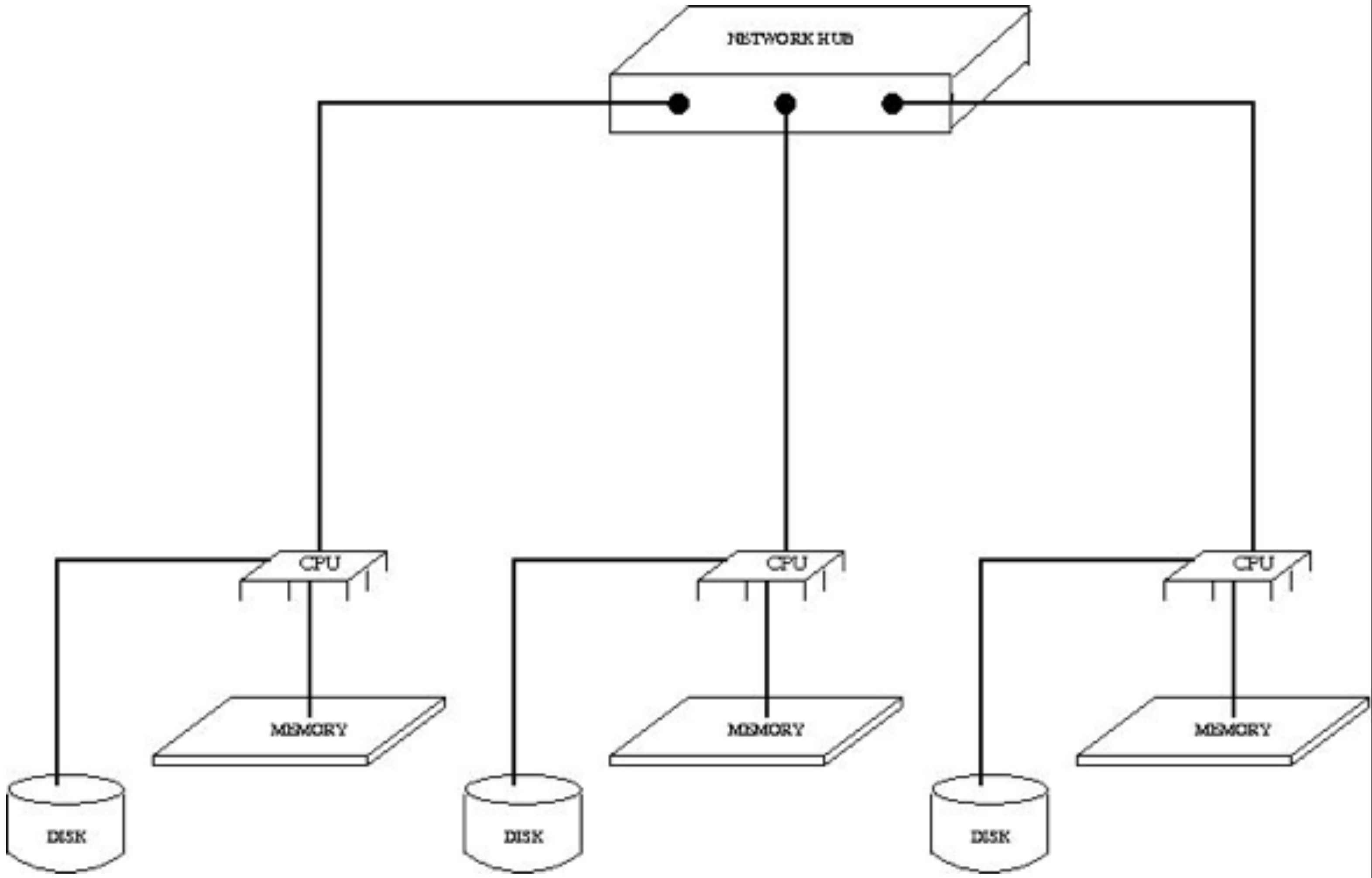
Message Passing Basics

Distributed – Memory Machines

- Each node in the computer has a locally addressable memory space
- The computers are connected together via some high-speed network
  - Infiniband, Myrinet, Giganet, etc..

