High Performance Scientific computing
Lecture 6

S. Gopalakrishnan

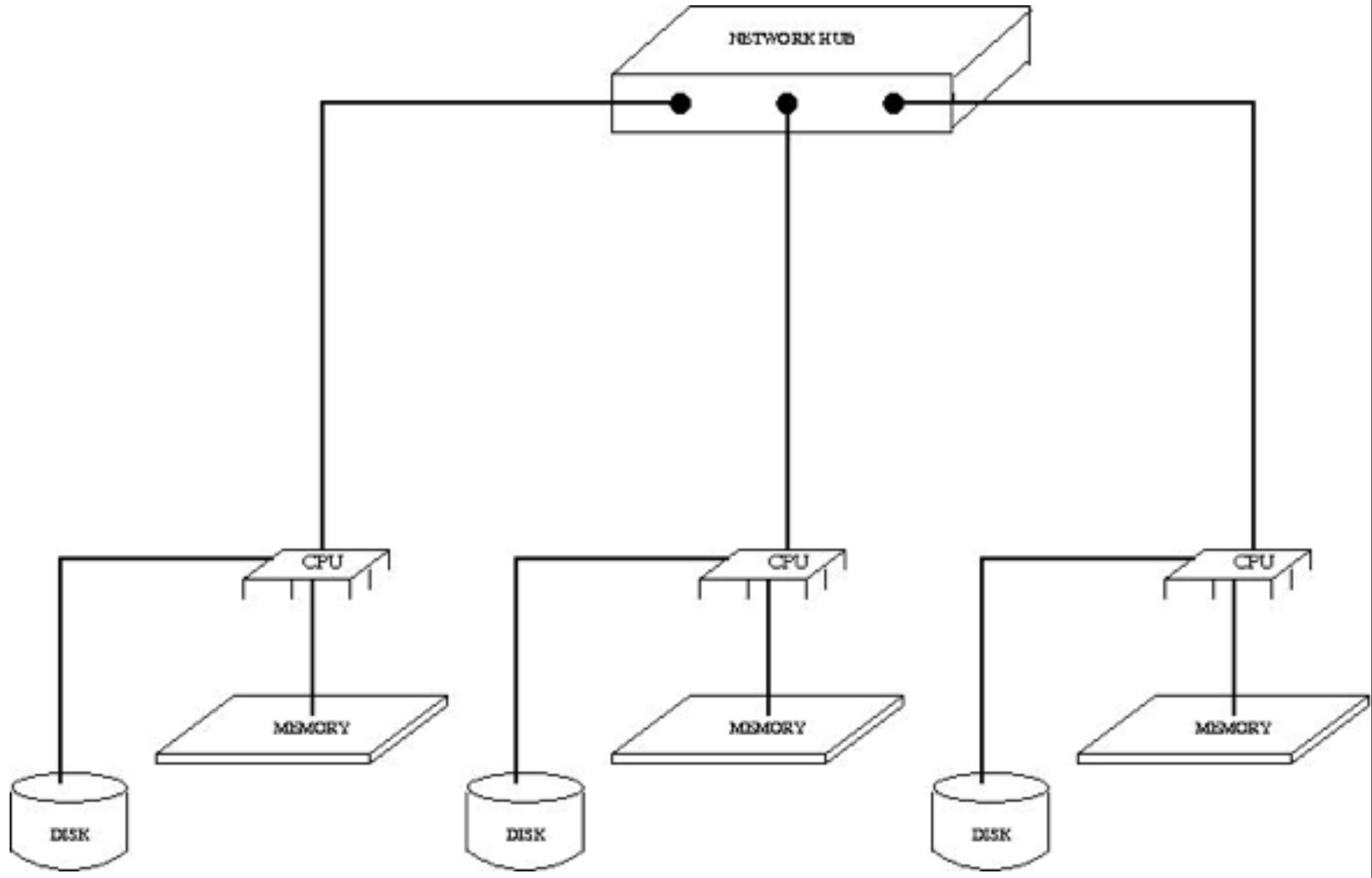
Message Passing Basics

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.

