CSC2002S Parallel and Concurrent Computing Assignment 1 Report

By: Noah Gonsenhauser

My parallelization approach:

My parallelization approach to the assignment was to divide up the searching for the minimum point of a function to different CPU threads. This was done using the ForkJoin framework where I would create thread pool which I would invoke on a process that ran in the class. This would set the minimum search worker to 0 and maximum number to (rows\*columns\*search density) and then it would start to compute the global minimum by forking the work to another thread if the amount of work one thread would have to do exceeded the sequential cutoff which could be anything I set it to. I also setup a system to store all finders that found a local minimum that at some point became a global minimum so I could double check later if there were more than once place a global minimum could be found. I would then join all the workers up and return the minimum found between all of them until the very smallest minimum was found and return that along with all the necessary information.

Validation:

In order to validate my data, I compared my results with that of the Serial Algorithm for all cutoffs on all machines and the general consensus was that it was as accurate as the serial algorithm. The larger the dataset got the less accurate the algorithm became, even with the higher search densities. It was found that based on my data, the accuracy of the program was around 95% or higher for testing on all the datasets I used by virtue of the global minimum almost always being found and usually at the same place as the serial program. When given enough rows and columns to be accurate with the Rosenbrock function, it found the global minimum at 0 and it was always very near 1,1.

Tested Systems Information:

I tested my algorithm on my own personal desktop computer in an ubuntu VM running x86 architecture, I tested it on my MacBook air which is ARM based, and I also tested it on the university’s Nightmare server which runs an x86 based Intel(r) Xeon(r) e5620 . There were no issues in testing the algorithm in batches, however when testing different sequential cut-offs, there was an issue where the program would run out of memory on my laptop. My theory of this which will be substantiated later in the report is that this is due to ARM being a new platform and there being issues with either java, make files, the computer itself or any other plethora of issues that caused this to happen as I did not experience similar behaviour on either of the other machines. After running the program through testing, I acquired multiple sets of results on each machine based on different column, row and grid sizes along with different search densities, concluding that the sequential cutoff with the best performance on all machines with differing number of cores happened to be 10,000 finders per thread (a cutoff of 10,000). I initially implemented an optimiser in the code to figure out the best sequential cutoff, however it was determined that this didn’t always improve timing and would result in variance that was harder to compare side by side and thus was scrapped. I also implemented a finder system that saved all finders in order to test later on if multiple finders could find a local minimum for example if there was a difference between the serial and parallel program’s output.

Benchmarking:

In order to benchmark my program, I ran the Serial version of the program on all the machines I was going to test using my special testing class and retrieved all the data I needed from that, I would then run my benchmarking (testing) class on the parallelized program testing.

The amount of work a thread would do would be determined by testing to find the best sequential cutoff point by running a wide range of x and y minimums and maximums, a number of different amounts of columns and rows, and of course different search densities. All of this raw data was interpreted, and the best cutoff point was chosen before running the same set of tests on the Rosenbrock function of terrain, to determine accuracy. This was all done by a function within a separate java testing class for both the serial and parallelized versions on different machines in order to have plenty of data for more conclusive results.

The sets used for testing were:

Rows and Columns: 10,100,1000 and 5000

Grid sizes: 10, 100, 1000 and 5000, which had the maximums as positive numbers and minimums as negative numbers.

Search densities: 0.1,0.2,0.3 … 0.9,1

And sequential cutoffs (which in this case, was the number of points allocated to a thread):

10,100,1000,10000,100000,1000000

Every possible combination of the above sets was tested for the parallelized version of the program, it was tested once with the Rosenbrock function with the best cutoff on all 3 machines used for testing (an Ubuntu VM, The Nightmare Server and my MacBook air), and all machines also were tested with the serial version too.

