Message Passing Basics

John Urbanic
urbanic@psc.edu

April 23, 2002

PITTSBURGH
SUPERCOMPUTING
CENTER

Introduction

What is MPI? The Message-Passing Interface Standard(MPI) is a library that allows you to do problems in parallel using message- passing to communicate between processes.

- Library
  It is not a language (like FORTRAN 90, C or HPF) or even an extension to a language. Instead, it is a library that your native, standard, serial compiler (f77, f90, cc, CC) uses.
- Message Passing
  Message passing is sometimes referred to as a paradigm itself. But it is really just a method of passing data between processes that is flexible enough to implement most paradigms (Data Parallel, Work Sharing, etc.) with it.
- Communicate
  This communication may be via a dedicated MPP torus network, or merely an office LAN. To the MPI programmer, it looks much the same.
- Processes
  These can be 512 PEs on a T3E, or 4 processes on a single workstation.

Basic MPI

In order to do parallel programming, you require some basic functionality, namely, the ability to:

- Start Processes
- Send Messages
- Receive Messages
- Synchronize

With these four capabilities, you can construct any program. We will look at the basic versions of the MPI routines that implement this. Of course, MPI offers over 125 functions. Many of these are more convenient and efficient for certain tasks. However, with what we learn here, we will be able to implement just about any algorithm. Moreover, the vast majority of MPI codes are built using primarily these routines.

Starting Processes on the T3E or TCS

On the T3E or TCS, the fundamental control of processes is fairly simple. There is always one process for each PE that your code is running on. At run time, you specify how many PEs you require and then your code is copied to each PE and run simultaneously. In other words, a 512 PE T3E or TCS code has 512 copies of the same code running on it from start to finish.

At first the idea that the same code must run on every node seems very limiting. We'll see in a bit that this is not at all the case.

Hello World: C Code

The easiest way to see exactly how a parallel code is put together and run is to write the classic "Hello World" program in parallel. In this case it simply means that every PE will say hello to us. Let's take a look at the code to do this.

Hello World C Code

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char** argv){

   int my_PE_num;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);
   printf("Hello from %d.\n", my_PE_num);
   MPI_Finalize();
}
```

Hello World: Fortran Code

```
program shifter
  include 'mpif.h'

integer my_pe_num, errcode

call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_pe_num, errcode)
print *, 'Hello from ', my_pe_num,'.'
call MPI_FINALIZE(errcode)
end
```

Output

```
Hello from 5.
Hello from 3.
Hello from 1.
Hello from 2.
Hello from 7.
Hello from 0.
Hello from 6.
Hello from 4.
```

There are two issues here that may not have been expected. The most obvious is that the output might seem out of order. The response to that is "what order were you expecting?" Remember, the code was started on all nodes practically simultaneously. There was no reason to expect one node to finish before another. Indeed, if we rerun the code we will probably get a different order. Sometimes it may seem that there is a very repeatable order. But, one important rule of parallel computing is don't assume that there is any particular order to events unless there is something to guarantee it. Later on we will see how we could force a particular order on this output.

Format of MPI Calls

The first thing to notice about these, or any, MPI codes is that the MPI header files,
in C: *"mpi.h"*
in Fortran: *'mpif.h'*
must be included. These contain all the MPI definitions you will ever need.

The next thing to note is the format of MPI calls:

- For Fortran, the general format is

```
Call MPI_XXXXX(parameter,..., ierror)
```

  Case is not important here. So, an equivalent form would be

