLD, ierr )
```

Array y and the number of elements of type real to send to each process. Only meaningful to root.

Array ypart and the number of elements of type real to receive.

```
 call  MPI_SCATTER( y, dim2, MPI_REAL, ypart, dim2, MPI_REAL, root, &
              MPI_COMM_WORLD, ierr )
```

The coefficients, *a* and *b*, are stored in an array of length 2, coeff, that is broadcast to all processes via MPI_BCAST from the process root.

```
 call  MPI_BCAST( coeff, 2, MPI_REAL, root, MPI_COMM_WORLD, ierr )

 if( rank == root )  then
   call posix_timer(sec_curr, nano_curr)
   sec_startup = sec_curr - sec_start
   nano_startup = nano_curr - nano_start
   sec_start = sec_curr
   nano_start = nano_curr
 endif
```

Now each processor computes the vector sum on its portion of the vector. The blocks of the vector sum are stored in zpart.

```
do i = 1, dim2
   zpart(i) = coeff(1)*xpart(i) + coeff(2)*ypart(i)
 enddo

 if( rank == root )  then
   call posix_timer(sec_curr, nano_curr)
   sec_comp = sec_curr - sec_start
   nano_comp = nano_curr - nano_start
   sec_start = sec_curr
   nano_start = nano_curr
 endif
```

Now we use MPI_GATHER to collect the blocks back to the root process.

The array zpart to be gathered and the number of elements each process sends to root.

For the root process, the array z contains the collected blocks from all processes on output. MPI_GATHER needs to know how much data to collect from each process.

```
      call  MPI_GATHER( zpart, dim2, MPI_REAL, z, dim2, MPI_REAL, root,
     &                   MPI_COMM_WORLD, ierr )

      if( rank == root )  then
        call posix_timer(sec_curr, nano_curr)
        sec_cleanup = sec_curr - sec_start
        nano_cleanup = nano_curr - nano_start
      endif

      print *, 'Finish processor ', rank

      if( rank == root ) then
          print *, 'Vector sum, elements 10 and 60, are: ', z(10), z(60)
          print *, 'Startup execution times (sec, nano): ', &
     &              sec_startup, nano_startup
          print *, 'Computation execution times (sec, nano): ', &
     &             sec_comp, nano_comp
          print *, 'Cleanup execution times (sec, nano): ', &
     &             sec_cleanup, nano_cleanup
      endif

      call  MPI_FINALIZE( ierr )

      end
```

Figure 4. An example performing a vector sum operation, $\vec{z} = a\vec{x} + b\vec{y}$, using MPI.

Exercise 5

Compile and run the above program, ***vecsummpi.f****, for various sizes of the arrays as well as single and multiple processes. What is the observed speedup as a function of the number of processes? When does it pay to run the program in parallel?*

The next example program, **Figure 5**, illustrates the use of the MPI_REDUCE call. In the example, the dot product of two large vectors is computed. As in the previous example, the vectors are distributed blockwise across 8 processors using MPI_SCATTER. Each processor then computes the dot product on the piece that it has. Finally, the pieces of the dot product must be summed together to form the full dot product on the root process. This kind of operation is called a reduction operation and is performed with the MPI_REDUCE call. The predefined constant MPI_SUM tells MPI_REDUCE that the desired reduction operation is to sum the values it has been given and return the sum to the process collecting the results. Other types of reduction operations include taking the product, determining maximum or minimum values, etc.

```
      program  vecprod

      integer, parameter ::  dim1 = 80, dim2 = 10

      include 'mpif.h'

      integer  ierr, rank, size, root

      real, dimension(dim1) :: x, y
      real, dimension(dim2) :: xpart, ypart
      real  z, zpart
      integer  sec_start, nano_start
      integer  sec_curr, nano_curr
      integer  sec_startup, nano_startup
      integer  sec_comp, nano_comp
      integer  sec_cleanup, nano_cleanup


      interface
        subroutine posix_timer(job_sec, job_nanosec)
          integer job_sec, job_nanosec
        end subroutine
      end interface

      root = 0
      call  MPI_INIT( ierr )
      call  MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
      call  MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )

      print *, 'START process on processor ', rank

      if( rank == root )  then
        call posix_timer(sec_start, nano_start)
        x = 1.0
        y = 2.0
      endif

      call  MPI_SCATTER( x, dim2, MPI_REAL, xpart, dim2, MPI_REAL, root, &
                        MPI_COMM_WORLD, ierr )
      call  MPI_SCATTER( y, dim2, MPI_REAL, ypart, dim2, MPI_REAL, root, &
                        MPI_COMM_WORLD, ierr )

      if( rank == root )  then
        call posix_timer(sec_curr, nano_curr)
        sec_startup = sec_curr - sec_start
        nano_startup = nano_curr - nano_start
        sec_start = sec_curr
        nano_start = nano_curr
      endif

      zpart = 0.0
      do i = 1, dim2
        zpart = zpart + xpart(i)*ypart(i)
      enddo

