

# MESSAGE-PASSING ENVIRONMENTS - MESSAGE-PASSING INTERFACE\*

Charles Severance  
Kevin Dowd

This work is produced by OpenStax-CNX and licensed under the  
Creative Commons Attribution License 3.0<sup>†</sup>

The Message-Passing Interface (MPI) was designed to be an industrial-strength message-passing environment that is portable across a wide range of hardware environments.

Much like High Performance FORTRAN, MPI was developed by a group of computer vendors, application developers, and computer scientists. The idea was to come up with a specification that would take the strengths of many of the existing proprietary message passing environments on a wide variety of architectures and come up with a specification that could be implemented on architectures ranging from SIMD systems with thousands of small processors to MIMD networks of workstations and everything in between.

Interestingly, the MPI effort was completed a year *after* the High Performance FORTRAN (HPF) effort was completed. Some viewed MPI as a portable message-passing interface that could support a good HPF compiler. Having MPI makes the compiler more portable. Also having the compiler use MPI as its message-passing environment insures that MPI is heavily tested and that sufficient resources are invested into the MPI implementation.

## 1 PVM Versus MPI

While many of the folks involved in PVM participated in the MPI effort, MPI is not simply a follow-on to PVM. PVM was developed in a university/research lab environment and evolved over time as new features were needed. For example, the group capability was not designed into PVM at a fundamental level. Some of the underlying assumptions of PVM were based “on a network of workstations connected via Ethernet” model and didn’t export well to scalable computers.<sup>1</sup> In some ways, MPI is more robust than PVM, and in other ways, MPI is simpler than PVM. MPI doesn’t specify the system management details as in PVM; MPI doesn’t specify how a virtual machine is to be created, operated, and used.

## 2 MPI Features

MPI has a number of useful features beyond the basic send and receive capabilities. These include:

**Communicators:** : A communicator is a subset of the active processes that can be treated as a group for collective operations such as broadcast, reduction, barriers, sending, or receiving. Within each communicator, a process has a *rank* that ranges from zero to the size of the group. A process may be a

---

\*Version 1.3: Aug 25, 2010 11:10 am +0000

<sup>†</sup><http://creativecommons.org/licenses/by/3.0/>

<sup>1</sup>One should not diminish the positive contributions of PVM, however. PVM was the first widely available portable message-passing environment. PVM pioneered the idea of heterogeneous distributed computing with built-in format conversion.

member of more than one communicator and have a different rank within each communicator. There is a default communicator that refers to all the MPI processes that is called `MPI_COMM_WORLD`.

**Topologies:** : A communicator can have a topology associated with it. This arranges the processes that belong to a communicator into some layout. The most common layout is a Cartesian decomposition. For example, 12 processes may be arranged into a  $3 \times 4$  grid.<sup>2</sup> Once these topologies are defined, they can be queried to find the neighboring processes in the topology. In addition to the Cartesian (grid) topology, MPI also supports a graph-based topology.

**Communication modes:** : MPI supports multiple styles of communication, including blocking and non-blocking. Users can also choose to use explicit buffers for sending or allow MPI to manage the buffers. The nonblocking capabilities allow the overlap of communication and computation. MPI can support a model in which there is no available memory space for buffers and the data must be copied directly from the address space of the sending process to the memory space of the receiving process. MPI also supports a single call to perform a send and receive that is quite useful when processes need to exchange data.

**Single-call collective operations:** : Some of the calls in MPI automate collective operations in a single call. For example, the broadcast operation sends values from the master to the slaves and receives the values on the slaves in the same operation. The net result is that the values are updated on all processes. Similarly, there is a single call to sum a value across all of the processes to a single value. By bundling all this functionality into a single call, systems that have support for collective operations in hardware can make best use of this hardware. Also, when MPI is operating on a shared-memory environment, the broadcast can be simplified as all the slaves simply make a local copy of a shared variable.

Clearly, the developers of the MPI specification had significant experience with developing message-passing applications and added many widely used features to the message-passing library. Without these features, each programmer needed to use more primitive operations to construct their own versions of the higher-level operations.