```
call mpi_xxxxx(parameter,..., ierror)
```

  Instead of the function returning with an error code, as in C, the Fortran versions of MPI routines usually have one additional parameter in the calling list, ierror, which is the return code. Upon success, ierror is set to MPI_SUCCESS.

MPI_Init, MPI_Fin, MPI_Comm_rank

All MPI codes must start with MPI_Init before doing any MPI work. Likewise, they should all issue a MPI_Finalize when they are done.

Besides these most basic of MPI routines, you will also always wish to use the MPI_Comm_Rank routine to determine what the number of the PE the routine is running on is. This will always be from 0 to N-1 for N PEs.

Remember, this exact same code is running on each of the PEs. Unless you want the same codes to use the same data in exactly the same manner and generate exactly the same results on each node (which is kind of pointless), you will want to have the PEs vary their behavior based upon their PE number.

In this case, the number is merely used to have each PE print a slightly different message out. In general, though, the PE number will be used to load different data files or take different branches in the code.

MPI_Comm_rank

The extreme case of this is to have different PEs execute entirely different sections of code based upon their PE number.

```
if (my_PE_num = 0)
        Routine1
else if (my_PE_num = 1)
        Routine2
else if (my_PE_num =2)
        Routine3
                .
                .
                .
```

So, we can see that even though we have a logical limitation of having each PE execute the same program, for all practical purposes we can really have each PE running an entirely unrelated program by bundling them all into one executable and then calling them as separate routines based upon PE number.

Master and Slaves PEs

The much more common case is to have a single PE that is used for some sort of coordination purpose, and the other PEs run code that is the same, although the data will be different. This is how one would implement a master/slave or host/node paradigm.

```
if (my_PE_num = 0)
     MasterCodeRoutine
else
     SlaveCodeRoutine
```

Of course, the above code is the trivial case of

```
EveryBodyRunThisRoutine
```

and consequently the only difference will be in the output, as it actually uses the PE number.

MPI_COMM_WORLD

In the Hello World program, we see that the first parameter in MPI_Comm_rank (MPI_COMM_WORLD, &my_PE_num) is MPI_COMM_WORLD.
MPI_COMM_WORLD is known as the "communicator" and can be found in many of the MPI routines. In general, it is used so that one can divide up the PEs into subsets for various algorithmic purposes. For example, if we had an array that we wished to find the determinant of distributed across the PEs, we might wish to define some subset of the PEs that holds a certain column of the array so that we could address only those PEs conveniently.

However, this is a convenience that can often be dispensed with. As such, one will often see the value MPI_COMM_WORLD used anywhere that a communicator is required. This is simply the global set that states we don't really care to deal with any particular subset here.

Compiling and Running

Well, now that we may have some idea how the above code will perform, let's compile it and run it to see if it meets our expectations. We compile using a normal ANSI C or Fortran 90 compiler (C++ is also available): While logged in the T3E (jaromir.psc.edu)

For C codes:

```
cc -lmpi hello.c
```

For Fortran codes:

```
f90 -lmpi hello.c
```

We now have an executable. To run on the T3E we must tell the machine how many copies we wish to run. In the T3E, you can choose any number. We'll try 8:

On the T3E we use `mpprun -n8 a.out`

On the TCS we use `prun -n8 a.out`

Where Will The Output Go?

The second issue, although you may have taken it for granted, is

"where will the output go?".

This is another question that MPI dodges because it is so implementation dependent.

On the T3E, the I/O is structured in about the simplest way possible. All PEs can read and write (files as well as console I/O) through the standard channels. This is very convenient, and in our case results in all of the "standard output" going back to your terminal window on the T3E. The TCS is very similar.

In general, it can be much more complex. For instance, suppose you were running this on a cluster of 8 workstations. Would the output go to eight separate consoles? Or, in a more typical situation, suppose you wished to write results out to a file:

With the workstations, you would probably end up with eight separate files on eight separate disks.

With the T3E, they can all access the same file simultaneously.

There are some good reasons why you would want to exercise some constraint even on the T3E. 512 PEs accessing the same file would be extremely inefficient.

Sending and Receiving Messages

Hello world might be illustrative, but we haven't really done any message passing yet.

Let's write the simplest possible message passing program.

It will run on 2 PEs and will send a simple message (the number 42) from PE 1 to PE 0. PE 0 will then print this out.

Sending a Message

Sending a message is a simple procedure. In our case the routine will look like this in C (the standard man pages are in C, so you should get used to seeing this format):