      if( rank == root )  then
        call posix_timer(sec_curr, nano_curr)
        sec_comp = sec_curr - sec_start
        nano_comp = nano_curr - nano_start
        sec_start = sec_curr
        nano_start = nano_curr
      endif
```

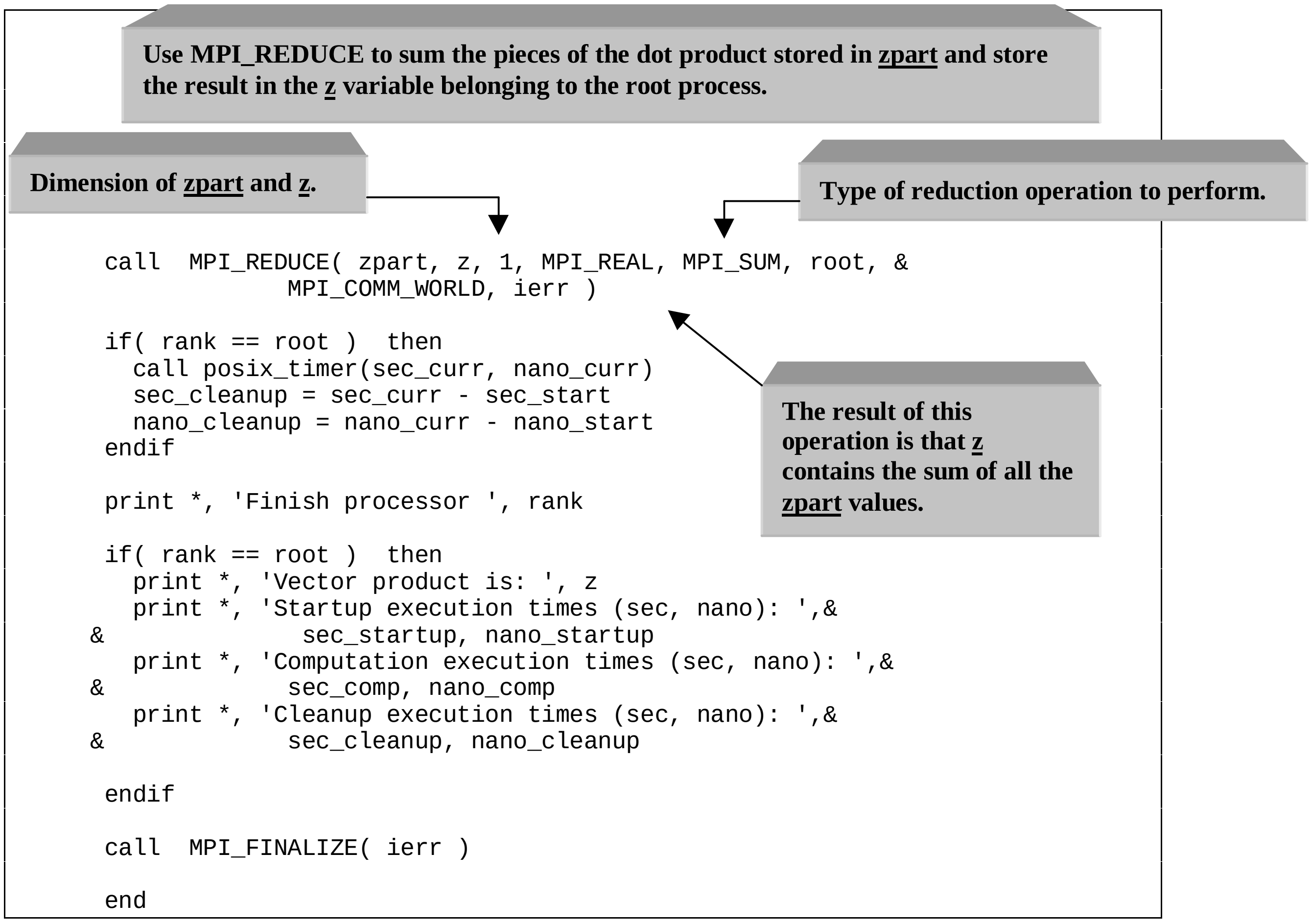
This program computes the dot product of two vectors,

$$z = \vec{x} \cdot \vec{y}$$

using MPI calls.

Distribute the arrays x and y as in the previous example.

Each process then computes the dot product of the pieces of the array to which it has access.



Use MPI_REDUCE to sum the pieces of the dot product stored in zpart and store the result in the z variable belonging to the root process.

Dimension of zpart and z.

Type of reduction operation to perform.

The result of this operation is that z contains the sum of all the zpart values.

```
      call  MPI_REDUCE( zpart, z, 1, MPI_REAL, MPI_SUM, root, &
                 MPI_COMM_WORLD, ierr )

      if( rank == root )  then
        call posix_timer(sec_curr, nano_curr)
        sec_cleanup = sec_curr - sec_start
        nano_cleanup = nano_curr - nano_start
      endif

      print *, 'Finish processor ', rank

      if( rank == root )  then
        print *, 'Vector product is: ', z
        print *, 'Startup execution times (sec, nano): ',&
     &              sec_startup, nano_startup
        print *, 'Computation execution times (sec, nano): ',&
     &             sec_comp, nano_comp
        print *, 'Cleanup execution times (sec, nano): ',&
     &             sec_cleanup, nano_cleanup

      endif

      call  MPI_FINALIZE( ierr )

      end
```

Figure 5. An example illustrating the computation of a dot product of 2 vectors using MPI.

Exercise 6

*Repeat Exercise 5 for this program, **vecprodmpi.f**.*

As our final example, in Figure 6, we illustrate the use of MPI calls in forming the product of a matrix with a vector to form another vector. In this example the vector being multiplied is distributed across the processors as blocks of rows, and the matrix is distributed across the processors as blocks of columns. This allows each processor to compute a column vector using the column-oriented multiplication algorithm. The column vectors computed by each processor are then added together in an MPI_REDUCE operation to form the final result vector.

```
      program  matvec2

      ! Perform matrix vector product -- Y = AX
      ! This is method two -- distribute A by block columns
      ! and X in blocks (of rows) and the partial vector sum of Y is on
      ! each processor.

      include 'mpif.h'

      integer, parameter ::  dim1 = 80, dim2 = 10, dim3 = dim1*dim2
      integer  ierr, rank, size, root, i, j
      integer  sec_start, nano_start
      integer  sec_curr, nano_curr
      integer  sec_startup, nano_startup
      integer  sec_comp, nano_comp
      integer  sec_cleanup, nano_cleanup

      real, dimension(dim1,dim1) :: a
      real, dimension(dim1,dim2) :: apart
      real, dimension(dim1) :: x, y, ypart
      real, dimension(dim2) :: xpart

      interface
        subroutine posix_timer(job_sec, job_nanosec)
          integer job_sec, job_nanosec
        end subroutine
      end interface

      root = 0
      call  MPI_INIT( ierr )
      call  MPI_COMM_RANK( MPI_COMM_WORLD, rank, ierr )
      call  MPI_COMM_SIZE( MPI_COMM_WORLD, size, ierr )

      print *, 'START process on processor ', rank

      if( rank == root )  then
        call posix_timer(sec_start, nano_start)
        do i = 1, dim1
          x(i) = 1.0
          do j = 1, dim1
            a(j,i) = i + j
          enddo
        enddo
      endif

      call  MPI_SCATTER( a, dim3, MPI_REAL, apart, dim3, MPI_REAL, root,&
     &                   MPI_COMM_WORLD, ierr )
      call  MPI_SCATTER( x, dim2, MPI_REAL, xpart, dim2, MPI_REAL, root,&
     &                   MPI_COMM_WORLD, ierr )

      if( rank == root )  then
        call posix_timer(sec_curr, nano_curr)
        sec_startup = sec_curr - sec_start
        nano_startup = nano_curr - nano_start
        sec_start = sec_curr
        nano_start = nano_curr
      endif
```

Distribute the 80x80 array A by columns as 80x10 blocks stored in APART.

Distribute the 80 dimensional array x in blocks of length 10.