### 3 Heat Flow in MPI

In this example, we implement our heat flow problem in MPI using a similar decomposition to the PVM example. There are several ways to approach the problem. We could almost translate PVM calls to corresponding MPI calls using the `MPI_COMM_WORLD` communicator. However, to showcase some of the MPI features, we create a Cartesian communicator:

```
PROGRAM MHEATC
  INCLUDE 'mpif.h'
  INCLUDE 'mpef.h'
  INTEGER ROWS, COLS, TOTCOLS
  PARAMETER(MAXTIME=200)
  * This simulation can be run on MINPROC or greater processes.
  * It is OK to set MINPROC to 1 for testing purposes
  * For a large number of rows and columns, it is best to set MINPROC
  * to the actual number of runtime processes
  PARAMETER(MINPROC=2) PARAMETER(ROWS=200, TOTCOLS=200, COLS=TOTCOLS/MINPROC)
  DOUBLE PRECISION RED(0:ROWS+1, 0:COLS+1), BLACK(0:ROWS+1, 0:COLS+1)
  INTEGER S, E, MYLEN, R, C
```

---

<sup>2</sup>Sounds a little like HPF, no?

```

INTEGER TICK,MAXTIME
CHARACTER*30 FNAME

```

The basic data structures are much the same as in the PVM example. We allocate a subset of the heat arrays in each process. In this example, the amount of space allocated in each process is set by the compile-time variable MINPROC. The simulation can execute on more than MINPROC processes (wasting some space in each process), but it can't execute on less than MINPROC processes, or there won't be sufficient total space across all of the processes to hold the array:

```

INTEGER COMM1D, INUM, NPROC, IERR
INTEGER DIMS(1), COORDS(1)
LOGICAL PERIODS(1)
LOGICAL REORDER
INTEGER NDIM
INTEGER STATUS(MPI_STATUS_SIZE)
INTEGER RIGHTPROC, LEFTPROC

```

These data structures are used for our interaction with MPI. As we will be doing a one-dimensional Cartesian decomposition, our arrays are dimensioned to one. If you were to do a two-dimensional decomposition, these arrays would need two elements:

```

PRINT *, 'Calling MPI_INIT'
CALL MPI_INIT( IERR )
PRINT *, 'Back from MPI_INIT'
CALL MPI_COMM_SIZE( MPI_COMM_WORLD, NPROC, IERR )

```

The call to MPI\_INIT creates the appropriate number of processes. Note that in the output, the PRINT statement before the call only appears once, but the second PRINT appears once for each process. We call MPI\_COMM\_SIZE to determine the size of the global communicator MPI\_COMM\_WORLD. We use this value to set up our Cartesian topology:

```

* Create new communicator that has a Cartesian topology associated
* with it - MPI_CART_CREATE returns COMM1D - A communicator descriptor

```

```

DIMS(1) = NPROC
PERIODS(1) = .FALSE.
REORDER = .TRUE.
NDIM = 1

CALL MPI_CART_CREATE(MPI_COMM_WORLD, NDIM, DIMS, PERIODS,
+ REORDER, COMM1D, IERR)

```

Now we create a one-dimensional (NDIM=1) arrangement of all of our processes (MPI\_COMM\_WORLD). All of the parameters on this call are input values except for COMM1D and IERR. COMM1D is an integer “communicator handle.” If you print it out, it will be a value such as 134. It is not actually data, it is merely a handle that is used in other calls. It is quite similar to a file descriptor or unit number used when performing input-output to and from files.

The topology we use is a one-dimensional decomposition that isn’t periodic. If we specified that we wanted a periodic decomposition, the far-left and far-right processes would be neighbors in a wrapped-around fashion making a ring. Given that it isn’t periodic, the far-left and far-right processes have no neighbors.

In our PVM example above, we declared that Process 0 was the far-right process, Process NPROC-1 was the far-left process, and the other processes were arranged linearly between those two. If we set REORDER to .FALSE., MPI also chooses this arrangement. However, if we set REORDER to .TRUE., MPI may choose to arrange the processes in some other fashion to achieve better performance, assuming that you are communicating with close neighbors.

Once the communicator is set up, we use it in all of our communication operations:

```
* Get my rank in the new communicator

CALL MPI_COMM_RANK( COMM1D, INUM, IERR)
```

Within each communicator, each process has a rank from zero to the size of the communicator minus 1. The MPI\_COMM\_RANK tells each process its rank within the communicator. A process may have a different rank in the COMM1D communicator than in the MPI\_COMM\_WORLD communicator because of some reordering.

