MA/CSSE 335 Project 1
Due 4/17

Directions

- All code must compile using `mpicc filename.c`
- The standard code is provided in *object files* this time. You can find them in `objs/`. Object files are more portable than executable files, you can think of them as "half-compiled". In order to run them on your VM or any other machine (like `grendel`) you will need to finish compiling them. For example: `mpicc objs/ring_broadcast_standard.o -o ring_broadcast_standard`
  You will need to do this on every different machine you try.

Level: Basic

1. Write a program that prompts the user for an integer value and distributes the value to all of the MPI processes. Rank `0` should do the prompting and reading, and should use `MPI_Send` to send the data to each worker process in turn. Upon receiving the message, each worker process should print out its rank and the value it received. Values should be read until a negative integer is given as input, at which point the master process should send out the negative integer to each worker, causing processes to exit gracefully. Name the file `broadcast_messages.c`
   - **Note:** The `printf` command is a bit finicky. If output of `printf` is not displaying immediately, try following the `printf` with `fflush(stdout)`.
   - **Note:** On this problem, the actual output of your code is not graded. Please don't make it too ugly. The only thing being tested is the actual communication pattern of your processes.
2. Write a program that behaves similarly to the previous program, but sends the data in a ring. That is, rank `0` prompts the user for data. Upon receiving the input, rank `0` passes the data to rand `1`, rank `1` prints the data and sends the data to rank `2`, and so on. The last rank should not send the data back to rank `0`. Name the file `ring_broadcast.c`
   - **Note:** Again, the actual output of your code does not matter.
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 $N/(T/2)$ where T is the round trip time.

   Write a program that, given an even number of processes to use, takes a message size in megabytes, B, a number of experiments to run, E. The program should exit *gracefully* if P is not even. The program should pair process $2i$ and $2i+1$, and test the communication speed between the processes using described above and average the outcome of all E experiments.

   Sample input is:

```
./comm_speed 1024 10
```

This would send 1024 megabytes between each pair of processors, and would repeat the round trip 10 times. The total elapsed time is calculated, and then the average communication speed is reported.

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

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

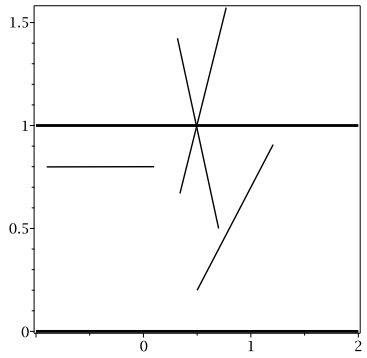
Name the file `comm_speed.c`

Note:

- You will need a working copy of the `comm_speed_standard` executable in order to run the testing code.
- A kilobyte is traditionally defined as 1024 bytes, and a megabyte is 1024 kilobytes.
- Most of the output of this program is not graded. However, you must generate one line of output for each pair of processes, and those lines must contain the phrase "`Rate:`". For example, if there are 6 total processes, there must be three lines detailing the communication speed between the 3 pairs of processes.

Level: Advanced
Complete the three problems in the basic level, in addition, complete the problems below.

1. Consider a long strip of paper with width one unit. Imagine dropping needles, also of length one unit, in such a way that one end of the needle is guaranteed to land between $x = 0$ and $x = 1$. It turns out (this is called the *Buffon's Needle problem*) that needles will cross the upper edge of the paper with probability $\frac{1}{\pi}$.



One method for estimating π is this:

(a) Randomly toss a needle.

(b) Record the number of needles that cross the upper edge of the paper. Estimate the probability that a needle crosses the upper edge by calculating $P = \frac{number\ of\ crossing}{number\ of\ trials}$. Then $\pi \approx \frac{1}{P}$.

The more trials the better. Write a program that takes an integer T from the command line, and 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.

Name the file `Buffon.c`.

- **Note on generating random numbers:** 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 classical method for seeding a random number generator is with `time(NULL)`, which returns the number of seconds elapsed since Jan 1, 1970. However, when running in parallel, this is often **not** a good way to seed the random number generator. This is because all of our processes will start at a similar time, possibly within one second of each other. In this case, many processes will seed the random number generator with identical seeds, which will then generate identical pseudorandom sequences. If we are interested in obtaining independent samples, this is obviously a Very Bad Thing.

  A typical way to overcome this problem is to have the master processor generate a "base seed" using `time(NULL)`. The master processor sends this seed to all workers, who then seed their random number generators with `base_seed+100rank`. Other possibilities certainly exist, but this does the job.

- **Note:** In order to "toss a needle", you should generate a random endpoint of the needle, `x_end` and `y_end`,both between 0 and 1, and then randomly generate an angle of elevation `theta`, between 0 and 2π.
- **Note:** In order for testing to work properly, you must use the supplied function `crosses_line` *every time* you check to see whether a needle crosses the upper edge of the paper. You must also leave the global variable `crosses_line_calls` defined.

2. A palindrome is a word that spells itself upon reversal. For instance, "dad", "radar", and "civic" are palindromes. A multi-word palindrome is a sequence of words that when concatenated form a palindrome. For instance, "my gym", "no lemon no melon", and "I did did I" are multi-word palindromes. Write a program that searches a given dictionary for n-word palindromes by considering all possible combinations of n words from the dictionary. As n grows, the number of possibilities grows quickly. If our dictionary contains w words, there are w^n combinations of words to check. Call the file `palindrome.c`.

   Your code must support an option for choosing the dictionary to check against, and it must support an option for changing the number of words/palindrome we are searching for.

   The file `palindrome_start.c` contains a start on the code, and already support all the desired command line options. To specify the number of words use the command line option `--numwords=<n>` and to specify the dictionary use `--dictfile=<filename>`.

   The file `dict.c` contains a simple dictionary data structure. Every `dictionary` contains a field `size` that reports the number of words stored in the dictionary. The `data` field of the dictionary stores the words int he dictionary; `data[i]` is a a string and is the i^{th} word in the dictionary. The supplied

function `dict_open(char* filename,dictionary* dict)` reads the text file in `filename` into the supplied `dict` structure. The function `lookup(char* word,dictionary* dict)` returns the index of `word` in `dict`, or `-1` if `word` is not in `dict`.

Your code should run on any number of processors, and all processors should check roughly the same number of combinations to see if they are palindromes. You must use the supplied `ispalindrome` function to see if a string is a palindrome. The code should just output the palindrome words if `n`=1 and should print the component words separated by `/` if `n>1`.

In addition to completing the test `test_palindrome`, you must include in your zip file `3wordpalindromes.txt` that lists all 3 word palindromes that can be made from words in `dict.txt`. You will certainly want to run this on `grendel`!

- **Note:** For fun, you may create your own dictionary files. A dictionary file is simply a file containing words, one word per line, sorted in alphabetical order. `lookup` does not account for capitalization, so Adam and adam are different words as far as `lookup` is concerned. In the directory `dictionary_stuff` you will find several dictionary files, `dict.txt`, `dict1.txt`, `dict2.txt`, and `dict3.txt`. You will also find several raw lists of words in the various `raw` files.

  A raw word list must contain one word per line. You may use the utility `create_dict.py` to create a proper dictionary file from a word list. Several options are supported, you can get a list of options by running `./create_dict.py -h`. The supplied dictionary files were created from their corresponding raw files, and include only words with at least 3 letters.