```
      do j = 1, dim1
        ypart(j) = 0.0
      enddo
      do i = 1, dim2
        do j = 1, dim1
          ypart(j) = ypart(j) + xpart(i)*apart(j,i)
        enddo
      enddo

      if( rank == root )  then
        call posix_timer(sec_curr, nano_curr)
        sec_comp = sec_curr - sec_start
        nano_comp = nano_curr - nano_start
        sec_start = sec_curr
        nano_start = nano_curr
      endif

      call  MPI_REDUCE( ypart, y, dim1, MPI_REAL, MPI_SUM, root, &
                        MPI_COMM_WORLD, ierr )

      if( rank == root )  then
        call posix_timer(sec_curr, nano_curr)
        sec_cleanup = sec_curr - sec_start
        nano_cleanup = nano_curr - nano_start
      endif

      print *, 'Finish processor ', rank

      if( rank == root ) then
          print *, 'Matrix vector product, elements 10 and 60, are: ',&
     &             y(10), y(60)
          print *, 'Startup execution times (sec, nano): ',&
     &              sec_startup, nano_startup
          print *, 'Computation execution times (sec, nano): ',&
     &             sec_comp, nano_comp
          print *, 'Cleanup execution times (sec, nano): ',&
     &             sec_cleanup, nano_cleanup
      endif

      call  MPI_FINALIZE( ierr )

      end
```

Each processor computes part of the product using a column-oriented algorithm.

Compute the final y as a vector sum of the pieces ypart using MPI_REDUCE.

Figure 6. Matrix vector multiplication example.

Exercise 7

Compile and run the above program, ***matvec2mpi.f****. Turn the above program into a matrix-matrix multiplication program.*

Exchanging messages of arbitrary types

In addition to the pre-defined types given above, MPI supports the construction of user-defined types. These user-defined types play two roles in practice. One is to allow the passing between processes of data structures (e.g. user-defined types in Fortran 90) containing an arbitrary combination of the pre-defined types. The other is to allow the user to reorganize data in an arbitrary way during the message passing process so that it can be efficiently transmitted to and used by the receiving process. For example, a matrix being sent from one process to another may be transposed on the fly (as might be required in order to perform a 2-D FFT). Or the data to be transmitted is not in contiguous memory, as is the case when a sub-matrix of a larger matrix is being exchanged between processes.

The usage of the function to pass arbitrary structures is very complicated, so the user is referred to the MPI reference, “MPI: The Complete Reference” for a discussion of this topic.

As an illustration of the use of user-defined types, we present MPI code that transposes a matrix on the fly. To do this, we create a new data type that describes the matrix layout in row-major order, send the matrix with this data type, and then receive the matrix in normal, column-major order.

```
REAL a(100,100), b(100,100)
INTEGER row, xpose, sizeofreal, myrank, ierr
INTEGER status(MPI_STATUS_SIZE)
.
.
.
! transpose matrix a into b.

CALL MPI_COMM_RANK(MPI_COMM_WORLD, myrank, ierr)
```

The MPI_TYPE_EXTENT function is used to obtain MPI’s internal value for the size of the real data type. This is needed by the MPI_TYPE_HVECTOR call.

```
CALL MPI_TYPE_EXTENT(MPI_REAL, sizeofreal, ierr)
```

MPI_TYPE_VECTOR is used to create a data type, row, for the new row which is a vector with 100 real entries and a stride of 100 in the original array.

```
CALL MPI_TYPE_VECTOR(100, 1, 100, MPI_REAL, row, ierr)
```

MPI_TYPE_HVECTOR is used to create a data type describing the matrix in row-major order. This is done by interleaving copies of the row data type created previously. The MPI_TYPE_HVECTOR call allows the user to specify alignment of data in units of bytes. This is used to generate the new xpose data type from 100 copies of the row data type with each copy offset by sizeofreal bytes from the previous copy.

```
CALL MPI_TYPE_HVECTOR(100, 1, sizeofreal, row, xpose, ierr)
```

MPI_TYPE_COMMIT lets MPI know about the new data type.

```
CALL MPI_TYPE_COMMIT(xpose, ierr)
```

Now we transpose the matrix by sending it in row-major order and telling the receiving process that the data is in normal, column-major order.

```
CALL MPI_SENDRECV(a, 1, xpose, myrank, 0, b, 100*100,MPI_REAL,myrank,0,
                  MPI_COMM_WORLD, status, ier)
.
.
.
```

Figure 7 . Using MPI user-defined datatypes to transpose a matrix "on the fly".

MPI Programming Exercises

With the aid of your instructor, select from the exercises below.

Molecular Dynamics

The MD exercises below ask you to convert the Euler and Verlet MD codes for the 1-D chain of atoms that you worked on earlier in the semester to MPI. The basic approach you will use is to assign blocks of atoms to processors. At each time-step, you will find that neighboring blocks of atoms will need to exchange information about the positions of the atoms at the adjoining ends so that the forces on the particles at the ends of the blocks can be computed. This problem is common to both the Euler and Verlet codes, so there is no need to do both exercises. Simply pick the one of most interest to you.

Exercise 8

Rewrite the serial Euler code, **euler1_h1.f**, as an MPI code. Vary the number of atoms per processor and the number of processors. How does the time to run your code scale with each?

Exercise 9

Rewrite the serial Verlet code, **verlet1_h1.f**, as an MPI code. Vary the number of atoms per processor and the number of processors. How does the time to run your code scale with each?

Gaussian Elimination.

Exercise 10

The Gaussian elimination code below is extracted from the Fortran 90 tutorial. Convert it to MPI by block distributing the data by columns. Note that this is not the most efficient way to convert the code since with each iteration fewer and fewer processors have anything to do. A better way, albeit more difficult to implement, is to distribute the array in a cyclic fashion. This has the virtue of keeping the processors busy for a longer period of time, but eventually one has the same problem.

```
module GaussianSolver
  implicit none

  ! The default value for the smallest pivot that will be accepted
  ! using the GaussianSolver subroutines. Pivots smaller than this
  ! threshold will cause premature termination of the linear equation
  ! solver and return false as the return value of the function.

  real, parameter :: DEFAULT_SMALLEST_PIVOT = 1.0e-6

contains

  ! Use Gaussian elimination to calculate the solution to the linear
  ! system, A x = b. No partial pivoting is done. If the threshold
  ! argument is present, it is used as the smallest allowable pivot
  ! encountered in the computation; otherwise, DEFAULT_SMALLEST_PIVOT,
  ! defined in this module, is used as the default threshold. The status
  ! of the computation is a logical returned by the function indicating
  ! the existence of a unique solution (.true.), or the nonexistence of
  ! a unique solution or threshold passed (.false.).
  ! Note that this is an inappropriate method for some linear systems.
  ! In particular, the linear system, M x = b, where M = 10e-12 I, will
  ! cause this routine to fail due to the presence of small pivots.
  ! However, this system is perfectly conditioned, with solution x = b.

  function gaussianElimination( A, b, x, threshold )
    implicit none
    logical gaussianElimination
    real, dimension( :, : ), intent( in ) :: A ! Assume the shape of A.
    real, dimension( : ), intent( in ) :: b    ! Assume the shape of b.
    real, dimension( : ), intent( out ) :: x   ! Assume the shape of x.