Given a Cartesian topology communicator,<sup>3</sup> we can extract information from the communicator using the MPI\_CART\_GET routine:

```
* Given a communicator handle COMM1D, get the topology, and my position
* in the topology

CALL MPI_CART_GET(COMM1D, NDIM, DIMS, PERIODS, COORDS, IERR)
```

In this call, all of the parameters are output values rather than input values as in the MPI\_CART\_CREATE call. The COORDS variable tells us our coordinates within the communicator. This is not so useful in our one-dimensional example, but in a two-dimensional process decomposition, it would tell our current position in that two-dimensional grid:

```
* Returns the left and right neighbors 1 unit away in the zeroth dimension
* of our Cartesian map - since we are not periodic, our neighbors may
* not always exist - MPI_CART_SHIFT handles this for us

CALL MPI_CART_SHIFT(COMM1D, 0, 1, LEFTPROC, RIGHTPROC, IERR)
```

---

<sup>3</sup>Remember, each communicator may have a topology associated with it. A topology can be grid, graph, or none. Interestingly, the MPI\_COMM\_WORLD communicator has no topology associated with it.

```

CALL MPE_DECOMP1D(TOTCOLS, NPROC, INUM, S, E)
MYLEN = ( E - S ) + 1
IF ( MYLEN.GT.COLS ) THEN
  PRINT *, 'Not enough space, need', MYLEN, ' have ', COLS
  PRINT *, TOTCOLS, NPROC, INUM, S, E
  STOP
ENDIF
PRINT *, INUM, NPROC, COORDS(1), LEFTPROC, RIGHTPROC, S, E

```

We can use `MPI_CART_SHIFT` to determine the rank number of our left and right neighbors, so we can exchange our common points with these neighbors. This is necessary because we can't simply send to `INUM-1` and `INUM+1` if MPI has chosen to reorder our Cartesian decomposition. If we are the far-left or far-right process, the neighbor that doesn't exist is set to `MPI_PROC_NULL`, which indicates that we have no neighbor. Later when we are performing message sending, it checks this value and sends messages only to real processes. By not sending the message to the "null process," MPI has saved us an `IF` test.

To determine which strip of the global array we store and compute in this process, we call a utility routine called `MPE_DECOMP1D` that simply does several calculations to evenly split our 200 columns among our processes in contiguous strips. In the PVM version, we need to perform this computation by hand.

The `MPE_DECOMP1D` routine is an example of an extended MPI library call (hence the MPE prefix). These extensions include graphics support and logging tools in addition to some general utilities. The MPE library consists of routines that were useful enough to standardize but not required to be supported by all MPI implementations. You will find the MPE routines supported on most MPI implementations.

Now that we have our communicator group set up, and we know which strip each process will handle, we begin the computation:

```

* Start Cold

DO C=0,COLS+1
  DO R=0,ROWS+1
    BLACK(R,C) = 0.0
  ENDDO
ENDDO

```

As in the PVM example, we set the plate (including boundary values) to zero.

All processes begin the time step loop. Interestingly, like in PVM, there is no need for any synchronization. The messages implicitly synchronize our loops.

The first step is to store the permanent heat sources. We need to use a routine because we must make the store operations relative to our strip of the global array:

```

* Begin running the time steps
DO TICK=1,MAXTIME

* Set the persistent heat sources
CALL STORE(BLACK,ROWS,COLS,S,E,ROWS/3,TOTCOLS/3,10.0,INUM)
CALL STORE(BLACK,ROWS,COLS,S,E,2*ROWS/3,TOTCOLS/3,20.0,INUM)

```

```
CALL STORE(BLACK,ROWS,COLS,S,E,ROWS/3,2*TOTCOLS/3,-20.0,INUM)
CALL STORE(BLACK,ROWS,COLS,S,E,2*ROWS/3,2*TOTCOLS/3,20.0,INUM)
```

All of the processes set these values independently depending on which process has which strip of the overall array.

Now we exchange the data with our neighbors as determined by the Cartesian communicator. Note that we don't need an IF test to determine if we are the far-left or far-right process. If we are at the edge, our neighbor setting is MPI\_PROC\_NULL and the MPI\_SEND and MPI\_RECV calls do nothing when given this as a source or destination value, thus saving us an IF test.