The raw data gathered by these programs was put into a folder called “Results” being run through the make file, and would then be imported later by me into excel for interpretation. The data was output in the following format:

(first 11 lines of Parallelized results from the Cutoff of 10,000 of the Macbook)

Rows Columns Xmin Xmax Ymin Ymax Search\_Density Global\_Minimum Xpos Ypos Search\_Time

10 10 -10.0 10.0 -10.0 10.0 0.1 -32205 -8.0 6.0 1

10 10 -10.0 10.0 -10.0 10.0 0.2 -30620 -2.0 6.0 0

10 10 -10.0 10.0 -10.0 10.0 0.3 -32205 -8.0 6.0 0

10 10 -10.0 10.0 -10.0 10.0 0.4 -32205 -8.0 6.0 1

10 10 -10.0 10.0 -10.0 10.0 0.5 -32205 -8.0 6.0 0

10 10 -10.0 10.0 -10.0 10.0 0.6 -32205 -8.0 6.0 1

10 10 -10.0 10.0 -10.0 10.0 0.7 -32205 -8.0 6.0 0

10 10 -10.0 10.0 -10.0 10.0 0.8 -32205 -8.0 6.0 0

10 10 -10.0 10.0 -10.0 10.0 0.9 -32205 -8.0 6.0 0

This information is easily importable into excel to be interpreted.

Grid Speedup Graphs:

Processor Speedup Graphs:

Discussion:

The information for these graphs were retrieved by running a set of tests initially outlined in the report, and then average times were taken in order to determine the information for the speedup graphs.

In the Grid speedup graphs, The variance for each grid size is due to multiple search densities being tested for each grid size which results in varying search speeds and thus varying speedups for each grid size

The information for the Comparison speedup graph averaged the average serial time from each of the serial tests done on each machine in order to have a comparison between the difference machines.

As can be seen in the given graphs, all of the programs indeed sped up when being run in parallel instead of serially, and generally speaking more cores meant a faster program on average although less of a difference compared to a program’s own serial version.

The anomaly in the above data can be seen on the MacBook though which exhibits a strange behaviour of having more cores than both the Ubuntu Virtual machine and nightmare server, but only manages to be moderately faster than the nightmare server and actually slower than the Ubuntu Virtual machine but still the greatest improvement from the serial to parallelised program in both speedup graphs.

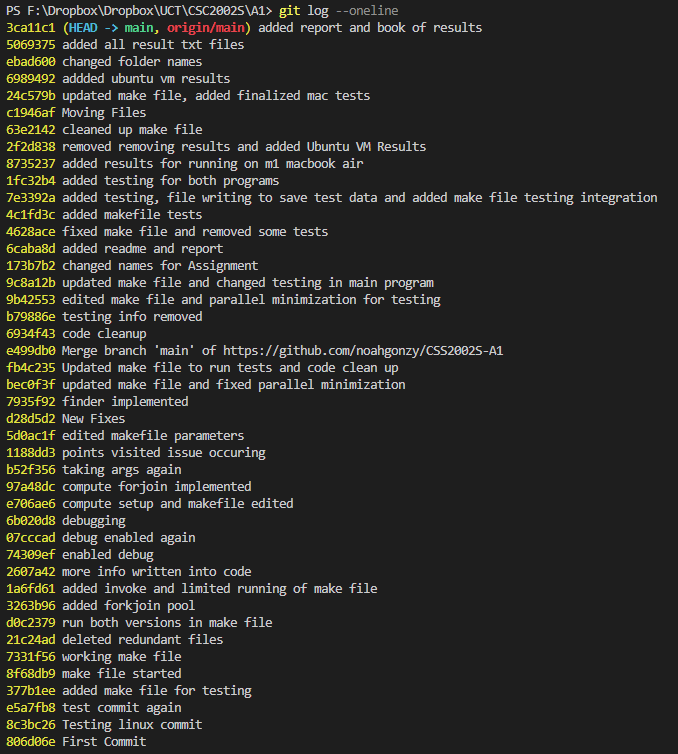
I believe the cause of this discrepancy to be the fact that the laptop I was using is running an ARM based chip (M1) which before this computer was generally only used in phones and thus is still relatively new and may have optimisation issues that x86 machines do not have due to it being the architecture of choice for many years now.

