ARIZONA STATE UNIVERSITY

CSE 310, All Sections — **Data Structures and Algorithms** — Fall 2015

Project #1

Available 08/24/2015; milestone due 09/09/2015; complete project due 09/23/2015

This project has two major goals:

1. The first is to implement various sorting and selection algorithms, both in serial and in parallel.
2. The second is to run experiments comparing the performance of the algorithms, to collect and plot data, and to interpret the results.

Note: This project is to be completed individually. Your implementation *must* use C/C++ and your code *must* run on the Linux machine `general.asu.edu`. See §2 for full project requirements. The output is prescribed because a script will be used to check the correctness of your program. Therefore, absolutely no changes to these project requirements are permitted.

It is suggested you use a version control system as you develop your solution to this project. Your code repository should be private to prevent anyone from plagiarizing your work.

1 The Wearable Fitness Craze

Wearable fitness tracker devices are hot commodities these days. They can collect metrics such as number of steps walked, heart rate, and GPS data, among others. In this project you will write a serial program that executes a sequence of commands that operate on individuals' fitness data. Valid commands include:

- `Start Name`, where `Name` is a character string of maximum length 20 alphabetic characters representing the name of the person whose fitness data is in the file `Name.txt`. The structure of file `Name.txt` is an integer N indicating that N additional lines of integer fitness data, namely number of steps taken in a day, follow. The commands that follow until the `End` command are to be applied to the fitness data in the file `Name.txt`. The output of the `Start` command is:

  `Processing fitness data of:  Name`

- `End Name`, indicates the end of the processing for the fitness data for `Name`; the `Start` and `End` commands will always come in pairs with matching names. Any memory dynamically allocated for `Name` must be freed on an `End` command. The output of the `End` command is:

  `End of processing fitness data for:  Name`

- `InsertionSort`, first reads N from the file `Name.txt`, dynamically allocates an array of integers of size N (if it has not already been allocated), and then reads the remaining N integers into the array. Then an insertion sort is used to sort the fitness data. You must implement the insertion sort based on the pseudocode discussed in class. Output the number of comparisons made c_i and the number of swaps s_i made in sorting the data, and the time t_i taken to run the insertion sort:

  `Number of comparisons made by insertion sort:` c_i

  `Number of swaps made by insertion sort:` s_i

  `Time to run insertion sort (ms):` t_i

- `MergeSort`, first reads N from the file `Name.txt`, dynamically allocates an array of size N (if it has not already been allocated), and then reads the remaining N integers into the array. Then a merge sort is used to sort the fitness data. You must implement the merge sort based on the pseudocode discussed in class. Output the number of comparisons c_m made and the number of swaps s_m made in sorting the data, and the time t_m taken to run the merge sort:

`Number of comparisons made by merge sort:` c_m

`Number of swaps made by merge sort:` s_m

`Time to run merge sort (ms):` t_m

- `ParallelMergeSort`, first reads N from the file `Name.txt`, dynamically allocates an array of size N (if it has not already been allocated), and then reads the remaining N integers into the array. Then a parallel merge sort is used to sort the fitness data. You must implement the parallel merge sort using `OpenMP`. Output the number of comparisons made c_p and the number of swaps s_p made in sorting the data, and the time t_p taken to run the sort using the parallel merge sort method. (A parallel merge sort algorithm will be presented in class. You will gain experience using `OpenMP` in Assignment #1.)

  `Number of comparisons made by parallel merge sort:` c_p

  `Number of swaps made by parallel merge sort:` s_p

  `Time to run parallel merge sort (ms):` t_p

  Every time a sort command is encountered — either serial or parallel — the array should be reinitialized. That is, if the array has been allocated, you may overwrite it with the N integers in `Name.txt`. If the array has not been allocated, it should be allocated and then initialized.