Note that we specify the communicator COMM1D because the rank values we are using in these calls are relative to that communicator:

```
* Send left and receive right
  CALL MPI_SEND(BLACK(1,1),ROWS,MPI_DOUBLE_PRECISION,
+             LEFTPROC,1,COMM1D,IERR)
  CALL MPI_RECV(BLACK(1,MYLEN+1),ROWS,MPI_DOUBLE_PRECISION,
+             RIGHTPROC,1,COMM1D,STATUS,IERR)

* Send Right and Receive left in a single statement
  CALL MPI_SENDRCV(
+   BLACK(1,MYLEN),ROWS,COMM1D,RIGHTPROC,2,
+   BLACK(1,0),ROWS,COMM1D,LEFTPROC, 2,
+   MPI_COMM_WORLD, STATUS, IERR)
```

Just to show off, we use both the separate send and receive, and the combined send and receive. When given a choice, it's probably a good idea to use the combined operations to give the runtime environment more flexibility in terms of buffering. One downside to this that occurs on a network of workstations (or any other high-latency interconnect) is that you can't do both send operations first and then do both receive operations to overlap some of the communication delay.

Once we have all of our ghost points from our neighbors, we can perform the algorithm on our subset of the space:

```
* Perform the flow
  DO C=1,MYLEN
    DO R=1,ROWS
      RED(R,C) = ( BLACK(R,C) +
+               BLACK(R,C-1) + BLACK(R-1,C) +
+               BLACK(R+1,C) + BLACK(R,C+1) ) / 5.0
    ENDDO
  ENDDO

* Copy back - Normally we would do a red and black version of the loop
  DO C=1,MYLEN
    DO R=1,ROWS
      BLACK(R,C) = RED(R,C)
```

```

        ENDDO
    ENDDO
ENDDO

```

Again, for simplicity, we don't do the complete red-black computation.<sup>4</sup> We have no synchronization at the bottom of the loop because the messages implicitly synchronize the processes at the top of the next loop.

Again, we dump out the data for verification. As in the PVM example, one good test of basic correctness is to make sure you get exactly the same results for varying numbers of processes:

```

* Dump out data for verification
  IF ( ROWS .LE. 20 ) THEN
    FNAME = '/tmp/mheatcout.' // CHAR(ICHAR('0')+INUM)
    OPEN(UNIT=9,NAME=FNAME,FORM='formatted')
    DO C=1,MYLEN
      WRITE(9,100)(BLACK(R,C),R=1,ROWS)
100    FORMAT(20F12.6)
    ENDDO
    CLOSE(UNIT=9)
  ENDIF

```

To terminate the program, we call `MPI_FINALIZE`:

```

* Lets all go together
  CALL MPI_FINALIZE(IERR)
  END

```

As in the PVM example, we need a routine to store a value into the proper strip of the global array. This routine simply checks to see if a particular global element is in this process and if so, computes the proper location within its strip for the value. If the global element is not in this process, this routine simply returns doing nothing:

```

SUBROUTINE STORE(RED,ROWS,COLS,S,E,R,C,VALUE,INUM)
  REAL*8 RED(0:ROWS+1,0:COLS+1)
  REAL VALUE
  INTEGER ROWS,COLS,S,E,R,C,I,INUM
  IF ( C .LT. S .OR. C .GT. E ) RETURN
  I = ( C - S ) + 1
*   PRINT *, 'STORE, INUM,R,C,S,E,R,I', INUM,R,C,S,E,R,I, VALUE RED(R,I) = VALUE
  RETURN

```

---

<sup>4</sup>Note that you could do two time steps (one black-red-black iteration) if you exchanged two ghost columns at the top of the loop.

END

When this program is executed, it has the following output:

```
% mpif77 -c mheatc.f mheatc.f:
MAIN mheatc:
store:
% mpif77 -o mheatc mheatc.o -lmpe
% mheatc -np 4
Calling MPI_INIT
Back from MPI_INIT
Back from MPI_INIT
Back from MPI_INIT
Back from MPI_INIT
0 4 0 -1 1 1 50
2 4 2 1 3 101 150
3 4 3 2 -1 151 200
1 4 1 0 2 51 100
%
```

As you can see, we call `MPI_INIT` to activate the four processes. The `PRINT` statement immediately after the `MPI_INIT` call appears four times, once for each of the activated processes. Then each process prints out the strip of the array it will process. We can also see the neighbors of each process including `-1` when a process has no neighbor to the left or right. Notice that Process 0 has no left neighbor, and Process 3 has no right neighbor. MPI has provided us the utilities to simplify message-passing code that we need to add to implement this type of grid- based application.

