Prof. Jingke Li (FAB 120-06, li@cs.pdx.edu); Class: MW 14:00-15:50 @ UTS 308; Office Hr: M 13:00-14:00 & by appt.

Programming Assignment 3
(Due Thursday, 5/20/15)

This assignment is to practice message-passing programming with the use of the MPI library. After a small warming-up program, you'll implement an external sorting algorithm, *i.e.*, reading data from a file, sorting the data, and write the result to another file.

For this assignment, all students will implement the same two programs. However, CS515 students are required to include timing results for the second program (see below). This assignment carries a total of 100 points.

Preparation

For this assignment, you'll need to use Open MPI on the CS Linux Lab machines. There are 49 identical machines in the Linux lab, all sharing a single file system. To remotely log into a lab machine, use the address `linuxlab.cs.pdx.edu`. You'll be connected to a random machine in the lab. Since all the lab machines are connected to the same file system, it does not matter which machine you use for compilation and execution. (*Note:* The multi-core server, `babbage`, is not a lab machine.) After logging in, run `addpkg` and select "Open MPI" so that it will be included in your environment after you re-login.

Download and unzip the file `assign3.zip`, you'll see a directory `assign3`. In it, there is a file `mpihosts`, which contains a list of the names of all 49 Linux Lab machines. You may set the environment variable `OMPI_MCA_orte_default_hostfile` to point to this file, so that you don't have to type the host names every time you run an MPI program. You should create a few additional host files to hold subsets of this list, *e.g.* `mpihosts2` (2 hosts), `mpihosts4` (4 hosts), `mpihosts8` (8 hosts), etc.

PART 1. A Simple Ring Program

Read and understand the provided program `simple.c`. It implements a pair of send-receive actions between two processes. The sender sends an integer to the receiver; the receiver decreases the value by one and sends it back. Even though the message-passing happens only between two processes, the program can run with any number of processes. It can also take a command-line argument, an integer to be used as the message value. Compile and run it. Here are some examples:

```
linux> mpicc -o simple simple.c                 // compile
linux> mpirun -hostfile mpihosts -n 2 simple    // run with 2 processes
linux> mpirun -hostfile mpihosts -n 4 simple    // run with 4 processes
linux> mpirun -hostfile mpihosts -n 4 simple 100 // run with 4 processes and 100 as message value
```

Now write a program `ring.c` based on this program. Instead of send-receive actions between two processes, your program will involve *all* active processes. In the program, process 0 (*i.e.* process with `rank==0`) sends an integer to process 1; upon receiving the integer, process 1 decreases its value by 1, and sends the new number to process 2; process 2 does the same thing, and sends a new number to process 3; and this action goes on. The last process in the active set sends its modified number back to process 0. Like in `simple.c`, each process should make the sending and receiving actions visible by printing out a message showing its rank, its host name, and the involved integer's value. Note that the total number of active processes is not controlled by the MPI program itself. Compile your program with `mpicc`, and test it with multiple combinations of runtime parameters.

PART 2. An External Sorting Program

This part is to implement an external sorting algorithm. Your program should be called `extsort.c`. It should be implemented in the SPMD style. It should take two command-line arguments: the input file name and the output file name. The program should reads data from the input file; follow the sorting algorithm shown below to sort the data; and writes the result to the output file.

Algorithm

The sorting algorithm is a simplified version of sample sort on integers. Assume data size is N and the total number of processes is P ($P \geq 2$). (Assume also that $N > 10P$.)

1. Process 0 reads in all data from the input file.
2. Process 0 sorts the first $10P$ elements, and selects elements at positions 10, 20, ..., and $10(P-1)$ as pivots. (They will be referred to as $pivot[0]$, $pivot[1]$, ..., and $pivot[P-1]$.)
3. Process 0 partitions the data into P buckets — elements whose values are smaller than $pivot[0]$ are placed in $bucket[0]$, elements whose values are in between of $pivot[0]$ and $pivot[1]$ are placed in $bucket[1]$, and so forth.
4. Process 0 keeps $bucket[0]$ to itself, and sends the rest $P-1$ buckets to their corresponding processes, *i.e.* $bucket[i]$ to process i.
5. Every process sorts its bucket using quicksort.
6. The processes writes their results to the output file, in the process rank order.

A copy of the quicksort program can be found in the `assign3` directory. You may copy the useful part into your `extsort.c` program.

Data File Format

The data to be sorted are four-byte integers (C type `int`). Both the input and output files are *byte* files in which each consecutive group of four bytes encode an integer. For example, for the four integers, 860, 386, 103, and 282, the data file will contain the following 16 bytes:

```
5c 03 00 00   // 1st int 860
82 01 00 00   // 2nd int 386
67 00 00 00   // 3rd int 103
1a 01 00 00   // 4th int 282
```

Note that the content of a binary file is not directly viewable. To see a binary content, use the Linux utility, `od`, with a proper switch:

```
linux> od -i data1k   -- display binary content as integers
```

A pair of programs, `datagen.c` and `verify.c`, are provided to you for dealing with the data files. The program `datagen` takes an integer command-line argument, `N`, and generates a random permutation of `N` integers, 1, ..., `N`, which can be saved in a data file:

```
linux> ./datagen 1024 > data1k
```

The program `verify` can be used to verify that the integer values in a given data file are sorted in an ascending order:

```
linux> ./verify out1k
Data in out1k are sorted.
```

File I/O

Use MPI's file I/O routines to handle input and output. For the program's input, only Process 0 is involved, so use `MPI_COMM_SELF` when opening the input file. In this case, there is no need to use the routine `MPI_File_set_view()`.

For the output, all processes are involved. Here you have two choices.

(1) You can arrange the processes to take turn to access the output file. Each process opens and closes the file for its own use. (Hence use `MPI_COMM_SELF` again.) It appends its output to the end of the file. For this approach, the issue to resolve is to have processes take turns in the process rank order. (*Hint:* Think about the `ring.c` program, and use messages to enforce the required order.)

(2) Have all processes write to the same file concurrently. For this approach to work, you need to have all processes call `MPI_File_open()` with `MPI_COMM_WORLD`, and use the routine `MPI_File_set_view()` to set their offset values. Reference the `iodemo.c` program for the usage pattern of this routine. One issue you need to resolve is to figure out the right offset for each process.

Timing [CS515 Students]

Insert the MPI timing routine `MPI_Wtime()` in your program to collect timing data. You may want to measure two versions of total elapsed time: one includes everything and the other excludes I/O actions. Find the right points to insert the timing routine calls for these measurements.

Report

As usual, write a summary report on the assignment (around 2 pages). You may include anything you want to discuss. If you have collected timing results, include them with some comments.

What to Turn In:

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