- `Select item`, selects an item from a sorted array. If the array has not been sorted (by a previous serial or a parallel sort), the following error must be reported:

  `Unable to select from an unsorted array.`

  Valid `item`s for selection include:

  - `max`, prints the largest value in the fitness data, i.e., data item N.
  - `min`, prints the smallest item in the fitness data, i.e., first data item.
  - `median`, prints the median value of the fitness data.
  - `k`, where $1 \leq$ `k` $\leq N$, selects the `k`th item in the fitness data and prints it. When `k` $= 1$ this is equivalent to selecting the `min`. When `k` $= N$ this is equivalent to selecting the `max`. (Note that `k` may not equal the array index position where the `k`th item is found.)

  You must implement the selection algorithm based on the pseudocode discussed in class. Output the selected item i, and the time taken t_s to run the selection of the `item` as follows:

  `Selecting item:` i

  `Time to run the selection (ms):` t_s

  If $k > N$, then output:

  `Invalid selection.`

- `Average`, prints the average of the fitness data, computed serially. Output the average a number of steps taken in a day, and the time t_a taken to compute the average.

  `Average number of steps:` a

  `Time to run the average (ms):` t_a

- `ParallelAverage`, prints the average of the fitness data where the summation of the N data items is computed in parallel. Output the average a_p number of steps, and the time t_{pa} taken to compute the average when the summation is computed in parallel using `OpenMP`. (A parallel summation algorithm will be presented in class.)

  `Average number of steps (parallel sum):` a_p

  `Time to run the average (parallel sum) (ms):` t_{pa}

- `Exit`, indicates that there are no more commands to execute, i.e., the program terminates. Output: `Program terminating.`

You may assume the format of the input data is correct. Only two error conditions need to be checked:

1. *Ensure that a sort (either serial or parallel) has run before the selection of any item.*
2. *If the parameter to the* `select` *command is* `k`*, then* $1 \leq k \leq N$.

Consider the following valid command sequence. Comments are provided for each command, though these are not part of the input.

```
Start Ravi            // Working on Ravi's fitness data
InsertionSort         // Use insertion sort to sort Ravi's fitness data
Select max            // Print the maximum number of steps Ravi has walked in a day
ParallelAverage       // The average number of steps walked using a parallel sum
MergeSort             // Ravi's data read in again and sorted using merge sort
Select min            // The fewest steps Ravi has walked in a day
Select 5              // The fifth smallest number of steps Ravi walked in a day
Select median         // The median number of steps Ravi walked in a day
Average               // The average number of steps walked in a day (serial)
End Ravi              // Free any memory dynamically allocated for Ravi
Start Amy             // Working on Amy's fitness data
ParallelMergeSort     // Sort Amy's data using a parallel merge sort algorithm
Select max            // The maximum number of steps Amy has walked in a day
End Amy               // Free any memory dynamically allocated for Amy
Start Bingli          // Working on Bingli's fitness data
Select max            // Error: can't select from data that has not been sorted
End Bingli            // Free any memory dynamically allocated for Bingli
Exit
```

2 Program Requirements for Project #1

1. Write a C/C++ program that implements all of the commands described in §1. You must use `OpenMP` to implement the commands `ParallelMergeSort` and textParallelAverage in parallel; all other commands are to be implemented in serial.
2. Design experiments that exercise your program to answer the questions in §3. A report with figures and data to support your answers is expected.
3. Provide a `Makefile` that compiles your program into an executable named `p1`. This executable must be able to run commands read from standard input directly, or from a script file redirected from standard input (this should require no change to your program).

Sample fitness data files and sample scripts will be provided on Blackboard; use them to test the correctness of your programs.

3 Experimentation

Devise script files with a valid command sequence to help you answer the following questions.

1. Plot the number of comparisons made, number of swaps made, and run time as a function of fitness data size N for insertion sort and merge sort.

- Do you observe a cross-over point? That is, can you recommend when you should use one algorithm over the other?