When you compare this example with a PVM implementation of the same problem, you can see some of the contrasts between the two approaches. Programmers who wrote the same six lines of code over and over in PVM combined them into a single call in MPI. In MPI, you can think “data parallel” and express your program in a more data-parallel fashion.

In some ways, MPI feels less like assembly language than PVM. However, MPI does take a little getting used to when compared to PVM. The concept of a Cartesian communicator may seem foreign at first, but with understanding, it becomes a flexible and powerful tool.

## 4 Heat in MPI Using Broadcast/Gather

One style of parallel programming that we have not yet seen is the *broadcast/gather* style. Not all applications can be naturally solved using this style of programming. However, if an application can use this approach effectively, the amount of modification that is required to make a code run in a message-passing environment is minimal.

Applications that most benefit from this approach generally do a lot of computation using some small amount of shared information. One requirement is that one complete copy of the “shared” information must fit in each of the processes.

If we keep our grid size small enough, we can actually program our heat flow application using this approach. This is almost certainly a less efficient implementation than any of the earlier implementations of this problem because the core computation is so simple. However, if the core computations were more complex and needed access to values farther than one unit away, this might be a good approach.

The data structures are simpler for this approach and, actually, are no different than the single-process FORTRAN 90 or HPF versions. We will allocate a complete RED and BLACK array in every process:

```
PROGRAM MHEAT
INCLUDE 'mpif.h'
INCLUDE 'mpef.h'
INTEGER ROWS, COLS
PARAMETER(MAXTIME=200)
PARAMETER(ROWS=200, COLS=200)
DOUBLE PRECISION RED(0:ROWS+1, 0:COLS+1), BLACK(0:ROWS+1, 0:COLS+1)
```

We need fewer variables for the MPI calls because we aren't creating a communicator. We simply use the default communicator MPI\_COMM\_WORLD. We start up our processes, and find the size and rank of our process group:

```
INTEGER INUM, NPROC, IERR, SRC, DEST, TAG
INTEGER S, E, LS, LE, MYLEN
INTEGER STATUS(MPI_STATUS_SIZE)
INTEGER I, R, C
INTEGER TICK, MAXTIME
CHARACTER*30 FNAME

PRINT *, 'Calling MPI_INIT'
CALL MPI_INIT( IERR )
CALL MPI_COMM_SIZE( MPI_COMM_WORLD, NPROC, IERR )
CALL MPI_COMM_RANK( MPI_COMM_WORLD, INUM, IERR )
CALL MPE_DECOMP1D(COLS, NPROC, INUM, S, E, IERR)
PRINT *, 'My Share ', INUM, NPROC, S, E
```

Since we are broadcasting initial values to all of the processes, we only have to set things up on the master process:

```
* Start Cold

IF ( INUM.EQ.0 ) THEN
  DO C=0, COLS+1
    DO R=0, ROWS+1
      BLACK(R, C) = 0.0
    ENDDO
  ENDDO
ENDIF
```

As we run the time steps (again with no synchronization), we set the persistent heat sources directly. Since the shape of the data structure is the same in the master and all other processes, we can use the real array coordinates rather than mapping them as with the previous examples. We could skip the persistent settings on the nonmaster processes, but it doesn't hurt to do it on all processes:

```
* Begin running the time steps
  DO TICK=1,MAXTIME

* Set the heat sources
  BLACK(ROWS/3, COLS/3)= 10.0
  BLACK(2*ROWS/3, COLS/3) = 20.0
  BLACK(ROWS/3, 2*COLS/3) = -20.0
  BLACK(2*ROWS/3, 2*COLS/3) = 20.0
```

Now we broadcast the entire array from process rank zero to all of the other processes in the `MPI_COMM_WORLD` communicator. Note that this call does the sending on rank zero process and receiving on the other processes. The net result of this call is that all the processes have the values formerly in the master process in a single call:

```
* Broadcast the array
  CALL MPI_BCAST(BLACK, (ROWS+2)*(COLS+2), MPI_DOUBLE_PRECISION,
+               0, MPI_COMM_WORLD, IERR)
```