- Pros
  - Really large machines
  - Size limited only by gross physical considerations:
    - Room size
    - Cable lengths (10’s of meters)
    - Power/cooling capacity
    - Money!
  - Cheaper to build and run
- Cons
  - Harder to program

    Data Locality



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, UPC 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.

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.

Hello World: C Code

How complicated is the code to do this? Not very:

```
#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();
}
```

First Example (Starting Processes): Hello World

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. Something like this:

```
mpirun –np 8 a.out
Hello from 0.
Hello from 1.
Hello from 2.
Hello from 3.
Hello from 4.
Hello from 5.
Hello from 6.
Hello from 7.
```

Hello World: Fortran Code

Here is the Fortran version:

```
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
```

We will make an effort to present both languages here, but they are really quite trivially similar in these simple examples, so try to play along on both.

Hello World: Fortran Code

Let's make a few general observations about how things look before we go into what is actually happening here.

We have to include the header file, either mpif.h or mpi.h.

The MPI calls are easy to spot, they always start with MPI_. Note that the MPI calls themselves are the same for both languages except that the Fortran routines have an added argument on the end to return the error condition, whereas the C ones return it as the function value. We should check these (for MPI_SUCCESS) in both cases as it can be very useful for debugging. We don't in these examples for clarity. You probably won't because of laziness.

```
#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();
}
```

```
   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
```

MPI_INIT, MPI_FINALIZE and MPI_COMM_RANK

OK, lets look at the actual MPI routines. All three of the ones we have here are very basic and will appear in any MPI code.

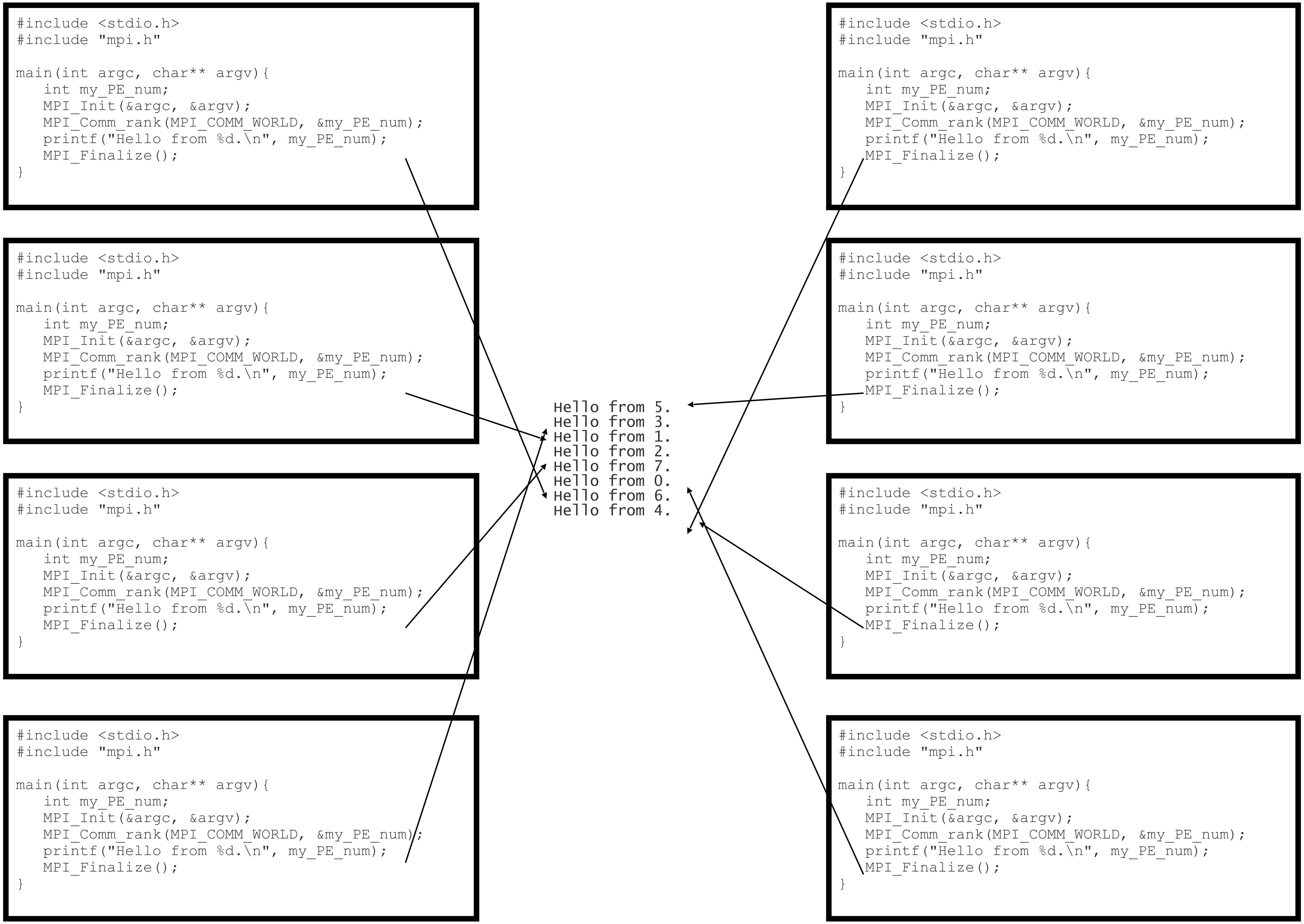
```
#include <stdio.h>
#include "mpi.h"

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();
}
```

