Lab 8

In this lab, you will perform some experiments to familiarize yourself with MPI.

MPI

Begin by copying the given code from the course directory:

```bash
cp -r /home/courses/cs308/lab8 .
```

The `-r` means to do a recursive copy; `lab8` is a directory and you’re copying it and all its contents. Change into this directory. You’ll see two files, `hello.c` and `hostfile`.

Compile the first of these using

```bash
mpicc hello.c -o hello-username
```

where `username` is your username. Notice that we’re using a different compiler in order to create an MPI program. To run it, use

```bash
time mpirun hello-username
```

We’re prepending `time` in order to measure the program’s running time. For this version, the only output will be from `time`: the program should have taken about 12 seconds of “wall clock time” (the amount of time you experience). Looking at the code, this is essentially all waiting in the first loop.

To run multiple copies of this program, add the `-np` argument to the command line:

```bash
imetype mpirun hello-username
```

The number following `-np` is the number of copies to run. Note that the order matters; if you put `-np` after the name of the executable, only one copy will be executed and the `-np` will be passed to it as an argument.

With 4 copies, the program will take about twice as long. The first two fields of the `time` output will be basically unchanged; these give the amount of actual work that your program is doing. They are unchanged because they measure only a single copy. The other difference is that the program will actually generate some output: three lines talking about what processors 1, 2, and 3 sent. Looking at the code, this is what the program does; the first copy (processor 0) receives messages from the others (and prints them) while the other processors each send it a message. The overall pattern of all the results being accumulated on processor 0 is very common in MPI programs, though all processors are allowed to print if they desire (all of their output will be mingled).

To get another view of the program, open another window and run `top`. This will display the CPU usage of different programs running on the system. Redo the `mpirun` lines and you’ll see that as many copies of the program start as are indicated by the argument to `-np`. This will also explain the speed difference between one copy and four; with four, each of them gets half the time on one core, making each half as fast as when a single copy is running.

Running many copies of the program on a single machine is not really the goal of MPI. Instead, we’d like to run one or maybe two copies (= the number of cores) on each machine. To do this, we need to tell `mpirun` which machines to use. This is the purpose of `hostfile`, which contains a list of the Crash and Burn lab
machines. You can tell `mpirun` to use these machines by including `--hostfile hostfile` (the first is the option and the second is the file name as an argument) on the command line (again, before the name of the executable). Use `top` locally and on the remote machines to see that this causes MPI to create a single instance on each machine.

That concludes our introduction to the basic send and receive calls of MPI. In the remaining time, see if you can use these to create a parallel application. I suggest using it to parallelize the numerical integration problem described in HW 6 of the most recent offering of CS 226 (see the 226 webpage for problem description). Start by using `MPI_Send` and `MPI_Recv` to bring all the values to node 0. If you complete this, modify the code to use `MPI_Reduce` since what you really want to do is to combine the value from each processor using addition, aka a reduction using `+`. 