Now we perform the subset computation on each process. Note that we are using global coordinates because the array has the same shape on each of the processes. All we need to do is make sure we set up our particular strip of columns according to `S` and `E`:

```
* Perform the flow on our subset

  DO C=S,E
    DO R=1,ROWS
      RED(R,C) = ( BLACK(R,C) +
+                 BLACK(R,C-1) + BLACK(R-1,C) +
+                 BLACK(R+1,C) + BLACK(R,C+1) ) / 5.0
    ENDDO
  ENDDO
```

Now we need to gather the appropriate strips from the processes into the appropriate strip in the master array for rebroadcast in the next time step. We could change the loop in the master to receive the messages in any order and check the `STATUS` variable to see which strip it received:

```

* Gather back up into the BLACK array in master (INUM = 0)
  IF ( INUM .EQ. 0 ) THEN
    DO C=S,E
      DO R=1,ROWS
        BLACK(R,C) = RED(R,C)
      ENDDO
    ENDDO
    DO I=1,NPROC-1
      CALL MPE_DECOMP1D(COLS, NPROC, I, LS, LE, IERR)
      MYLEN = ( LE - LS ) + 1
      SRC = I TAG = 0
      CALL MPI_RECV(BLACK(0,LS),MYLEN*(ROWS+2),
+                 MPI_DOUBLE_PRECISION, SRC, TAG,
+                 MPI_COMM_WORLD, STATUS, IERR)
*       Print *,'Recv',I,MYLEN
    ENDDO
  ELSE
    MYLEN = ( E - S ) + 1
    DEST = 0
    TAG = 0
    CALL MPI_SEND(RED(0,S),MYLEN*(ROWS+2),MPI_DOUBLE_PRECISION,
+               DEST, TAG, MPI_COMM_WORLD, IERR)
*       Print *,'Send',INUM,MYLEN
  ENDIF
ENDDO

```

We use MPE\_DECOMP1D to determine which strip we're receiving from each process.

In some applications, the value that must be gathered is a sum or another single value. To accomplish this, you can use one of the MPI reduction routines that coalesce a set of distributed values into a single value using a single call.

Again at the end, we dump out the data for testing. However, since it has all been gathered back onto the master process, we only need to dump it on one process:

```

* Dump out data for verification
  IF ( INUM .EQ.0 .AND. ROWS .LE. 20 ) THEN
    FNAME = '/tmp/mheatout'
    OPEN(UNIT=9,NAME=FNAME,FORM='formatted')
    DO C=1,COLS
      WRITE(9,100)(BLACK(R,C),R=1,ROWS)
100     FORMAT(20F12.6)
    ENDDO
    CLOSE(UNIT=9)
  ENDIF

  CALL MPI_FINALIZE(IERR)

```

END

When this program executes with four processes, it produces the following output:

```
% mpif77 -c mheat.f
mheat.f:
MAIN mheat:
% mpif77 -o mheat mheat.o -lmpe
% mheat -np 4
Calling MPI_INIT
My Share 1 4 51 100
My Share 0 4 1 50
My Share 3 4 151 200
My Share 2 4 101 150
%
```

The ranks of the processes and the subsets of the computations for each process are shown in the output.

So that is a somewhat contrived example of the broadcast/gather approach to parallelizing an application. If the data structures are the right size and the amount of computation relative to communication is appropriate, this can be a very effective approach that may require the smallest number of code modifications compared to a single-processor version of the code.

## 5 MPI Summary

Whether you chose PVM or MPI depends on which library the vendor of your system prefers. Sometimes MPI is the better choice because it contains the newest features, such as support for hardware-supported multicast or broadcast, that can significantly improve the overall performance of a scatter-gather application.

A good text on MPI is *Using MPI — Portable Parallel Programming with the Message-Passing Interface*, by William Gropp, Ewing Lusk, and Anthony Skjellum (MIT Press). You may also want to retrieve and print the MPI specification from <http://www.netlib.org/mpi/><sup>5</sup>.

---

<sup>5</sup><http://www.netlib.org/mpi/>