Project 2 Threads

Collin Lowing, Charles Travis
Systems & Networks I COP4634
October 25, 2020

Background

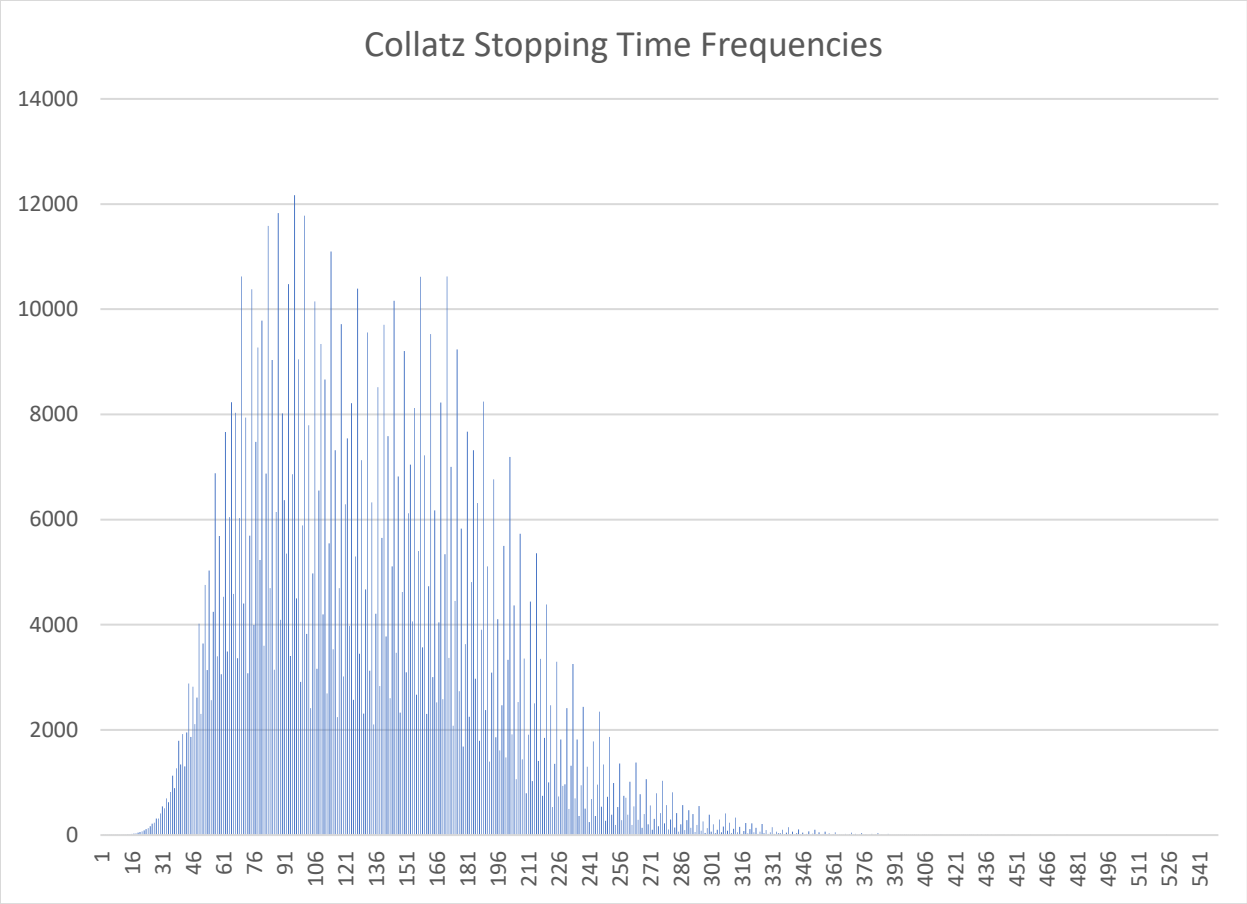
This project consists of creating a program that calculates a Collatz sequence for a range of numbers. The program runs multiple threads in parallel so that the computational work is shared, allowing the system to maximize its use of resources and not have wasted idle time while large calculations are being executed. The purpose of this study is to get a deeper understanding of threads, multi-threading and just generally getting hands on experience with the implementation of threads. The Collatz sequence for large numbers ranges involves many calculations and some of them can take many iterations of very large numbers, because of this we implemented the option to choose the range for testing and the option to choose how many threads will be created to to work together to solve the Collatz sequence of any given range.