Another piece of evidence to support this theory is how the Nightmare server, despite having half the computing power of the MacBook, manage to average 1852ms to run the Serial program, while the MacBook took an average of 1805ms. However, when optimising the program and parallelising it, the MacBook sped up far more, averaging 414ms to run the program in parallel and the Nightmare server averaging 563ms which doesn’t make much sense when you think about how the Nightmare Server’s chipset is from 2010 and has half of the computing power, while the Mac’s chip is from 2019 and double said computing power. had this chip been x86 based, it should’ve been far faster in both serial and parallel performance, and also as shown by the Ubuntu virtual machine, had a lower speedup due to the faster serial program.

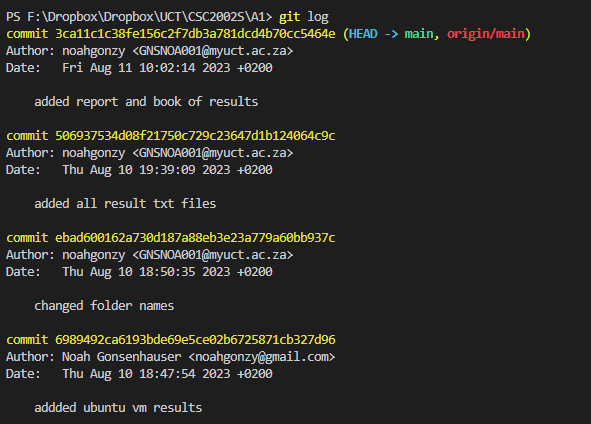
Conclusion:

GIT LOG:

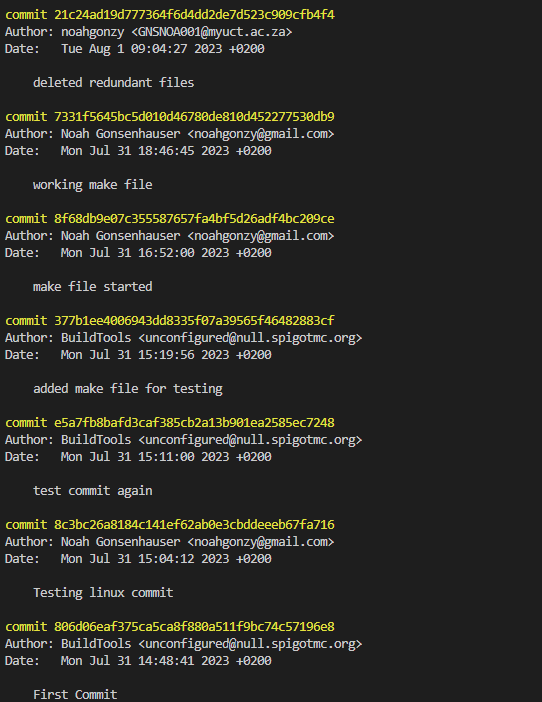
All Commits:



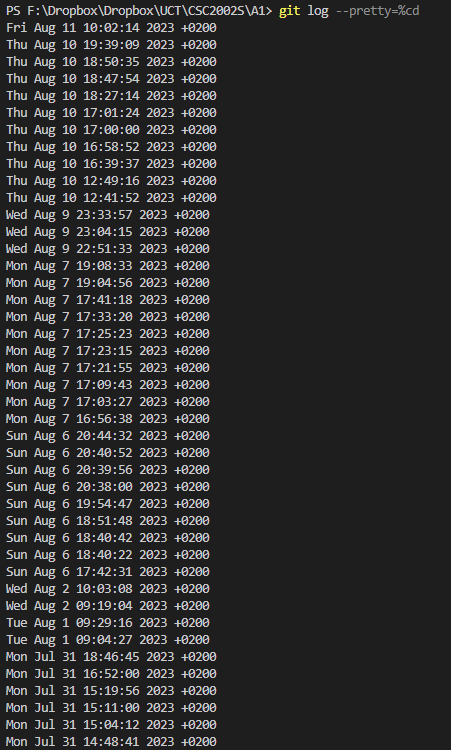
Most Recent commits:



Least Recent Commits:



All commit dates:



**All Raw Data can be found inside the “All Final Results” Folder, along with the report books of summarized data**