    ! The optional attribute specifies that the indicated argument 40
    ! is not required to be present in a call to the function. The
    ! presence of optional arguments, such as threshold, may be checked
    ! using the intrinsic logical function, present (see below).

    real, optional, intent( in ) :: threshold
    integer i, j                              ! Local index variables.
    integer N                                 ! Order of the linear system.
    real m                                    ! Multiplier.
    real :: smallestPivot = DEFAULT_SMALLEST_PIVOT

    ! Pointers to the appropriate rows of the matrix during the elmination.

    real, dimension( : ), pointer :: pivotRow
    real, dimension( : ), pointer :: currentRow

    ! Copies of the input arguments. These copies are modified during
    ! the computation.  The target attribute is used to indicate that
    ! the specified variable may be the target of a pointer. Rows of
    ! ACopy are targets of pivotRow and currentRow, defined above.

    real, dimension( size( A, 1 ), size( A, 2) ), target :: ACopy
    real, dimension( size( b ) ) :: bCopy

    !
    ! Status of the computation. The return value of the function.
    !

    logical successful
    !
    ! Change the smallestPivot if the threshold argument was included.
    !

    if ( present( threshold ) ) smallestPivot = abs( threshold )

    !
    ! Setup the order of the system by using the intrinsic function size.
    ! size returns the number of elements in the specified dimension of
    ! an array or the total number of elements if the dimension is not
    ! specified. Also assume that a unique solution exists initially.
    !

    N = size( b )
    ACopy = A
    bCopy = b
    successful = .true.

    !
    ! Begin the Gaussian elimination algorithm.  Note the use of array
    ! sections in the following loops. These eliminate the need for
    ! many do loops that are common in Fortran 77 code.  Pointers are
    ! also used below and enhance the readability of the elimination
    ! process.  Begin with the first row.
    !

    i = 1

    ! Reduce the system to upper triangular.

    do while ( ( successful ) .and. ( i < N ) )

       !
       ! The following statement is called pointer assignment and uses
       ! the pointer assignment operator '=>'. This causes pivotRow
       ! to be an alias for the ith row of ACopy. Note that this does
       ! not cause any movement of data.
       ! Assign the pivot row.
       !

       pivotRow => ACopy( i, : )

       !
       ! Verify that the current pivot is not smaller than smallestPivot.
       !

       successful = abs( pivotRow( i ) ) >= smallestPivot
       if ( successful ) then

          !
          ! Eliminate the entries in the pivot column below the pivot row.
          !

          do j = i+1, N

             ! Assign the current row.

             currentRow => ACopy( j, : )

             ! Calculate the multiplier.

             m = currentRow( i ) / pivotRow( i )
```

```
              ! Perform the elimination step on currentRow and right
              ! hand side, bCopy.

              currentRow = currentRow  - m * pivotRow
              bCopy( j ) = bCopy( j )  - m * bCopy( i )

           enddo
        endif

        ! Move to the next row.

        i = i + 1
     end do

     ! Check the last pivot.

     pivotRow => ACopy( N, : )
     if ( successful ) successful = abs( pivotRow( N ) ) >= smallestPivot
     if ( successful ) then
        do i = N, 2, -1 ! Backward substitution.

           ! Determine the ith unknown, x( i ).

           x( i ) = bCopy( i ) / ACopy( i, i )

           ! Substitute the now known value of x( i ), reducing the order of
           ! the system by 1.

           bCopy = bCopy - x( i ) * ACopy( :, i )
        enddo
     endif

     ! Determine the value of x( 1 ) as a special case.

     if ( successful ) x( 1 ) = bCopy( 1 ) / ACopy( 1, 1 )

     ! Prepare the return value of the function.

     gaussianElimination = successful

   end function gaussianElimination

   ! Output A in Matlab format, using name in the Matlab assignment statement.

   subroutine printMatrix( A, name )
     implicit none
     real, dimension( :, : ) :: A ! Assume the shape of A.
     character name              ! Name for use in assignment, ie, name =
......
     integer n, m, i, j
     n = size( A, 1 )
     m = size( A, 2 )
     write( *, fmt="(a1,a5)", advance = "no" ) name, ' = [ '

     ! Output the matrix, except for the last row, which needs no `;'.

     do i = 1, n-1

        ! Output current row.

        do j = 1, m-1
           write( *, fmt="(f10.6,a2)", advance = "no" ) A( i, j ), ', '
        enddo
```

```
       ! Output last element in row and end current row.
       write( *, fmt="(f10.6,a1)" ) A( i, m ), ';'
    enddo
    ! Output the last row.
    do j = 1, m-1
       write( *, fmt="(f10.6,a2)", advance = "no" ) A( i, j ), ', '
    enddo
    ! Output last element in row and end.
    write( *, fmt="(f10.6,a1)" ) A( i, m ), ']'
  end subroutine printMatrix

  ! Output b in Matlab format, using name in the Matlab assignment statement.
  subroutine printVector( b, name )
    implicit none
    real, dimension( : ) :: b ! Assume the shape of b.
    character name ! Name for use in assignment, ie, name = ......
    integer n, i
    n = size( b )
    write( *, fmt="(a1,a5)", advance = "no" ) name, ' = [ '
    do i = 1, n-1
       write( *, fmt = "(f10.6,a2)", advance = "no" ) b( i ), ', '
    enddo
    write( *, fmt = "(f10.6,a2)" ) b( n ), ']'
  end subroutine printVector
end module GaussianSolver