2. Plot the performance (run time) as a function of fitness data size N for computing the average serially, and the average when the sum is computed in parallel.
   - Do you observe a cross-over point? That is, is there a size at which the parallel sum always improves the time for the computation of the average?
3. Plot the number of comparisons made, number of swaps made, and run time as a function of fitness data size N for merge sort and parallel merge sort.
   - Do you observe a cross-over point? That is, is there a size at which the parallel algorithm overtakes the serial algorithm?

4 Submission Instructions

All submissions are electronic. This project has two submission deadlines.

1. The milestone deadline is before midnight on Wednesday, 09/09/2015.
2. The complete project deadline is before midnight on Wednesday, 09/23/2015.

4.1 Requirements for Milestone Deadline

By the milestone deadline, your project must implement the following commands as a minimum: `Start`, `End`, `InsertionSort`, `Select`, `Average`, and `Exit`.

Submit electronically, before midnight on Wednesday, 09/09/2015 using the submission link on Blackboard for the Project #1 milestone, a zip[1] file named `yourFirstName-yourLastName.zip` containing the following items:

Project State (5%): In a folder (directory) named `State` provide a brief report (.txt, .doc, .docx, .pdf) that addresses the following:

1. Describe any problems encountered in your implementation for this project milestone.
2. Describe any known bugs and/or incomplete command implementation for the project milestone.
3. Describe any significant collaboration with anyone (peers or otherwise) and/or clearly reference any external code bases used.

Implementation (50%): In a folder (directory) named `Code` provide:

1. Your well documented C/C++ source code implementing the commands required for this project milestone.
2. A `Makefile` that compiles your program to an executable named `p1` on the Linux machine `general.asu.edu`. Our TA will write a script to compile and run all student submissions on `general.asu.edu`; therefore executing the command `make p1` in the `Code` directory must produce the executable `p1` also located in the `Code` directory.

Correctness (45%): The correctness of your program will be determined by running a series of scripts as input to your program, some of which will be provided to you on Blackboard prior to the deadline for testing purposes. For the milestone deadline, the scripts will only contain a subset of the commands described in §1 given above. As described in §2, your program must be able to run commands read from standard input directly, or from a script file redirected from standard input. **You must not use file operations to read the input!**

The milestone is worth 30% of the total project grade.

[1] **Do not** use any other archiving program except `zip`.

4.2 Requirements for Complete Project Deadline

Submit electronically, before midnight on Wednesday, 09/23/2015 using the submission link on Blackboard for the complete Project #1, a zip[2] file named `yourFirstName-yourLastName.zip` containing the following items:

Project State (5%): Follow the same instructions for Project State as in §4.1.

Experimentation and Report (15%): In a folder (directory) named `Report` provide a brief report (.txt, .doc, .docx, .pdf) that addresses the following:

1. Discuss the design of your experiments.
2. Present figures plotting the results of experimentation as requested in §3 along with your answers to the questions found there. Note that there is no single correct answer! You should just try to explain the results you obtained.

Implementation (50%): Follow the same instructions for Implementation as in §4.1.

Correctness (20%): The same instructions for Correctness as in §4.1 apply except that the scripts will exercise all commands from §1 rather than a subset of them.

5 Marking Guide

The project milestone is out of 100 marks.

Project State (5%): Summary of project state, use of a zip file, and directory structure required.

Implementation (50%): 40% for your code including proper memory management, 10% for a correct `Makefile`.

Correctness (45%): 40% for correct output from script files, 5% for redirection from standard input.

The full project is out of 100 marks.

Project State (5%): Summary of project state, use of a zip file, and directory structure required.

Experimentation and Report (15%): Experiment design, results of experimentation, and answers to questions.

Implementation (50%): 40% for your code including proper memory management, 10% for a correct `Makefile`.

Correctness (20%): 15% for correct output from script files, 5% for redirection from standard input.

[2] **Do not** use any other archiving program except `zip`.