```
MPI_Send( &numbertosend, 1, MPI_INT, 0, 10, MPI_COMM_WORLD)
```

Sending a Message Cont'd

Let's look at the parameters individually:

&numbertosend	a pointer to whatever we wish to send. In this case it is simply an integer. It could be anything from a character string to a column of an array or a structure. It is even possible to pack several different data types in one message.
1	the number of items we wish to send. If we were sending a vector of 10 int's, we would point to the first one in the above parameter and set this to the size of the array.
MPI_INT	the type of object we are sending. Possible values are: MPI_CHAR, MPI_SHORT, MPI_INT, MPI_LONG, MPI_UNSIGNED_CHAR, MPI_UNSIGNED_SHORT, MPI_UNSIGNED, MPI_UNSIGNED_LING, MPI_FLOAT, MPI_DOUBLE, MPI_LONG_DOUBLE, MPI_BYTE, MPI_PACKED Most of these are obvious in use. MPI_BYTE will send raw bytes (on a heterogeneous workstation cluster this will suppress any data conversion). MPI_PACKED can be used to pack multiple data types in one message, but it does require a few additional routines we won't go into (those of you familiar with PVM will recognize this).
0	Destination of the message. In this case PE 0.
10	Message tag. All messages have a tag attached to them that can be useful for sorting messages. For example, one could give high priority control messages a different tag then data messages. When receiving, the program would check for messages that use the control tag first. We just picked 10 at random.
MPI_COMM_WORLD	We don't really care about any subsets of PEs here. So, we just chose this "default".

Receiving a Message

Receiving a message is equally simple. In our case it will look like:

MPI_Recv(&numbertoreceive, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG,MPI_COMM_WORLD, &status)	
&numbertoreceive	A pointer to the variable that will receive the item. In our case it is simply an integer that has has some undefined value until now.
1	Number of items to receive. Just 1 here.
MPI_INT	Datatype. Better be an int, since that's what we sent.
MPI_ANY_SOURCE	The node to receive from. We could use 1 here since the message is coming from there, but we'll illustrate the "wild card" method of receiving a message from anywhere.
MPI_ANY_TAG	We could use a value of 10 here to filter out any other messages (there aren't any) but, again, this was a convenient place to show how to receive any tag.
MPI_COMM_WORLD	Just using default set of all PEs.
&status	A structure that receive the status data which includes the source and tag of the message.

Send and Receive C Code

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char** argv){

      int my_PE_num, numbertoreceive, numbertosend=42;
      MPI_Status status;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);

if (my_PE_num==0){
      MPI_Recv( &numbertoreceive, 1, MPI_INT, MPI_ANY_SOURCE,
                MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      printf("Number received is: %d\n", numbertoreceive);
}
else MPI_Send( &numbertosend, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);

MPI_Finalize(); }
```

Send and Receive Fortran Code

```
program shifter
implicit none
include 'mpif.h'

integer my_pe_num, errcode, numbertoreceive, numbertosend integer
status(MPI_STATUS_SIZE)

call MPI_INIT(errcode)
call MPI_COMM_RANK(MPI_COMM_WORLD, my_pe_num, errcode)

numbertosend = 42

if (my_PE_num.EQ.0) then
    call MPI_Recv( numbertoreceive, 1, MPI_INTEGER,MPI_ANY_SOURCE,
                   MPI_ANY_TAG, MPI_COMM_WORLD, status, errcode)

    print *, 'Number received is:‘     ,numbertoreceive
endif
```

```
if (my_PE_num.EQ.1) then
     call MPI_Send( numbertosend, 1,MPI_INTEGER, 0, 10, MPI_COMM_WORLD,
                           errcode)
endif

call MPI_FINALIZE(errcode)

end
```

Non-Blocking Recieves

All of the receives that we will use are blocking. This means that they will wait until a message matching their requirements for source and tag has been received. It is possible to use non-blocking communications. This means a receive will return immediately and it is up to the code to determine when the data actually arrives using additional routines.

