Prof. Jingke Li (FAB 120-06, li@cs.pdx.edu); Classes: TTh 10:00-11:50 @ FAB 40-06; Office Hours: TTh 12:00-13:00.

Programming Project 3
(Due Thursday, 5/31/12)

This project is to practice message-passing programming with the use of the LAM MPI system. This is a two-part project: the first part is a simple exercise, and the second part implements a real algorithm.

Preparation

For this project, you'll need to use the Linux machines in the CS Linux Lab. The following are the names of available machines:

```
african chatham chinstrap emperor erect-crested fiordland galapagos
gentoo king kororaa little macaroni magellanic rockhopper royal
snares white-flipper yellow-eyed
```

LAM MPI (version 7.1.2) is already installed on these machines. You just need to create one or more LAM host files in your home directory. You may want to create two host files, one containing just two nodes for debugging use, and the other containing all nodes for timing use.

PART 1. A Simple Ring Program

Read and understand the attached program `trivial.c`. Compile and run it. Now write a program **ring.c** based on the given program. Instead of sending and receiving a message between two processes, your program will involve *all* processes that are active in the MPI run (*Note:* the total number of processes is not controlled by an MPI program itself.)

In your program, process 0 (i.e. process with `rank==0`) sends a message containing an arbitrary integer number (say 10) to process 1; upon receiving the number, process 1 increases its value by 1, and sends the new number to process 2; and this goes on. The last process in the chain increases the number it receives by 1, and sends it back to process 0. Like in `trivial.c`, each process should make the sending and receiving of messages visible by printing out a message containing it's rank and the integer's value, and it should terminate itself after printing.

A sample output from the new program is shown here (*Note:* The order of the messages do not necessarily reflect the order of the corresponding print actions.):

```
mpirun -v N ring
8442 ring running on n0 (o)
4962 ring running on n1
7165 ring running on n2
23380 ring running on n3
rank 1 received integer 10
rank 1 sent integer 11
rank 2 received integer 11
rank 2 sent integer 12
rank 3 received integer 12
rank 3 sent integer 13
rank 0 sent integer 10
rank 0 received integer 13
```

Test your program with at least 6 processes/hosts in the ring.

PART 2. A Sorting Algorithm

This part is to implement a sorting algorithm and to collect and analyze timing data from its execution. More specifically, your task is to implement a sorting algorithm to sort integer data from an input file into an ascending order and write the result to an output file. You have the freedom to choose the algorithm.

- **Algorithm** — You may choose *any* parallel sorting algorithm to implement. However, if you choose a fine-grained algorithm, such as parallel bubble sort or odd-even transposition sort, you need to develop a reasonably good coarse-grained version of it.
- **Program** — The sorting program should be implemented in the *SPMD* style. It should take two command-line arguments: the names of the input and output files. The program should reads data from the input file; sort it into an ascending order; and writes the result to the output file.
- **Data File Format** — The data to be sorted are four-byte unsigned integers (`unsigned int`). Both the input and output files are *byte* files of the same format:
  - The very first four bytes encode an unsigned integer value representing the number the data items in the file (not including the count itself).
  - The rest of the bytes are the encoding of the data items, which are simply laid out one after another.

  For example, for an eight-item data file, there will be $9 \times 4 = 36$ bytes.

  A pair of programs, `datagen.c` and `verify.c`, are provided to you for dealing with data files. `datagen` is a simple random data generator. It takes an integer command-line argument, `N`, and generates `N` random four-byte values, which can be saved in a data file:

  ```
  datagen 1024 > data1k
  ```

  `verify` is a result-verifying program. It can be used to verify that the four-byte values in a given data file are sorted in an ascending order:

  ```
  verify <datafile>
  ```

  Finally, there is a Unix/Linux utility, `od`, that can be used to view the content of a byte file:

  `od -t u1 < datafile` — translates each byte in `datafile` into an unsigned integer;
  `od -t u4 < datafile` — translates each four-byte group in `datafile` into an unsigned integer.
- **I/O Handling** — You may decide to have a single process do the I/O, or you may decide to let everyone do their part. Some of the considerations are: How to minimize memory buffer size? How to guarantee data ordering in the output file?
- **Timing** — Insert timing routines in your program to collect timing data. Use the `MPI_Wtime()` routine. We are interested in measuring two versions of total elapsed time: one including everything and one excluding I/O. Find the right points to insert the timing routine calls for these measurements.

Your program should work correctly with any number of processes and hosts, including the case of just one process, and any size of of data file. The workload among the processes need to be roughly balanced.

Run your program with different input data sizes, including at least 10,000, 50,000, and 100,000 data item cases. For each data size, try the program with different number of hosts, including at least, 1, 2, 4, and 8.

Don't forget to run `lamhalt` to terminate your LAM daemons after you are done with running your programs.

Report

Write a summary report covering the following topics:

- *The algorithm* — What algorithm did you implement? Why did you choose this algorithm?
- *Implementation* — Given a brief description of your program's organization. Did you encounter any difficulties in the implementation?
- *Timing results* — Tabulate your timing results nicely. What is the expected performance of the algorithm? Did you observe that performance from the measured performance data? Did you observe any speedup when running the program on more hosts? Explain why the observed performance is what it is.
- *General comments* — Do you have any advice for people who wants to sort a large number of data items? What's your recommendation on algorithms, number of processes, and number of hosts?

Grading

The two parts respectively worth 2 and 8 points. Normal criteria apply, *i.e.* correctness, performance, and report. Undergraduates will be graded more leniently. Extra credits may be awarded to exceptionally good or challenging project.

What to Turn In:

Make a `zip` or `tar` file containing your two source programs and the summary. Use the Dropbox on the D2L site to submit.