•**Processes**
These can be 88,128 PEs on Kei Computer, 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.

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: 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();
}
```

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.

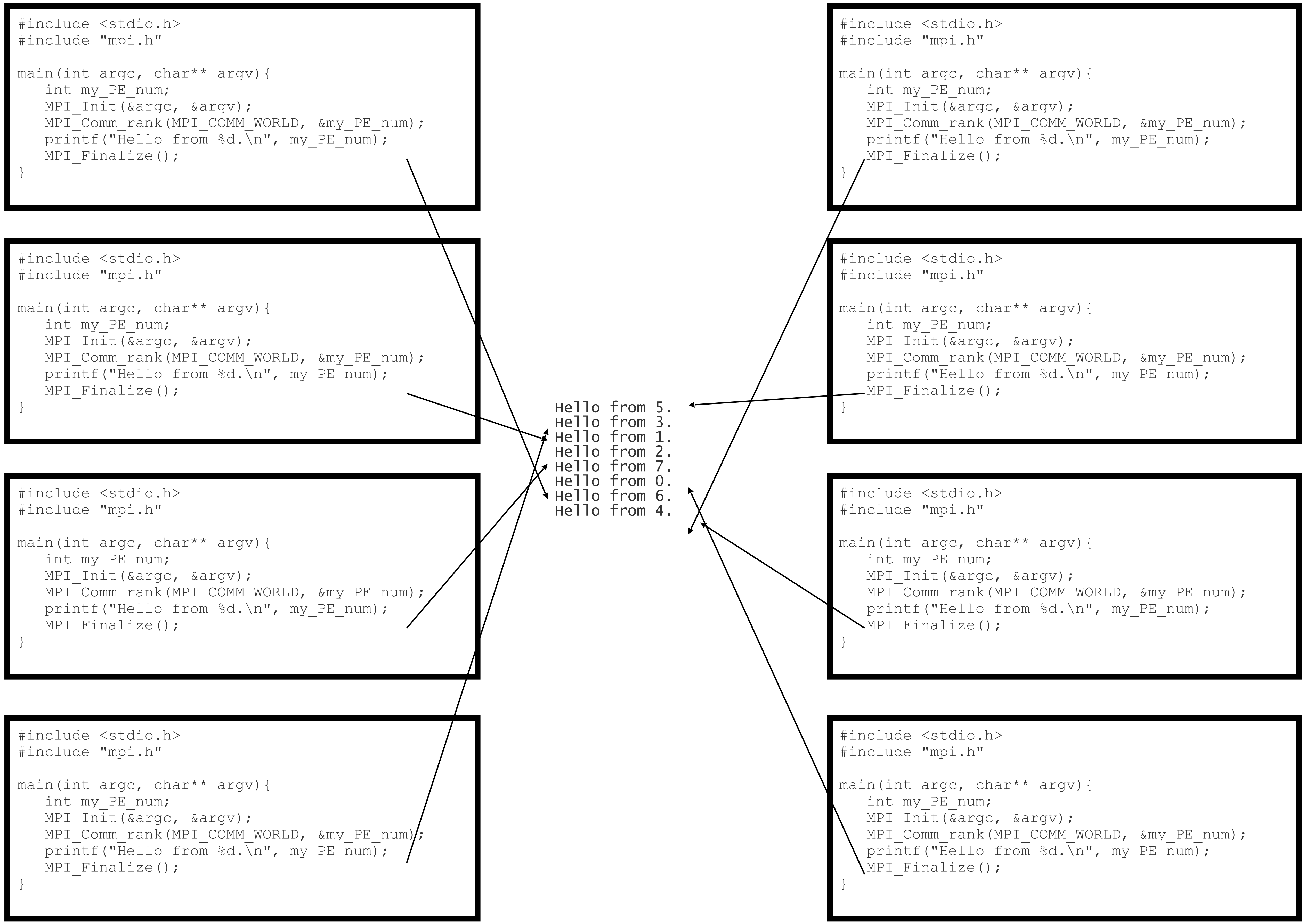
MPI_FINALIZE

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();
}
```

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

```
#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();
}
```

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.