1. **Consider the following program, called mpi1.c. This program is written in C with MPI commands included. It initializes MPI, executes a single print statement, then Finalizes (Quits) MPI.**

```
#include < mpi.h>   /* PROVIDES THE BASIC MPI DEFINITION AND TYPES */
#include < stdio.h>

int main(int argc, char **argv) {

  MPI_Init(&argc, &argv); /*START MPI */
  printf("Hello world\n");
  MPI_Finalize();  /* EXIT MPI */
}
```

To Compile

```
mpicc -o myprog myprog.c
```

To Run

```
mpirun -np 4 ./myprog
```

In the above example, the code will execute on four processors (-np 4). Output printed to the screen will look like:

```
Hello world
Hello world
Hello world
Hello world
```

Discussion: The four processors each perform the exact same task. Each processor prints a single line. MPI_Init and MPI_Finalize take care of all of the details associated with running the code on multiple processors. MPI_Init must be called before any other MPI command. MPI_Finalize must be called last. So, these two commands form a wrapper around the body of the parallel code.

2) Most parallel codes assign different tasks to different processors. For example, parts of an input data set might be divided and processed by different processors, or a finite difference grid might be divided among the processors available. This means that the code needs to identify processors. In this example, processors are identified by rank - an integer from 0 to total number of processors - 1.

```
#include < mpi.h>   /* PROVIDES THE BASIC MPI DEFINITION AND TYPES */
#include  < stdio.h>

int main(int argc, char **argv) {

  int my_rank;
  int size;
  MPI_Init(&argc, &argv); /*START MPI */
```

```
/*DETERMINE RANK OF THIS PROCESSOR*/
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

 /*DETERMINE TOTAL NUMBER OF PROCESSORS*/
  MPI_Comm_size(MPI_COMM_WORLD, &size);


  printf("Hello world! I'm rank (processor number) %d of size %d\n", my_rank, size);

  MPI_Finalize();  /* EXIT MPI */

}
```

Compiling and executing the code using 4 processors yields this result:

```
Hello world! I'm rank 0 of size 4
Hello world! I'm rank 2 of size 4
Hello world! I'm rank 1 of size 4
Hello world! I'm rank 3 of size 4
```

Discussion: Two additional MPI commands appear in this code: MPI_Comm_rank(,) and MPI_Comm_size(,). MPI_Comm_rank(,) returns the processor id assigned to each processor during start-up. This rank is returned as an integer (in this case called *my_rank*). The first parameter (MPI_COMM_WORLD) is predefined in MPI, and includes information about all the processors started when the program execution begins. Similarly, MPI_Comm_size returns the total number of processors - in this case 4. It is important to know the total number of processors so the problem set (data for example) can be divided among all available. Note that the code does not print the output in any particular order. In this case, rank 2 prints before rank 1. This order changes depending on the vagaries of communication between the nodes. Additional MPI commands are needed to structure communication more effectively.

3. **Two additional MPI commands may be used to direct traffic (message queuing) during the program execution: MPI_Send() and MPI_Recv(). It is safe to say these two commands are at the heart of MPI. Use of these statements makes the program appear more complicated, but it is well worth it if the flow of the program needs to be controlled.**

```
#include < mpi.h>   /* PROVIDES THE BASIC MPI DEFINITION AND TYPES */
#include < stdio.h>

int main(int argc, char **argv) {

  int my_rank;
  int partner;
  int size, i,t;
  char greeting[100];
  MPI_Status stat;

  MPI_Init(&argc, &argv); /*START MPI */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); /*DETERMINE RANK OF THIS PROCESSOR*/
```

```
  MPI_Comm_size(MPI_COMM_WORLD, &size); /*DETERMINE TOTAL NUMBER OF PROCESSORS*/


  sprintf(greeting, "Hello world: processor %d of %d\n", my_rank, size);

  /* adding a silly conditional statement like the
     following graphically illustrates "blocking" and
     flow control during program execution

     if (my_rank == 1) for (i=0; i<1000000000; i++) t=i;

  */

  if (my_rank ==0) {
    fputs(greeting, stdout);
    for (partner = 1; partner < size; partner++){

      MPI_Recv(greeting, sizeof(greeting), MPI_BYTE, partner, 1, MPI_COMM_WORLD, &stat);
      fputs (greeting, stdout);

    }
  }
  else {
    MPI_Send(greeting, strlen(greeting)+1, MPI_BYTE, 0,1,MPI_COMM_WORLD);
  }



  if (my_rank == 0) printf("That is all for now!\n");
  MPI_Finalize();  /* EXIT MPI */

}
```

Output from this code looks like:

```
Hello world: processor 0 of 4
Hello world: processor 1 of 4
Hello world: processor 2 of 4
Hello world: processor 3 of 4
That is all for now!
```

Discussion: The structure of the program has been changed to assure that the output is in the proper order (the processors are now listed in ascending order). Furthermore, the statement "That is all for now!" prints last. Program execution was effectively blocked until all processors had the opportunity to print. This was done by communicating between processors. Rather than simply printing, most processors now send the greeting back to processor 0 - usually called the root of the master processor. This processor first prints its own greeting, then polls successive processors - waiting to receive a message from each one. Only when the message is received does proccessor 0 move on. Using the MPI_Send and MPI_Recv commands blocks program execution. This blocking is illustrated graphically by inserting a long loop in the code, causing one of the processors to take a long time to complete its tasks. The cost of this structure is added syntax.

Here is the syntax for MPI_Send and MPI_Recv:

```
int MPI_Send (
                      message ,    /* actual information sent */
                      length,         /* length of the message */
                      datatype,   /* MPI datatype of the message */
                      destination, /* rank of the processor getting the message */
                      tag,             /* tag helps sort messages - likely an int and
the same as in MPI_Recv
                      MPI_Comm  /* almost alway MPI_Comm_World
                      )

int MPI_Recv (
                      message ,     /* actual information received*/
                      length,          /* length of the message */
                      datatype,     /* MPI datatype of the message */
                      source,         /* rank of the processor sending the message */
                      tag,               /* tag helps sort messages - likely an int and
the same as in MPI_Recv* /
                      MPI_Comm,  /* almost alway MPI_Comm_World */
                      Status           /* a data structure that contains info on what
was recv'd */
                      )
```

```
/*************************************************************
This is a simple send/receive program in MPI
*************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <math.h>

int main(argc,argv)
int argc;
char *argv[];
{
    int myid, numprocs;
    int tag,source,destination,count;
    int buffer;
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    tag=1;
    source=0;
    destination=1;
    count=1;
    if(myid == source){
      buffer=100;
```

```
      MPI_Send(&buffer,count,MPI_INT,destination,tag,MPI_COMM_WORLD);
      printf("processor %d  sent %d\n",myid,buffer);
    }
    if(myid == destination){
        MPI_Recv(&buffer,count,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
        printf("processor %d  got %d\n",myid,buffer);
    }
    MPI_Finalize();
}
```