Mina Vu

Assignment 2 Report

CS515 Parallel Programming

Spring 2022

**Prime Finding Programs in C++ Using Threads and OpenMP**

For assignment 2, we created six programs that worked to find prime numbers using Eratosthenes’s prime-finding algorithm. The first program runs sequentially while the rest utilizes multi-threading to execute the algorithm in parallel. Timing calculations were also added to all programs to measure the real time in milliseconds it took for each execution to run to completion. This allows for comparison of the various executions with different numbers of threads and different ranges of real numbers. I also wrote a bash script to efficiently run all programs many times and collect their relevant outputs into tabulated views. Four text files are included in this assignment’s submission showing the various outputs. Findings and analyses are below.

The sequential version mostly ran as expected with more time required for larger values of N. Timing of the program for the same values of N varies over each execution. The parallel versions unexpectedly yielded longer running times than their sequential counterpart. Both methods of threads and OpenMP presented similar increased timing. Executions of the same N value with various number of threads do not show a consistent increase in time but also defy expectation of a decreasing time trend with more execution power. This leads to my conclusion that more threads may not perform better than having a single thread do all the work. This result is very surprising to me because I did expect the threaded versions to do better with a larger range of N.

My experience with this assignment was generally good. It was interesting to program the different algorithms to solve the same problem. It was also interesting to see the timing differences and to see how they differ so widely with my hypothesis. I also got to practice writing a bash script to perform the program executions efficiently and to extract only the data that I needed for analysis from the output of each run. The school Linux server only allows for an N value of 10^6 so perhaps the parallel programs may run better with larger N values than that. I had also observed that the composite work is not distributed evenly among the workers at every run. My best explanation is that some workers lose more times than others in grabbing the mutex lock.

I did have a lot of trouble implementing the prime-par3.cpp and the prime-omp2.cpp programs. For prime-par3.cpp, I observed a lot of hanging because the master would complete its sieve prime searching and stop signaling to the workers. The algorithm stated that the master signals every time it finds a sieve prime, so once it has completed that section, the master simply waits for all worker threads to complete its work and join back. My solution to this problem was for the worker threads to check whether all sieve primes were found and all composites of those primes were handled. Then the workers will help each other out by signaling before it terminates. This seems to fix the problem.

For prime-omp2.cpp, I also had a lot of trouble working out the logic to making the threads compete for work. I originally thought of having a lead worker thread follow behind the main thread to pick up the next sieve prime that is found. This leader would then task out the composite work to another thread by using the task directive. The regular workers would “line-up” to grab the next task. After considering the algorithm specification, I decided that this method did not fully allow the workers to compete for work because they were not themselves checking the global index variable. The leader thread would also not be “working” because it is not working the composites. I had also considered have the main thread task out a sieve prime whenever it finds one. This method also seemed to have issues to me. If there are few worker threads and the master can work quickly, I thought that this method would hinder the master from work because it has to wait to task out primes to the busy workers. On the other hand, if there were many worker threads, they do not get to compete with each other for work. The solution was simple. I had just overthink the algorithm for this version.

Last note: I deleted my debug statements for the programs, but the timing is still as previously observed and analyzed. The file time5.txt shows the timing output of executions.