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
```