EE/CSCI 451
Fall 2024
Programming Homework 2

Assigned: September 6, 2024
Due: September 20, 2024, 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_phw2.zip` should contain the source codes, makefile[2], the report and other necessary files.
- The executables should be named `p1a` and `p1b` for problem 1 and `p2a` and `p2b` for problem 2. Both `p1` and `p2` should only take one parameter, the number of threads. For example, to run the problem 1.1 with 4 threads, you should enter './p1a 4'.
- 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.

Example Program

print_msg.c is an example program. You can refer it for thread creation, joining and passing arguments to threads.
To compile it, type: gcc **-lpthread** -o go print_msg.c
To run it, type: ./go

1 Parallel Matrix Multiplication [40 points]

1.1 Local Output Variables

Parallelize the **naive** matrix multiplication which you implemented in PHW 1 using Pthreads. One simple way to parallelize the algorithm is evenly partitioning and distributing the computation works of the elements of output matrix among the threads and letting the threads work independently in parallel. Note, each thread (i, j) is responsible for computing output block $C(i, j)$. For example, assuming the matrix size is $n \times n$ and the number of threads is $p \times p$; let thread $\#(0, 0)$ compute the elements of rows 0 to $\frac{n}{p} - 1$ and columns 0 to $\frac{n}{p} - 1$ of the output matrix, thread $\#(0, 1)$ compute the elements of rows 0 to $\frac{n}{p} - 1$ and columns $\frac{n}{p}$ to $\frac{2n}{p} - 1$ of the output matrix, and so on.
w

- 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. An example diagram is shown in Figure 1. Briefly discuss your observation.
- In your report, discuss how you partition the computation works of the elements of the output matrix among the threads.

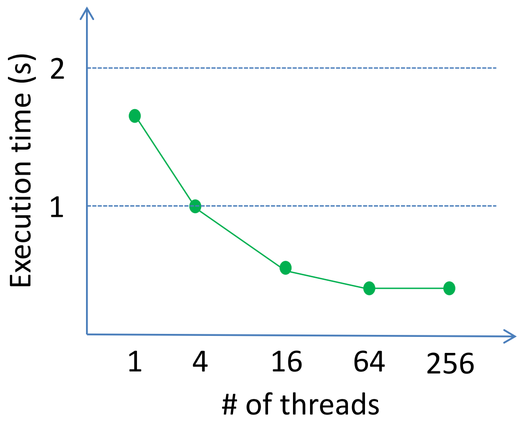

Figure 1: Example diagram

1.2 Shared Output Variables

We now used another method as discussed in Lecture 5 slides 18-24 to parallelize the algorithm, where we evenly partition and distribute the computation works of the elements of one input matrix among the threads and letting the threads work in parallel. Note each thread (i, j) accesses and owns input block $A(i, j)$, and is responsible to update all output $C(i, k)$ via matrix multiplication with $B(j, k)$, where k iterates from 1 to p. For example, assuming the matrix size is $n \times n$ and the number of threads is $p \times p$; let thread $\#(0, 0)$ compute the elements of rows 0 to $\frac{n}{p} - 1$ and columns 0 to $\frac{n}{p} - 1$ of the input matrix A multiplied by elements of rows 0 to $\frac{n}{p} - 1$ of the input matrix B (contributes to elements of rows 0 to $\frac{n}{p} - 1$ of the input matrix C), thread $\#(0, 1)$ compute the elements of rows 0 to $\frac{n}{p} - 1$ and columns $\frac{n}{p}$ to $\frac{2n}{p} - 1$ of the input matrix A multiplied by elements of rows $\frac{n}{p}$ to $\frac{2n}{p} - 1$ of the input matrix B (contributes to elements of rows 0 to $\frac{n}{p} - 1$ of the input matrix C), and so on. (Hint: You will use mutex in this problem to handle the situation of multiple thread manipulating the same row of output matrix C)

- 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].
- 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. An example diagram is shown in Figure 1. Briefly discuss your observation.
- Submit two .c/.cpp files and the report. ‘p1a.c’ for local output matrix multiplication; ‘p1b.c’ for shared output matrix multiplication. The output executables should be named ’p1a’ and ’p1b’ respectively.
- In your report, discuss how you partition the computation works of the elements of the output matrix among the threads.
- In your report, compare the execution time of problem 1.2 with 1.1 and explain any difference.

2 Parallel K-Means [60 points]

2.1 Iteratively creating and joining threads

In PHW 1, you implemented the K-Means algorithm. In this problem, you will need to parallelize your implementation using Pthreads. Let p denote the number of threads you create. The parallel 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 locally keeps track of the number of data element assigned to each cluster in current iteration and the corresponding sum value.
4. Synchronize the threads.
5. Recompute the mean value of each cluster.
6. Check convergence; if the algorithm converges, replace the value of each data with the mean value of the cluster which the data belongs to, then terminate the algorithm; otherwise, go to Step 2.

In this problem, the input data is the same matrix as in PHW 1 which is stored in the 'input.raw'; the value of each matrix element ranges from 0 to 255. Thus, the matrix can be displayed as an image. However, we will have **6** clusters (K=6). The initial mean values for the clusters are 0, 65, 100, 125, 190 and 255, respectively. To simplify the implementation, you do not need check the convergence; run **50** iterations (Step 2-5) then terminate the algorithm and output the matrix into the file named 'output1.raw'. Pass the number of threads p as a command line parameter. In your report, you will need to report the execution time for the **50** iterations (excluding the read/write time) for $p = 1, 2, 4, 8$.

2.2 One-time thread creation

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 2.1. To simplify the implementation, you do not need check the convergence; run **50** iterations and output the matrix into the file named 'output2.raw'. Pass the number of threads p as a command line parameter similar to 2.1. In your report, you will need to report the execution time for the **50** iterations (excluding the read/write time) for $p = 1, 2, 4, 8$.

- Implement a serial version without using Pthreads (No need to submit this program; already completed in PHW 1).
- Submit two .c/.cpp files and the report. 'p2a.c' for the iteratively creating and joining threads K-Means; 'p2b.c' for the one-time thread creation K-Means. The output executables should be named 'p2a' and 'p2b' respectively. Pass the number of threads p as a command line parameter
- Report the execution time for the **50** iterations (excluding the read/write time) for $p = 1, 2, 4, 8$. Compare between the serial, 2.1, and 2.2 versions with respect to execution time, discuss your observations.
- Show the images of the output matrix in your report. You can display the output images, 'output1.raw' and 'output2.raw', using the imageJ [4] software or the given Matlab script file, 'show_raw.m'. If you use 'show_raw.m', remember to specify the file name in the script.

References

[1] "Command Line Parameter Parsing,"
`http://www.codingunit.com/c-tutorial-command-line-parameter-parsing`

[2] "Using make and writing Makefiles,"
`http://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html`

[3] "rand - C++ Reference"
`http://www.cplusplus.com/reference/cstdlib/rand/`

[4] "imageJ,"
`https://imagej.net/ij/download.html`