EE/CSCI 451
Fall 2019
Programming Homework 3

Assigned: September 27, 2019
Due: October 11, 2019, before 11:59 pm, submit via blackboard
Total Points: 50

General Instructions

- You may discuss the algorithms. However, the programs have to be written individually.
- Submit **the source code and the report** via **Blackboard**. The report file should include reports for all problems and should be in pdf format with file name `report.pdf`. Put all the files in a zip file with file name `<firstname>_<uscid>_phw<programming homework number>.zip` (do not create an additional folder inside the zip file). For example, `alice_123456_phw1.zip` should contain the source codes, makefile, the report and other necessary files.
- The executables should be named `p1a` and `p1b` for the two versions of problem 1. For problem 2, you only need to submit the OpenMP version (which calls the serial version by each thread). The executables should be named `p2`. For problem 3, the executables should be name `p3`.
- Your program should be written in C or C++. You can use any operating systems and compilers to develope your program. However, we will test your program on a x86 linux with the latest version of g++ and gcc. Make sure you makefile could compile the executables for **all** the problems (hint: set multiple targets for the makefile) and the name of the executables are correct. If your program has error when we compile or run your program, you will lose at least 50% of credits.

1 Estimating π [10 points]

In this problem, you will use OpenMP directives to parallelize a serial program. The serial program is given in 'p1_serial.c', which uses the algorithm we discussed to estimate the value of π. You need use OpenMP to parallelize the loop between Line 28 and Line 32. To compile the program, type: gcc -lrt -fopenmp -o run p1_serial.c

1. Version a: Use 4 threads and the **DO/for** directive to parallelize the loop.
2. Version b: Use 2 threads and the **SECTIONS** directive to parallelize the loop.
3. Report the execution time of serial version, Version a and Version b, respectively.

Hint: you may also need the **REDUCTION** data attribute.

2 Sorting [15 points]

In this problem, you need implement the quick sort algorithm to sort an array in ascending order. Quick sort is a divide and conquer algorithm. The algorithm first divides a large array into two smaller sub-arrays: the low elements and the high elements; then recursively sorts the sub-arrays. The steps are:

- Pick an element, called a pivot, from the array.
- Reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

In the given program 'p2.c', the array m ($size$ of $m = 16M$) which you need sort is generated.

1. Serial Quicksort function: implement a Quicksort function to sort the array. The input of the function includes the pointer of the array, the index of the starting element and ending element. For example, **Quicksort(m, 20, 100)** will sort the array m from $m[20]$ to $m[100]$. Report the execution time of the serial version by calling Quicksort(m, 0, $size - 1$).
2. OpenMP Quicksort function: randomly pick up an element of m, $m[rand()\%size]$, as the pivot, reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position, say f. Run 2 threads in parallel, one calling Quicksort$(m, 0, f - 1)$ and the other calling Quicksort$(m, f, size - 1)$. Report its execution time. **Note that the Quicksort function is the same as the one used in serial version.**

Note: you can get more details about quick sort algorithm from textbook, Section 9.4.

3 Parallel K-Means [25 points]

In PHW 2, you implemented the K-Means algorithm using Pthreads. In this problem, you will use mutex and condition variable to realize synchronization instead of iteratively joining the threads. Let p denote the number of threads that you need. In this version, you only create and join p threads once (not iteratively create and join the threads). This version of K-Means algorithm has the following steps:

1. Initialize a mean value for each cluster.
2. Partition and distribute the data elements among the threads. Each thread is responsible for a subset of data elements.
3. Each thread assigns its data elements to the corresponding cluster. Each thread also keeps track of the number of data element assigned to each cluster in current iteration and the corresponding sum value.
4. Let us use r to denote the number of threads which have completed the work for the current iteration. At the beginning of each iteration, $r = 0$. When a thread finishes its work for the current iteration, it checks the value of r. (Note that r is a shared variable, a mutex is needed whenever any thread tries to read/write it.)
    - If $r < p - 1$, $r \leftarrow r + 1$, the thread goes to sleep.
    - If $r = p - 1$, $r \leftarrow 0$, the thread recomputes the mean value of each cluster based on the local intermediate data of all the threads, then reinitializes the intermediate data of each thread. If the algorithm does not converge after the current iteration, this thread sends a broadcast single to wake up all the threads. Go to Step 3 to start a new iteration.
5. Join the threads. Replace the value of each data with the mean value of the cluster which the data belongs to.

In this problem, the input data and initial mean values for the clusters are the same as in PHW2. To simplify the implementation, you do not need check the convergence; run **50** iterations and output the matrix into the file named 'output.raw'. Pass the number of threads p as a command line parameter similar to PHW2. In your report, you need report the execution time for the **50** iterations (excluding the read/write time) for $p = 1, 2, 4, 8$. Compare the execution time with the version which you implemented in PHW 2, discuss your observation in your report.