In most cases this additional coding is not worth it in terms of performance and code robustness. However, for certain algorithms this can be useful to keep in mind.

Communication Modes

There are four possible modes (with slight differently named MPI_XSEND routines) for buffering and sending messages in MPI. We use the standard mode here, and you may find this sufficient for the majority of your needs. However, these other modes can allow for substantial optimization in the right circumstances:

Standard mode	Send will usually not block even if a receive for that message has not occurred. Exception is if there are resource limitations (buffer space).
Buffered Mode	Similar to above, but will never block (just return error).
Synchronous Mode	will only return when matching receive has started.
Ready Mode	will only work if matching receive is already waiting.

Synchronization

We are going to write one more code which will employ the remaining tool that we need for general parallel programming: synchronization. Many algorithms require that you be able to get all of the nodes into some controlled state before proceeding to the next stage. This is usually done with a synchronization point that require all of the nodes (or some specified subset at the least) to reach a certain point before proceeding. Sometimes the manner in which messages block will achieve this same result implicitly, but it is often necessary to explicitly do this and debugging is often greatly aided by the insertion of synchronization points which are later removed for the sake of efficiency.

Our code will perform the rather pointless operation of having PE 0 send a number to the other 3 PEs and have them multiply that number by their own PE number. They will then print the results out (in order, remember the hello world program?) and send them back to PE 0 which will print out the sum.

Synchronization: C Code

```
#include <stdio.h>
#include "mpi.h"

main(int argc, char** argv){

    int my_PE_num, numbertoreceive, numbertosend=4,index, result=0;
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_PE_num);

    if (my_PE_num==0)
        for (index=1; index<4; index++)
          MPI_Send( &numbertosend, 1,MPI_INT, index, 10,MPI_COMM_WORLD);
    else{
        MPI_Recv( &numbertoreceive, 1, MPI_INT, 0, 10, MPI_COMM_WORLD,
                    &status);
        result = numbertoreceive * my_PE_num;
    }
```

```
    for (index=1; index<4; index++){
        MPI_Barrier(MPI_COMM_WORLD);
        if (index==my_PE_num)
                    printf("PE %d's result is %d.\n", my_PE_num, result);
    }

    if (my_PE_num==0){
        for (index=1; index<4; index++){
            MPI_Recv( &numbertoreceive, 1,MPI_INT,index,10, MPI_COMM_WORLD,
                    &status);
            result += numbertoreceive;
        }
        printf("Total is %d.\n", result);
    }
    else
        MPI_Send( &result, 1, MPI_INT, 0, 10, MPI_COMM_WORLD);

    MPI_Finalize();
}
```

Synchronization: Fortran Code

```
program shifter
implicit none

include 'mpif.h'

integer my_pe_num, errcode, numbertoreceive, numbertosend
integer index, result
integer status(MPI_STATUS_SIZE)

call MPI_INIT(errcode)

call MPI_COMM_RANK(MPI_COMM_WORLD, my_pe_num, errcode)

numbertosend = 4
result = 0

if (my_PE_num.EQ.0) then
     do index=1,3
         call MPI_Send( numbertosend, 1, MPI_INTEGER, index, 10, MPI_COMM_WORLD, errcode)
     enddo
 else
     call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, status, errcode)
     result = numbertoreceive * my_PE_num
 endif
```

```
do index=1,3
     call MPI_Barrier(MPI_COMM_WORLD, errcode)
     if (my_PE_num.EQ.index) then
          print *, 'PE ',my_PE_num,'s result is ',result,'.'
     endif
enddo

if (my_PE_num.EQ.0) then
     do index=1,3
         call MPI_Recv( numbertoreceive, 1, MPI_INTEGER, index,10, MPI_COMM_WORLD, status, errcode)
         result = result + numbertoreceive
     enddo
     print *,'Total is ',result,'.'
 else
     call MPI_Send( result, 1, MPI_INTEGER, 0, 10, MPI_COMM_WORLD, errcode)
 endif

call MPI_FINALIZE(errcode)
end
```

Results of "Synchronization"

The output you get when running this codes with 4 PEs (what will happen if you run with more or less?) is the following:

PE 1's result is 4.