MPI_INIT
This routine must be the first MPI routine you call (it certainly does not have to be the first statement). It sets things up and might do a lot on some cluster-type systems (like start daemons and such). On most dedicated MPPs, it won't do much. We just have to have it. In C, it requires us to pass along the command line arguments. These are very standard C variables that contain anything entered on the command line when the executable was run. You may have used them before in normal serial codes. You may also have never used them at all. In either case, if you just cut and paste them into the MPI_INIT, all will be well.

MPI_FINALIZE
This is the companion to MPI_Init. It must be the last MPI_Call. It may do a lot of housekeeping, or it may not. Your code won't know or care.

MPI_COMM_RANK
Now we get a little more interesting. This routine returns to every PE its rank, or unique address from 0 to PEs-1. This is the only thing that sets each PE apart from its companions. 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. There is also another argument, the communicator, that we will ignore for a few minutes.

What Actually Happened…



```
#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();
}
```

```
#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();
}
```

```
#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();
}
```

```
#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();
}
```

```
#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();
}
```

```
#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();
}
```

```
#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();
}
```

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

What Actually Happened…

```
#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 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 seems 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.

The second question you might ask is “how does the output know where to go?” A good question. In the case of a cluster, it isn’t at all clear that a bunch of separate unix boxes printing to standard out will somehow combine them all on one terminal. Indeed, you should appreciate that a dedicated MPP environment will automatically do this for you – even so you should expect a lot of buffering (hint: use flush if you must). Of course most “serious” IO is file-based and will depend upon a distributed file system (you hope).

Do all nodes really run the *same* code?

Yes, they do run the same code independently. You might think this is a serious constraint on getting each PE to do unique work. Not at all. They can use their PE numbers to diverge in behavior as much as they like.

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)
    Routine_SpaceInvaders
else if (my_PE_num = 1)
    Routine_CrackPasswords
else if (my_PE_num =2)
    Routine_WeatherForecast
        .
        .
        .
```

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 Hello World code is the trivial case of

```
    EveryBodyRunThisRoutine
```

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

Communicators

The last little detail in Hello World is the first parameter in

```
MPI_Comm_rank (MPI_COMM_WORLD, &my_PE_num)
```

This parameter 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 - distributed across the PEs - that we wished to find the determinant of, we might wish to define some subset of the PEs that holds a certain column of the array so that we could address only that column conveniently. Or, we might wish to define a communicator for just the odd PEs. Or just the top one fifth…you get the idea.

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 and states we don't really care to deal with any particular subset here. We will use it in all of our examples.

Second Example: Sending and Receiving Messages

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

Let's write about 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)
```

&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 and very symmetric (hint: cut and paste is your friend here). 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
```