! A program to solve linear systems using the GaussianSolver module.
program SolveLinearSystem
  ! Include the module for the various linear solvers.
  use GaussianSolver
  implicit none
  integer, parameter :: N = 5 ! Order of the linear system.
  real, parameter :: TOO_SMALL = 1.0e-7 ! Threshold for pivots.
  ! Declare the necessary arrays and vectors to solve the linear system
  ! A x = b.
  real, dimension( N, N ) :: A ! Coefficient matrix.
  real, dimension( N ) :: x, b ! Vector of unknowns, and right hand side.
  real, dimension( N, N ) :: LU ! Matrix for LU factorization of A.
  logical successful ! Status of computations.
  ! The intrinsic subroutine, random_number, fills a real array or scalar,
  ! with uniformly distributed random variates in the interval [0,1).
  call random_number( A ) ! Initialize the coefficient matrix.
  call random_number( b ) ! Initialize the right-hand side.
  ! Output the matrix in Matlab format for ease of checking the solution.
  call printMatrix( A, 'A' )
  call printVector( b, 'b')
  ! Use Gaussian elmination to calcuate the solution of the linear system.
  ! The call below uses the default threshold specified in the
  ! GaussianSolver module by omitting the optional argument.
  successful = gaussianElimination( A, b, x )
  print *, '==================================='
  print *, 'Gaussian Elimination:'
  print *, '-----------------------------------'
  if ( successful ) then
     call printVector( x, 'x' )
     print *, 'Infinity Norm of Difference = ', &
          maxval( abs ( matmul( A, x ) - b ) )
  else
     print *, 'No unique solution or threshold passed.'
  endif
end program SolveLinearSystem
```

Figure 8. gauss.f90 – A Fortran 90 program for performing Gaussian Elimination.

Appendix A. Compiling and running a parallel program.

Parallel computing facilities at the AHPCC include a 128-node IBM SP2 (consisting of eight 16-node compute towers and one communications tower) and workstation clusters. The first tower of the SP2 contains various servers that manage the SP2 and the four interactive nodes. The latter have the imaginative names of fr1n05.arc.unm.edu (frame 1 node 5), fr1n06.arc.unm.edu, fr1n07.arc.unm.edu, and fr1n08.arc.unm.edu. These are the nodes from which you will compile and submit your jobs. Each node contains 64MB RAM and 1GB of hard disk space.

In this tutorial, we present instructions for compiling and running your code either on the SP2 or on a cluster of workstations. *Unless the SP2 is unavailable, we recommend running your code on the SP2 during daytime hours, so that you do not adversely affect the interactive use of the workstations.*

Compiling and linking MPI code

The compilers for MPI are invoked via scripts that invoke the standard compilers with additional command line information about the MPI libraries.

On the IBM SP2, the standard Fortran compiler for MPI programs is **mpxlf** , which invokes the **xlf** compiler to compile f77 and f90 codes, respectively. The difference between these compilers and the serial compilers is simply the choice of libraries. For C and C++ codes, the standard IBM compilers are **mpcc** and **mpCC**, respectively. The calling syntax for these compilers is

IBM_MPI_Compiler_Script [standard compiler flags] program file(s) [object files]

On the LINUX or the AIX cluster, the Fortran compilers are **mpif77** and **mpif90** and the C/C++ compilers are **mpicc** and **mpiCC**. These are scripts which should be located in */usr/local/bin*. On the LINUX cluster, the only MPI library available is the MPICH library. On the AIX cluster, you can choose between MPICH and IBM's own message passing library via command line arguments to the scripts**.** These libraries are selected as command line options in the scripts.

The calling syntax for these scripts is as follows:

```
MPI_Compiler_Script [script options] -- [compiler options]
```

where the script options control the selection of the library and architecture and the compiler options are options for the underlying Fortran compiler. The script options are described below. The double dashes, **--**, are used to separate the script options from the usual compiler options. Depending upon whether the compiler script is **mpif77** or **mpif90**, some options for the underlying are automatically set.

To compile a Fortran 77 program contained in a single file for a cluster of workstations using the MPICH library, the command is

```
mpif77 –i mpich –c ch_p4 -- -o prgm prgm.f
```

where the **–i mpich** and the **–c ch_p4** are script flags that link in the MPICH libraries needed for workstation clusters and specify the communications method to be used. **ch_p4** specifies communications over a TCP/IP connection. The corresponding command to compile a Fortran 90 code is

```
mpif90 –i mpich –c ch_p4 -- -o prgm prgm.f
```

Both commands invoke the same compiler, in this case **xlf**, but with options specific to either Fortran 77 or Fortran 90. To compile an object file to be linked at a later time, we give the usual **–c** flag in the compiler options, e.g.

```
mpif77 –i mpich –c ch_p4 -- -c prgm.f
```

Programs can be compiled for the SP2 with either the MPICH library,

```
mpif77 –i mpich –c ch_eui -- -o prgm prgm.f
```

or with the IBM library,

```
mpif77 –i ibm -- -o prgm prgm.f
```

where **–i ibm** is the script option that specifies the IBM MPI library.

Running a compiled MPI program on a cluster of workstations

MPICH

Programs compiled for a cluster of workstations using MPICH can be run in one of two ways, either automatically using the script *mpirun* or manually. The program should be run manually if you are running on a heterogeneous cluster. The syntax for *mpirun* is:

```
mpirun –n3 –a ARCH -- prgm
```

where the option **–n3** specifies 3 processors and **–a ARCH** specifies the architecture (rs6000, sun, etc.) of the executable and the name of the sub-directory of the current directory containing the executable program.

To run an MPICH job manually, you will need to create a list of the machines where you want to run the program and the locations of the executables on each. (Each machine must have an executable matching its architecture.) The format of the list is as follows:

```
local    0
turing02 1  /home/user/program_directory/Linux/program
grants   1  /home/user/program_directory/rs6000/program
gallup   1  /home/user/program_directory/rs6000/program
```

The first line in the file refers to the copy of the program that is to be run locally. The second and subsequent lines consist of the name of the workstation, a 1 denoting the number of processes to start, and the full path to the executable to run on that workstation. (*Note: To run more than one copy of the program on a single machine, enter the name of the machine on more than 1 line. Entering a number greater than 1 for the number of processes to start does not work.*) After storing this list in the file **filename.pg**, you run the program by typing the command

```
program [–p4pg filename.pg]
```

Note the arguments in brackets are optional if **filename.pg=program.pg**.

IBM MPI

Programs using IBM's MPI implementation can also be run interactively, either on the interactive nodes of the SP2 or on the AIX workstation cluster. Note however that both uses are generally discouraged, except perhaps for debugging purposes, since these machines are heavily used. For completeness, we give the necessary directions here.

First, you will need to create a list of the nodes on which you intend to run your code and store them in a file. Then, you will need to create a script to initialize several environment variables relating to the parallel operating environment (POE). This script should be *sourced* before you run your program. (In other shells, you need to be sure to export the environment variables in the script.) Below is an example script. It assumes you are using the C-shell or a derivative of it (*csh* or *tcsh*).

