EE/CSCI 451
Fall 2024
Programming Homework 3

Assigned: September 20, 2024
October 8, 2024 before 11:59 pm AOE, submit via Brightspace
Total Points: 100

General Instructions

- You may discuss the algorithms. However, the programs have to be written individually.
- Submit **the source code and the report** via **Brightspace**. 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_phw3.zip` should contain the source codes, makefile, the report and other necessary files.
- The executables should be named `p1a`, `p1b`, and `p1c` for the three versions of problem 1. The executables should be named `p2a` and `p2b` for the two versions of problem 2. For problem 3, you only need to submit the OpenMP version (which calls the serial version by each thread). The executable should be named `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.
- If your are running your code on CARC, use the following command to allocate more computation resource to run your program: *salloc –time=1:00:00 –cpus-per-task=8*

1 Parallel Matrix Multiplication [50 points]

In this problem, you will parallelize several version of Matrix Multiplication using OpenMP.

1. Version a: Parallelize the **naive** matrix multiplication which you implemented in PHW 1 using the **DO/for** directive.
2. Version b: Parallelize the **blocked** matrix multiplication which you implemented in PHW 1 using the **DO/for** directive. Use a block size of 16.
3. Version c: For this version, we will modify the **naive** matrix multiplication which you implemented in PHW 1. Transpose the B matrix in order to store it in column major order and modify your matrix multiplication. Parallelize this version using the **DO/for** directive.

- The matrix initialization is the same as that in PHW 1. The matrix size is $4K \times 4K$. Print out the execution time and the value of $C[100][100]$ in your program.
- Pass the square root of total number of threads p as a command line parameter [1](for # of threads = 16, pass 4 as an argument).
- Report the execution time for $p \times p = 1$ (the serial version you did in PHW1), $4, 16, 64$ and 256. Draw a diagram to show the execution time under various p. Briefly discuss your observation. Explain why you see the performance differences between the three versions.

2 Estimating π [20 points]

In this problem, you will use OpenMP directives to parallelize a serial program. The serial program is given in 'p2_serial.c', which uses an algorithm 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 -fopenmp -o run p2_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.

3 Sorting [30 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 'p3.c', the array m ($size$ of $m = 2M$) 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.**
3. Now, instead of $m[i] = size - i$, use the next line $m[i] = rand()$ to initialize your array and rerun your program with 2 threads. Report its execution time. Observe the performance difference and explain why.

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