PE 2's result is 8.

PE 3's result is 12.

Total is 24

Analysis of "Synchronization"

The best way to make sure that you understand what is happening in the code above is to look at things from the perspective of each PE in turn. THIS IS THE WAY TO DEBUG ANY MESSAGE-PASSING (or MIMD) CODE.

Follow from the top to the bottom of the code as PE 0, and do likewise for PE 1. See exactly where one PE is dependent on another to proceed. Look at each PEs progress as though it is 100 times faster or slower than the other nodes. Would this affect the final program flow? It shouldn't unless you made assumptions that are not always valid.

Reduction

MPI_Reduce: Reduces values on all processes to a single value.

Synopsis

```
#include "mpi.h"
int MPI_Reduce ( sendbuf, recvbuf, count, datatype, op, root, comm )
void *sendbuf;
void *recvbuf;
int count;
MPI_Datatype datatype;
MPI_Op op;
int root;
MPI_Comm comm;
```

Reduction Cont'd

Input Parameters:

sendbuf	address of send buffer
count	number of elements in send buffer (integer)
datatype	data type of elements of send buffer (handle)
op	reduce operation (handle)
root	rank of root process (integer)
comm	communicator (handle)

Output Parameter:

recvbuf	address of receive buffer (choice, significant only at root)

Algorithm: This implementation currently uses a simple tree algorithm.

Finding Pi

Our last example will find the value of pi by integrating $4/(1 + x^2)$ for -1/2 to +1/2.

This is just a geometric circle. The master process (0) will query for a number of intervals to use, and then broadcast this number to all of the other processors.

Each processor will then add up every n^{th} interval (x = -1/2 + rank/n, -1/2 + rank/n + size/n).

Finally, the sums computed by each processor are added together using a new type of MPI operation, a reduction.

Finding Pi

```
program FindPI
implicit none

include 'mpif.h'
integer n, my_pe_num, numprocs, index, errcode
real mypi, pi, h sum, x

call MPI_Init(errcode)
call MPI_Comm_size(MPI_COMM_WORLD, numprocs, errcode)
call MPI_Comm_rank(MPI_COMM_WORLD, my_pe_num, errcode)

if (my_pe_num.EQ.0) then
    print *,'How many intervals?:'
    read *, n
endif

call MPI_Bcast(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, errcode)
```

```
h = 1.0 / n
sum = 0.0

do index = my_pe_num+1, n, numprocs
      x = h * (index - 0.5)
      sum = sum + 4.0 / (1.0 + x*x)
enddo
mypi = h * sum

call MPI_Reduce(mypi, pi, 1, MPI_REAL, MPI_SUM, 0,  MPI_COMM_WORLD, errcode)

if (my_pe_num.EQ.0) then
      print *,'pi is approximately ',pi
      print *,'Error is ',pi-3.14159265358979323846
endif

call MPI_Finalize(errcode)
end
```

Do Not Make Any Assumptions

Do not make any assumptions about the mechanics of the actual message- passing. Remember that MPI is designed to operate not only on fast MPP networks, but also on Internet size meta-computers. As such, the order and timing of messages may be considerably skewed.

MPI makes only one guarantee: two messages sent from one process to another process will arrive in that relative order. However, a message sent later from another process may arrive before, or between, those two messages.