```
setenv MP_PROCS 2
setenv MP_HOSTLIST host.list
setenv MP_EUILIB ip
setenv MP_EUIDEVICE en0
setenv MP_INFOLEVEL 2
setenv MP_RESD no
setenv MP_RMPOOL 0
setenv MP_LABELIO yes
setenv MP_PGMMODEL spmd
setenv MP_PARTITION 1
setenv MP_RETRY 30
setenv MP_RETRYCOUNT 5
```

Specify Ethernet communications.

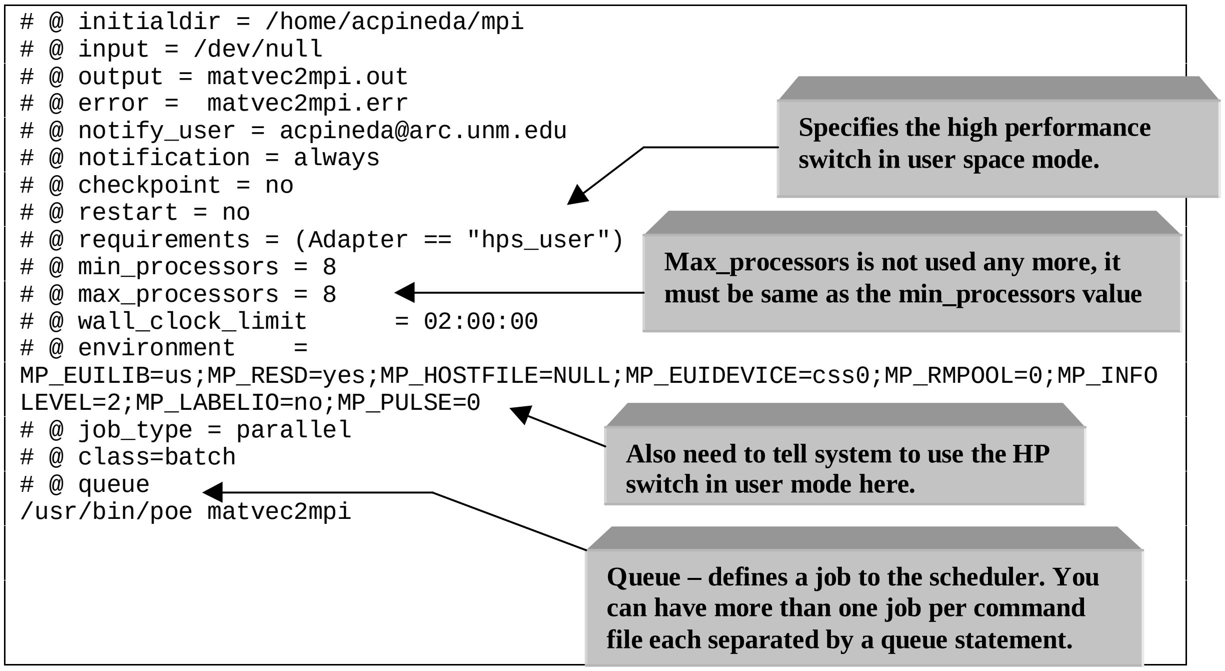
Here **host**.**list** is a file containing the list of the nodes on which the job will run. For details on the meanings of the environment variables, consult the **poe** man page.

Running your program in a batch queue on the SP2

If you have compiled your program for the SP2, you can submit your job to a batch queue using the Maui Scheduler, a batch job scheduling application created at the Maui High Performance Computing Center. The Maui scheduler is based upon the IBM *Loadleveler* scheduling application. The Maui scheduler provides the facility for submitting and processing batch jobs within a network of machines. It matches the job requirements specified by users with the best available machine resources.

Before using the scheduler, you will need to make a few minor modifications to your .cshrc file. Open the file with your favorite editor and comment out the lines that initialize any environment variables that start with **MP_**. These are the same variables found in the script created in the previous section. (The comment character is the pound sign, #.) These environment variables are normally configured by the scripts you will create for the scheduler and since your job will run your .cshrc file *after* the scheduler configures them, your program environment could be put into a confused state.

The scheduler uses a control file that tells how many processors you intend to use, where your input and output files are, where your program is, and how much time it can use, among other things. The control file is usually named with a .cmd extension, but this is not required. The control file used to run the *matvec2mpi* program is included below.



```
# @ initialdir = /home/acpineda/mpi
# @ input = /dev/null
# @ output = matvec2mpi.out
# @ error =  matvec2mpi.err
# @ notify_user = acpineda@arc.unm.edu
# @ notification = always
# @ checkpoint = no
# @ restart = no
# @ requirements = (Adapter == "hps_user")
# @ min_processors = 8
# @ max_processors = 8
# @ wall_clock_limit      = 02:00:00
# @ environment    =
MP_EUILIB=us;MP_RESD=yes;MP_HOSTFILE=NULL;MP_EUIDEVICE=css0;MP_RMPOOL=0;MP_INFO
LEVEL=2;MP_LABELIO=no;MP_PULSE=0
# @ job_type = parallel
# @ class=batch
# @ queue
/usr/bin/poe matvec2mpi
```

Specifies the high performance switch in user space mode.
Max_processors is not used any more, it must be same as the min_processors value
Also need to tell system to use the HP switch in user mode here.
Queue – defines a job to the scheduler. You can have more than one job per command file each separated by a queue statement.

Figure 9. matvec2mpi.cmd – a control file for running matvec2mpi in the SP2 batch queue.

The lines starting with **# @** are options used to set up the environment in which the program will run. The remaining lines run as a shell script on each processor. Most of the options in the control file are fairly obvious. For the purposes of this course, you will only need to modify a few lines. The **input**, **output**, and **error** lines tell the scheduler the names of the input, output and error files. The **initialdir** is the directory containing the executable; **notify_user** is the e-mail address to which error messages are sent. The **notification** = **always** line tells the scheduler to always send e-mail to the user in response to various events, such as errors and program termination. Once you have your command file and program debugged, you will probably want to change *always* to *never*. **min_processors** and **max_processors** control the number of processors that are allocated to your program. **wall_clock_limit** sets a time limit on the job. The **requirements** and **environment** lines tell the scheduler to enable communications over the high-speed switch on the SP2. The class line specifies the batch queue. Currently, we have only one queue called **batch**. The **queue** line tells the scheduler to treat all the option lines above it as options for a single parallel job. (That is, multiple jobs processing different inputs can be submitted from the same file.) Finally, the line **/usr/bin/poe matvec2mpi** runs the program *matvec2mpi* in the parallel operating environment. For more details on creating command files, the user is referred to the relevant MHPCC web site, http://www.mhpcc.edu/training/workshop/html/workshop.html. Serial programs can be run on the SP2 by changing the number of processors to 1 and the **job_type** to serial.

A word is in order about how the scheduler prioritizes tasks. Basically, tasks for non-privileged users are prioritized based upon the amount of system resources they consume and the amount of time they have been waiting in the queue. The more resources, i.e. number of nodes and amount of CPU time required, you use the longer you can expect to wait for your job to run. Therefore, it is to your advantage to estimate the time required by your job as well as you can. Most of the example codes provided in this tutorial will run in a handful of seconds, so you should set **wall_clock_limit** to be no more than a few minutes. The queue also enforces limits on the number of jobs per user, the number of nodes per job, and total wall clock time. Currently, the following limits are in effect: 2 jobs per user, 8 nodes per job, and 36 hours of wall clock time per job.

You submit your command file to the scheduler using the command, **llsubmit,** which has the syntax:

