CSMC 23010-1 Due April 16, 2019

Homework 1

Professor: Hank Hoffmann *TA: Miao He & Will Kong*

Note: Each assignment contains a theory section and programming section. The theory section involves solving problems from the book. The programming assignment involves designing, testing, and coding.

For each programming assignment you will submit the following: 1) a document including an initial design and test plan, 2) the final code and test programs, makefile, scripts, etc. and 3) a report describing and analyzing your results. The design and test document will be due before the code and writeup. We will discuss design and test for each programming assignment in class at least one week before the code and writeup are due. Requirements for the design and test document will be discussed in class.

Fair Warning: These assignments involve significant testing and measurement efforts. You will not be able to complete these by the due date without starting early. **No late assignments will be accepted.** Your professor and TAs are available for help with both theory and programming—do not be afraid to ask for help.

Theory

Provide solutions to questions 2, 3, 4, 5, 7, 8, 11, 14, 15, 16.

Programming

The design and test document is due 4/9/2019.

Programming Environment

We will be programming in C with POSIX threads (pthreads). There are many pthreads tutorials and overviews on the web. I prefer the one from Lawrence Livermore National Laboratory: https://computing.llnl.gov/tutorials/pthreads/

Pthreads provides several useful functions for mutual exclusion and synchronization. For most assignments, your final codes should not make use of these unless explicitly stated. However, they might be useful for debugging. For example, if you are not sure whether you have correctly implemented mutual exclusion, you can use a pthread mutex (MUTual EXclusion) object to replace your mutual exclusion protocol in the code. If the program starts to work, then you have a better idea of where to look for bugs.

The gcc compiler provides access to hardware synchronization instructions through gcc atomic builtins. You can read more about those here: https://gcc.gnu.org/onlinedocs/gcc-5.2.0/gcc/_005f_005fatomic-Builtins.html. Please do make use of these unless we explicitly tell you not to.

The `volatile` keyword will be essential to this course (although probably not for this first assignment). Understand what it means and understand how to use it. If you are unsure, post to Piazza—you are probably not the only one!

Compiling your Code

Programming assignments need to be 1) correct and 2) fast. We can use the compiler to help us (of course, it doesn't do everything).

The compiler can assist with correctness when you turn on the `--Wall --Werror` flags. These flags turn on all warnings and turn all warnings into errors. Using these flags catches a surprisingly large amount of bugs before you even run your program.

There are many flags that will affect performance, but the main one you need is -O3 (that is a capital letter "o"). Remember, there is no point in parallelizing code for performance if the single threaded code is compiled without optimization. Additionally, parallelizing code without optimization will generate misleading results that make your parallelization of a problem look better than it actually is.

Compile all code with –Wall –Werror -O3. Do not report performance numbers for code that was not compiled in this manner.

0.1 The First Programming Assignment

This assignment is a simple parallel program designed to get you familiar with the environment, pthreads, and parallelism. You will be implementing a parallel version of the Floyd-Warshall algorithm, which takes an input graph (in adjacency matrix format) and computes the shortest paths between all pairs of vertices. Info on this algorithm can be found here: https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm.

Your goal is to take a text file representing an adjacency matrix and output a text file containing a table with the shortest paths between each pair of vertices. The first line of the input file should have an integer representing the number n of vertices in the graph. The remainder of the text file should have n lines containing n integers each of which represents the weight of a directed edge connecting vertex i to vertex j. Let the maximum edge length be 1000. Use edge length of 10000000 to represent an infinite weight; i.e. no direct edge from i to j. Entry k on line k should be 0 to represent that the shortest path from a node to itself has no cost. For the output file, line i should represent starting at vertex i, and the jth entry of line i should be the shortest path from i to j.

Your program should operate in three phases. Phase 1 should read the file into memory. Phase 2 should execute the Floyd-Warshall algorithm. Phase 3 should output the shortest paths table. For this assignment, we only care about the performance (and thus parallelization) of phase 2. Your program should time phase 2 and report its performance, we will provide a `stopwatch` data object to make this easy. We will also provide access to a Linux/x86 system. This system should be used for taking all performance measurements, but it should not be used for development. Develop your code elsewhere and then evaluate its performance on the server.

For this assignment you should implement the following:

- SERIAL: a serial version of the program (with no reference to pthreads). For this portion, you should essentially copy the example code from wikipedia. This will serve as a performance reference for the parallel code.

- PARALLEL: a version of the program where phase 2 operates in parallel; i.e. multiple threads work together to implement the algorithm and finish the computation in a shorter amount of time. It is your responsibility to figure out how to parallelize this algorithm. The parallel implementation should accept two command line arguments: the first specifies the number of threads to use in the parallel portion of the code, and the second specifies the name of the text file containing the input graph. **Do not use the Internet for help in this portion of the assignment.**

For this assignment, you can make use of the pthreads synchronization objects and methods in the parallel implementation.

Test

You are responsible for devising and documenting a test plan that demonstrates that your code is likely correct. You should test your code and evaluate its correctness (for both implementations) before moving on to the experimental portion of the assignment. **Remember that if code is not correct, its performance is inconsequential.** Anyone can write super-fast code that gets the wrong answer.