What We Did Not Cover

Obviously, we have only touched upon the 120+ MPI routines. Still, you should now have a solid understanding of what message-passing is all about, and (with manual in hand) you will have no problem reading the majority of well-written codes. The best way to gain a more complete knowledge of what is available is to leaf through the manual and get an idea of what is available. Some of the more useful functionalities that we have just barely touched upon are:

- Communicators
  - We have used only the "world" communicator in our examples. Often, this is exactly what you want. However, there are times when the ability to partition your PEs into subsets is convenient, and possibly more efficient. In order to provide a considerable amount of flexibility, as well as several abstract models to work with, the MPI standard has incorporated a fair amount of detail that you will want to read about in the Standard before using this.
- Varieties of MPI
  - There are several implementations of MPI, each of which supports a wide variety of platforms. You can find two of these at PSC, the EPCC version and the MPICH version. Cray will has a proprietary version of their own as does Compaq. Please note that all of these are based upon the official MPI standard.
- MPI I/O
  - These are some new routines to facilitate I/O in parallel codes. They have many performance pitfalls and you should discuss use of them with someone familiar with the I/O system of your particular platform before investing much effort into them.
- User Defined Data Types
  - MPI provides the ability to define your own message types in a convenient fashion. If you find yourself wishing that there were such a feature for your own code, it is there.

What We Did Not Cover Cont'd

- Related to this are the "gather" routines. These are in some sense the inverse of the gather routines.
- Communicators We have used only the "world" communicator in our examples. Often, this is exactly what you want. However, there are times when the ability to partition your PEs into subsets is convenient, and possibly more efficient. In order to provide a considerable amount of flexibility, as well as several abstract models to work with, the MPI standard has incorporated a fair amount of detail that you will want to read about in the Standard before using this.
- Varieties of MPI There are several implementations of MPI, each of which supports a wide variety of platforms. You can find two of these at PSC, the EPCC version that we compiled with, and the MPICH version. Cray will soon have a proprietary version of their own. Please note that all of these are based upon the official MPI standard.

References

There is a wide variety of material available on the Web, some of which is intended to be used as hardcopy manuals and tutorials. Besides our own local docs at

http://www.psc.edu/htbin/software_by_category.pl/hetero_software

you may wish to start at one of the MPI home pages at

http://www.mcs.anl.gov/Projects/mpi/index.html

from which you can find a lot of useful information without traveling too far. To learn the syntax of MPI calls, access the index for the Message Passing Interface Standard at:

http://www-unix.mcs.anl.gov/mpi/www/

Books:

- *Parallel Programming with MPI*. Peter S. Pacheco. San Francisco: Morgan Kaufmann Publishers, Inc., 1997.
- *PVM: a users' guide and tutorial for networked parallel computing*. Al Geist, Adam Beguelin, Jack Dongarra et al. MIT Press, 1996.
- *Using MPI: portable parallel programming with the message-passing interface*. William Gropp, Ewing Lusk, Anthony Skjellum. MIT Press, 1996.

Exercise

LIST OF MPI CALLS:

To view a list of all MPI calls, with syntax and descriptions, access the Message Passing Interface Standard at:

http://www-unix.mcs.anl.gov/mpi/www/

Exercise 1: Write a code that runs on 8 PEs and does a "circular shift." This means that every PE sends some data to its nearest neighbor either "up" (one PE higher) or "down." To make it circular, PE 7 and PE 0 are treated as neighbors. Make sure that whatever data you send is received.

Exercise 2: Write, using only the routines that we have covered in the first three examples, (MPI_Init, MPI_Comm_Rank, MPI_Send, MPI_Recv, MPI_Barrier, MPI_Finalize) a program that determines how many PEs it is running on. It should perform as the following:

```
mpprun -n4 exercise
I am running on 4 PEs.

mpprun -n16 exercise
I am running on 16 PEs.
```

Exercise

The solution may not be as simple as it first seems. Remember, make no assumptions about when any given message may be received. You would normally obtain this information with the simple `MPI_Comm_size()` routine.