```
llsubmit cmdfile.cmd
```

the scheduler will then reply with:

submit: The job "fr1n05.332" has been submitted.

The first part of the job name fr1n05 specifies the node from which you submitted the job.

As mentioned previously, the uncommented portion of the command file is run as a shell script on the SP2. Hence, you can use ordinary Unix commands to perform setup and cleanup operations. Typically, this is done for disk I/O intensive programs. Such programs must access disk drives that are locally attached to the SP2 in order to perform optimally. Ask your system administrator if such space is available.

You can check the status of your program with either the *Loadleveler* status program **llq** or with the Maui status program **showq.** To use the latter, you must ensure that the directory **/home/loadl/maui/bin** in your command **path**. If it is not present, your environment initialization scripts can be updated by running the **reset_environment** command. You can kill a running program using the command **llcancel** or better yet with the Maui scheduler command **canceljob.**

Appendix B. Common MPI Library Calls

In this section, we document the more commonly used MPI library calls. For a more complete listing the reader is referred to "MPI: The Complete Reference". It can be downloaded from the CS471 class web site. These calls fall into three groups: (1) commands used for initializing an MPI session, (2) commands that actually do MPI communications, and (3) commands for terminating an MPI session. In Fortran, all MPI calls are subroutine calls. In C, they are function calls that return an integer error message. In C, the names of the MPI functions are case sensitive. In Fortran, they are not.

Initializing communications

There are three MPI calls that you will always use in initializing your MPI programs. They are MPI_INIT which starts up communications and initializes data structures used for communication, MPI_COMM_RANK which allows a process to determine its ID within the group of processes running under MPI, and MPI_COMM_SIZE which allows a process to determine how many MPI processes are running the current program. In Fortran, these subroutines are called as follows:

MPI_INIT USAGE
CALL MPI_INIT (IERROR)

MPI_INIT takes a single argument IERROR (integer, output) which returns the value MPI_SUCCESS if the call completed successfully and one of 19 other error codes in the event of a failure. MPI_INIT can only be called once in a program.

MPI_COMM_RANK USAGE
CALL MPI_COMM_RANK (COMM, RANK, IERROR)

In MPI, processes are labeled with an integer rank from 0 through N-1 where N is the number of processes. MPI_COMM_RANK takes 3 integer arguments:

- The argument COMM (integer, input) is the communicator, which is a handle to an internal MPI structure that defines the set of processes that may communicate with each other. It is a local object that represents a communication domain. A communication domain is a global structure that allows processes in a group to communicate with each other or with processes in another group. Processes can belong to more than one group and have a different rank in each group. Unless you are doing something requiring specialized communications, use the predefined value MPI_COMM_WORLD

here. This tells the program to use all available processors in a single processor group without additional management.
- The argument RANK (integer, output) is a number between 0 and N-1 that serves the label for a process within the communications group.
- IERROR (integer, output) is as defined above.

MPI_COMM_SIZE USAGE
CALL MPI_COMM_SIZE (COMM, SIZE, IERROR)

MPI_COMM_SIZE is used to find out how many processes are running the current program. This number is returned in the integer SIZE. The arguments COMM and IERROR are as described above.

Communications Calls

MPI communications calls generally fall into 2 classes: point-to-point and collective communications. In addition there are calls for defining data types other than the standard ones included in MPI.

Point-to-point Communications Calls

MPI_SEND USAGE
CALL MPI_SEND(BUF, COUNT, DATATYPE, DEST, TAG, COMM, IERROR)

MPI_SEND is used to send a message between two processes. It performs a "blocking" send, which in this case means that it does not return until the user can safely use the message buffer BUF again. It does not necessarily mean that the receiving process has received the data yet. BUF (input) is an array of type <type> containing the data to be sent. COUNT (integer, input) is the number of elements in BUF. DATATYPE (integer, input) is an integer code that tells MPI what is the type <type> of BUF. The allowed pre-defined values of DATATYPE in Fortran are MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_COMPLEX, MPI_DOUBLE_COMPLEX, MPI_LOGICAL, MPI_CHARACTER, MPI_BYTE, and MPI_PACKED. There is another set of pre-defined values for C programs. MPI has functions for the construction of user-defined types built up from these types, see below. DEST (integer, input) is the rank of the destination process. The TAG (integer, input) acts as a label to match corresponding SENDs and RECVs. The range of valid tag values is from 0 to MPI_TAG_UB. The MPI standard requires MPI_TAG_UB to be at least 32767. The other arguments are as defined previously. The details of how the MPI_SEND implements the blocking send operation are left to the person implementing the particular MPI library. Other more specialized versions of send are MPI_BSEND, MPI_ISEND, MPI_SSEND, and MPI_RSEND allow finer control over how the messages are sent. The reader is referred to the MPI manual for details on these functions.

MPI_RECV USAGE
CALL MPI_RECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, STATUS, IERROR)

MPI_RECV is used to receive messages sent by MPI send calls. It is blocking call. It does not return until a matching send has been posted. BUF (output) is an array of type <type> containing the received data. COUNT (integer, input) is the number of elements in BUF. DATATYPE (integer, input) is an integer code describing <type>. See MPI_SEND for the codes. SOURCE (integer, input) is the rank of the sending process. TAG (integer, input) is the label matching SENDs and RECVs. Detailed status information is returned by the integer array STATUS (output) of size MPI_STATUS_SIZE. The other arguments are as in previously described calls.

MPI_IRECV USAGE
CALL MPI_IRECV(BUF, COUNT, DATATYPE, SOURCE, TAG, COMM, REQUEST, IERROR)

MPI_IRECV is the non-blocking version of MPI_RECV. Its arguments are identical with the MPI_RECV call above except that one argument, the request handle REQUEST (integer, output), replaces the STATUS argument. The request handle is used to query the status of the communication or to wait for its completion. The receive operation is completed with an MPI_WAIT call.