Experiment

Unlike many of your classes up to this point, implementing a correct program is only a small part of the assignment. Once you are convinced the program is correct, you must evaluate its performance on the class server system. We are interested in performance as a function of two parameters N, the number of vertices in the graph, and T, the number of threads used in the parallel implementation. You will conduct the following experiments, and each should be documented (results in both a table and chart form) in your write ups:

1. **Parallel Overhead**: Independently run SERIAL and PARALLEL with $T = 1$ and $N \in \{16, 32, 64, 128, 256, 512, 1024\}$. Plot the speedup (i.e. ratio of SERIAL to PARALLEL) runtime for each data point. What do you observe?

2. **Parallel Speedup**: Run parallel with $N \in \{16, 32, 64, 128, 256, 512, 1024\}$ and $T \in \{2, 4, 8, 16, 32, 64\}$. For each N plot the speedup on the y-axis and T on the x-axis. What is your expected speedup, how does it compare to the measured speedup?

0.2 Design and Test Writeup

You first task for this programming assignment is to come up with a design for the parallel and serial implementations and a test plan. These two deliverables should be documented in a (typeset) write up and submitted at the beginning of class on Tuesday 4/9/2019.

The design portion should cover three aspects of the software you plan to build: (1) the modules, interfaces, assignment of tasks to threads, and invariants that will comprise the larger software system, (2) the correctness testing plan, and (3) the performance hypotheses and test plan. Your TAs and instructor should be able to read this document and feel that you have clearly thought through the project. You are responsible for making a readable/understandable document. For the first assignment you might consider the following issues:

- What data structures will your SERIAL program use internally?
- What additional data structure will your PARALLEL program use internally?
- How will work be divided up among threads? You should describe this in terms of the i,j, and k indices that form the triply nested loop in the Floyd-Warshall algorithm.
- What data (if any) must be communicated between threads?
- What synchronization (if any) is needed to ensure correctness?
- What invariants must be maintained to ensure correctness? You should think about what invariants are required for the SERIAL implementation to be correct and what (if any) additional invariants are required for the PARALLEL implementation.

The test portion should cover testing for both correctness and performance. I prefer to think about testing in terms of hypotheses and experiments. It is very important that you form hypotheses before you do experiments (in fact, it should be done before you write code). You should have an idea of what to expect from your program before evaluating it. Doing so is one of the skills that separates computer scientists from programmers.

When testing for correctness, your hypothesis should be that your program operates correctly, but that is too vague. A more specific hypothesis is that none of your programs invariants are violated for a specific set of inputs. Therefore, your test plan should refer to your invariants. Additionally, you should consider inputs or behaviors that might stress these invariants and use these as test cases. In many cases, you might want to test error conditions as well. For example, perhaps you designed your program so that it won't work with a graph that has only a single node. In that case, you might want to detect

this input and print an error message rather than output an incorrect answer. This example shows the importance of testing on both well- and mal-formed inputs.

Thus, your correctness testing plan should cover the invariants that you will stress and the inputs or behavior you will use to generate this stress.

Your performance testing should also involve hypotheses about how performance will vary as a function of various parameters. All assignments will have an experiment section which will serve as a guide for your performance testing plan. You will need to add the following: hypotheses about the expected performance for each of the experiments. For example, this assignment asks you to measure performance (and overhead) as a function of input size: what do you expect to see at small input sizes? What about large input sizes? You should justify these hypotheses with consistent reasoning. We will not grade the correctness of the hypotheses (i.e. there is no penalty for not making a good guess about performance behavior); however, we will grade the reasoning so guesses without solid thought behind them will be penalized. For later assignments, you may need to form additional hypotheses and performance tests beyond what is listed in the experiment section. Any additional tests you believe are required should be listed in your performance testing plan.

When you are running your experiments, you may find that your hypotheses are violated. It is your responsibility to determine why that is the case. For example, suppose you said one algorithm should be faster than another, but your data does not support this. You need to determine whether the problem is your implementation or some other aspect that you did not consider (or were not aware of) at design time.

0.3 Final Write-up

The final write up (to be submitted on the due date) should discuss any changes that you made to your design and why. You might change your design because we discussed a better one in class, if so attribute the new design to the class discussion (that is not a problem). You also might change your design if either your performance or correctness hypotheses are violated. The final write up should document any changes to the design and the reason, including specific tests and inputs that did not work as expected with the original design.

In addition, the final writeup should have all the data (both tables and graphs) for the listed experiments as well as some analysis. This analysis should include an evaluation of your original hypotheses about performance.

1 A Final Note on Class Responsibilities

I have found that this class represents a big leap in responsibilities for students and this was difficult for many students in previous iterations of the course. You are responsible for producing correct code that performs well, documenting this process, and demonstrating insight (through the discussion of your hypotheses and experimental results). For many of you, software design, testing, and performance evaluation will be new skills. More time will be spent on these activities than on coding. This can be frustrating for students, but these are essential skills for developing software systems.

If you feel uncertain about what is required of you please see me or your TA early for help. I want people to learn these skills because I am sure they will benefit you in the future and contribute to your success. I am happy to help anyone who comes looking for help before assignments are due. I cannot help anyone who comes to me after the fact, though, so seek help early if you are unsure.

Time management is also an essential part of this class. If you wait till the last minute to start work on these assignments, you will not do well in this class. Please start early.