MA/CSSE Homework 4
Due 3/31

Directions

- **Each problem must be self-contained in a file named** `problem_<problemnum>.c` **and must compile using the command** `mpicc problem_<problemnum>.c`

- Submit your files to the Moodle dropbox, and hand in a printout containing a screenshot for each problem showing your code compiling and running as expected.

- If your code does not run perfectly, you may include a `README.txt` file in the dropbox to state the status of your code (what works, what doesn't, what you think the problem is, etc.)

Problems:

1. Write a program that prompts the user for an integer value and distributes the value to all of the MPI processes. Each process should print out its rank and the value it received. Values should be read until a negative integer is given as input.

   Example output is

```
Input: 114
Rank: 2 Received: 114
Rank: 11 Received: 114
.
.
.
Input: 3
Rank: 5 Received: 3
Rank: 2 Received: 3
.
.
.
Input: -1
```

2. Write a program that takes data from process zero and sends it to all of the other processes by sending it in a ring. That is, process i should receive the data and send it to process i+1, until the last process is reached.

Assume that the data consists of a single integer. The master process reads the data from the user.

Each process should print out its rank, the data it received, and where the data came from when it gets the data. For example, the process with rank `5` should print

```
Rank: 5 Sender: 4 Data: 113
```

3. The MPI function `MPI_Wtime()` determines the number of seconds since some predetermined time. You can measure elapsed wall-clock time for some code by using

```
 double starttime, endtime;
       starttime = MPI_Wtime();
        ....  stuff to be timed  ...
       endtime   = MPI_Wtime();
       printf("That took %f seconds\n",endtime-starttime);
```

The accuracy of the `MPI_Wtime` command can be determined using `MPI_Wtick()`, which returns the accuracy of the `MPI_Wtime` command in seconds (a double again).

We can determine the communication speed between process of rank a and rank b by sending a random message of length N from process a to b, from b to a, and calculating $2N/T$ where T is the round trip time.

Write a program that takes an even number of processes to use, P, a message size in bytes, B, a number of experiments to run, E. The program should exit *gracefully* if P is not even. The program should randomly pair the processes, and test the communication speed between the processes using described above and average the outcome of all E experiments.

Sample input is:

```
./problem_4 4 1024 10
```

Sample output is

```
Data Size: 1024B
Number of Samples: 10
Communication Speed between rank 1 and 3 was 10.4 MB/s
Communication Speed between rank 2 and 0 was 10.1 MB/s

Here is the machine-process pairing

Rank 0: mpi-01.csse.rose-hulman.edu
Rank 1: mpi-01.csse.rose-hulman.edu
Rank 2: mpi-03.csse.rose-hulman.edu
Rank 3: mpi-03.csse.rose-hulman.edu
```

Another sample output, for the same input, is Sample output is

```
Data Size: 1024B
Number of Samples: 10
Communication Speed between rank 0 and 3 was 9.4 MB/s
Communication Speed between rank 1 and 0 was 11.3 MB/s

Here is the machine-process pairing

Rank 0: mpi-01.csse.rose-hulman.edu
Rank 1: mpi-01.csse.rose-hulman.edu
Rank 2: mpi-03.csse.rose-hulman.edu
Rank 3: mpi-03.csse.rose-hulman.edu
```

Getting the machine name is done using `gethostname`. You can find an example in `hostname.c` in the MPI directory on Moodle.

For yourself, you should note what factors seem to impact communication speed.

4. The following code generates pseudo-random floats between 0 and 1.

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<time.h>

int main(){

  srand(time(NULL));
  float r;
  int i;
  for (i=0;i<10;i++){
    r=(float)rand()/RAND_MAX;
    printf("%f\n",r);
  }

}
```

Notice the need to seed the random number generator (we'll see this again later), and the `float` beside the `rand()`. Those are important.

The first quadrant of the unit circle has area $\frac{\pi}{4}$. If we generate random pairs in the unit square $0 \le x \le 1$, $0 \le y \le 1$, we expect them to fall in the first quadrant of the unit circle with probability $\frac{\pi}{4}$.

One method for estimating π is this:

(a) Generate T random pairs, (x_i, y_i), with $0 \le x, y \le 1$ and T large.

(b) Record the number of pairs that fall in the unit circle, call that number S.

(c) S/T converges to $\pi/4$ for large T (if the distribution is uniform). Thus, $4S/T$ converges to π.

The more trials the better. Write a program that runs T trials on each process. After T trials each process sends its estimate of π back to the master process. The master process averages the results and displays the estimate of π to the user. The master node should prompt the user for the value of T and distribute it to each process.