MPI_WAIT USAGE
CALL MPI_WAIT(REQUEST, STATUS, IERROR)

MPI_WAIT is used to complete an MPI_ISEND or MPI_IRECV operation. The handle REQUEST is obtained from MPI_ISEND or MPI_IRECV. The STATUS argument is the same as that in MPI_RECV.

MPI_SENDRECV USAGE
CALL MPI_SENDRECV(SENDBUF, SENDCOUNT, SENDTYPE, DEST, SENDTAG, RECVBUF,RECVCOUNT, RECVTYPE, SOURCE, RECVTAG, COMM, STATUS, IERROR)

MPI_SENDRECV is a combination of an MPI_SEND and an MPI_RECV operation. The send and receive buffers must be disjoint, and may have different lengths and data types. The combined operation is frequently used as a means of avoiding potential processor deadlocks in for example a shift operation across a chain of processes.

The array SENDBUF (input) of type <type> contains the data being sent. SENDCOUNT (integer, input) is the number of elements in SENDBUF. SENDTYPE (integer, input) describes the type of data being sent. DEST (integer, input) contains the rank of the destination process. SENDTAG (integer, input) is the tag for the sending operation. The array RECVBUF (output) of type <type> contains the data being received. RECVCOUNT (integer, input) is the size of the RECVBUF buffer. RECVTYPE (integer, input) is the type of the data being received. SOURCE (integer, input) is the rank of the source process. RECVTAG (integer, input) is the tag for the receiving operation. The other arguments are as described previously.

Collective Communications Calls

MPI_BARRIER USAGE
CALL MPI_BARRIER (COMM, IERROR)

MPI_BARRIER is used to synchronize processes. All processes stop at this call until every process has reached it. The arguments COMM and IERROR are as described above.

MPI_BCAST USAGE
CALL MPI_BCAST (BUFFER, COUNT, DATATYPE, ROOT, COMM, IERROR)

MPI_BCAST broadcasts values from the sending, ROOT(integer, input), process to all processes (including itself) via BUFFER. The array BUFFER (input/output) is an array of type <type> containing the data to be sent/received. COUNT(integer, input) is the number of elements in BUFFER. DATATYPE (integer, input) is the code describing <type>. The other arguments are as described previously.

MPI_GATHER USAGE
CALL MPI_GATHER (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT,RECVTYPE, ROOT, COMM, IERROR)

MPI_GATHER collects an array that has been distributed across multiple processes back to the ROOT process. The array SENDBUF (input) is an array containing the data from the sending process to be collected back to ROOT(integer, input). SENDCOUNT (integer, input) is the number of elements in SENDBUF. SENDTYPE (integer, input) is the code describing <type> of the data in SENDBUF. The array RECVBUF (output) is an array containing the data collected from all processes. It is ignored for all but the ROOT process. RECVCOUNT (integer, input) is the number of elements received from any one processor. *(Note that this value is not SENDCOUNT times the number of processors.)* RECVTYPE (integer, input) is the integer code describing the <type> of the data in the RECVBUF array. The other arguments are as described previously.

This call assumes that an equal amount of data is distributed across the processes. The more general MPI_GATHERV call allows processes to send unequal amounts of data to ROOT. See the MPI reference for details.

MPI_SCATTER USAGE
CALL MPI_SCATTER (SENDBUF, SENDCOUNT, SENDTYPE, RECVBUF, RECVCOUNT, RECVTYPE, ROOT, COMM, IERROR)

MPI_SCATTER distributes an array from one task to all other tasks in the group. The array SENDBUF (input) is an array containing the data to be distributed among the processes. SENDCOUNT (integer, input) is the number of elements sent to each process. SENDTYPE (integer, input) is the code describing the type of the data in the SENDBUF array. The RECVBUF (output) is an array containing the data distributed to the receiving process. RECVCOUNT (integer, input) is the number of elements received by each process. RECVTYPE (integer, input) is the MPI code describing the type of the data in the RECVBUF array. ROOT (integer, input) is the rank of the process that will be sending the data. The other arguments are as previously described.

Like MPI_GATHER, this call assumes that the data is to be distributed in equal size pieces across processes. The more general MPI_SCATTERV call allows processes to distribute unequal amounts of data among processes.

MPI_REDUCE USAGE
MPI_REDUCE (SENDBUF, RECVBUF, COUNT, DATATYPE, OP, ROOT, COMM, IERROR)

MPI_REDUCE performs a reduction operation across processors on the data stored in SENDBUF, placing the reduced results in RECVBUF. By a reduction operation, we mean an operation such as a sum, multiplication, logical AND, etc. depending upon the data involved. The array SENDBUF (input) is an array containing the data upon which the reduction operation, OP, is to be performed. The result is forwarded to ROOT. The array RECVBUF (output) is an array containing the result of the reduction. COUNT (integer, input) is the number of elements in SENDBUF and RECVBUF. DATATYPE (integer, input) is the MPI code describing the type of the data in SENDBUF and RECVBUF. OP (integer, input) is the MPI code for the reduction operation to be performed. The common values for OP are MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND (logical and), MPI_LOR (logical or), and MPI_LXOR (logical exclusive or). ROOT (integer, input) is the rank of the process that will be receiving the data. The other arguments are as described previously.

Creating Derived Data Types

MPI_TYPE_CONTIGUOUS USAGE
MPI_TYPE_CONTIGUOUS(COUNT, OLDTYPE, NEWTYPE, IERROR)

A COUNT (integer, input) number of copies of OLDTYPE (integer, input) are concatenated to form the NEWTYPE (integer, output).

MPI_TYPE_VECTOR USAGE
MPI_TYPE_VECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)

MPI_TYPE_VECTOR allows the construction of a new data type NEWTYPE (integer, output) that consists of COUNT (integer, input) blocks of OLDTYPE (integer, input) of length BLOCKLENGTH(integer, input) that are spaced STRIDE (integer, input) units apart. The BLOCKLENGTH and STRIDE are in units of the OLDTYPE. COUNT and BLOCKLENGTH must be non-negative, while STRIDE can be of either sign.

MPI_TYPE_HVECTOR USAGE
MPI_TYPE_HVECTOR(COUNT, BLOCKLENGTH, STRIDE, OLDTYPE, NEWTYPE, IERROR)

MPI_TYPE_HVECTOR is identical to MPI_TYPE_VECTOR except that the STRIDE argument is in units of bytes instead of the size of the OLDTYPE. Typically used in conjunction with the function MPI_TYPE_EXTENT.

MPI_TYPE_EXTENT USAGE
MPI_TYPE_EXTENT(DATATYPE, EXTENT, IERROR)

MPI_TYPE_EXTENT returns the size EXTENT(integer, output) in bytes of a data type DATATYPE (integer, input). This information is used to compute stride information in bytes for MPI functions such as MPI_TYPE_HVECTOR. *Note to C programmers: the C operator sizeof() should not be used in place of MPI_TYPE_EXTENT.*