Experiment

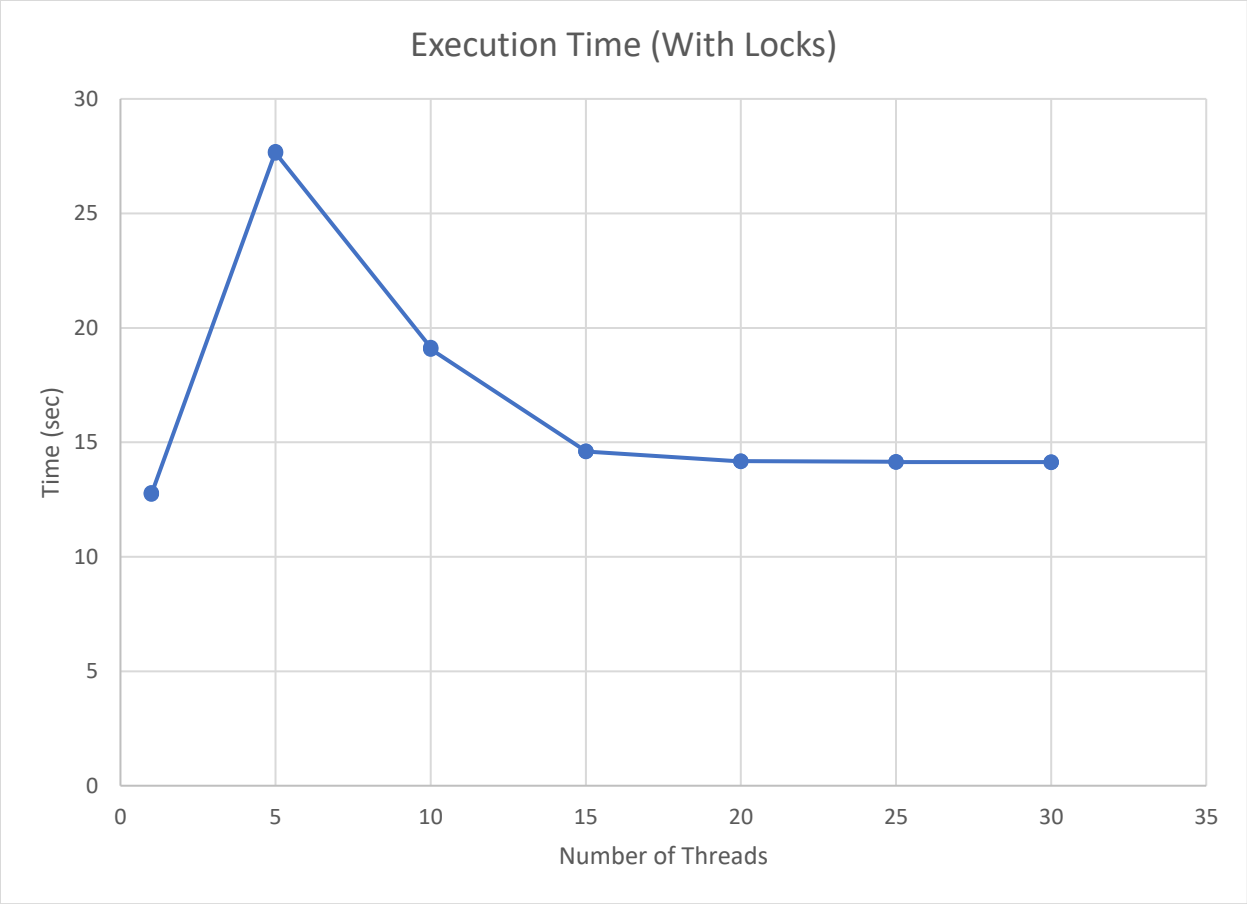
The program is executed several times using the same 1,000,000 range value. Each runtime is recorded several times for accuracy and an average is taken. The number of threads for each run of the program is increased from 1 to 30 and the difference between process thread locks and no process thread locks is recorded as well. Additionally, a histogram of the Collatz stopping time frequencies is recorded and graphed. The data is collected using a start output to the terminal and then redirected to a CSV file that can be easily opened using Excel.

Results

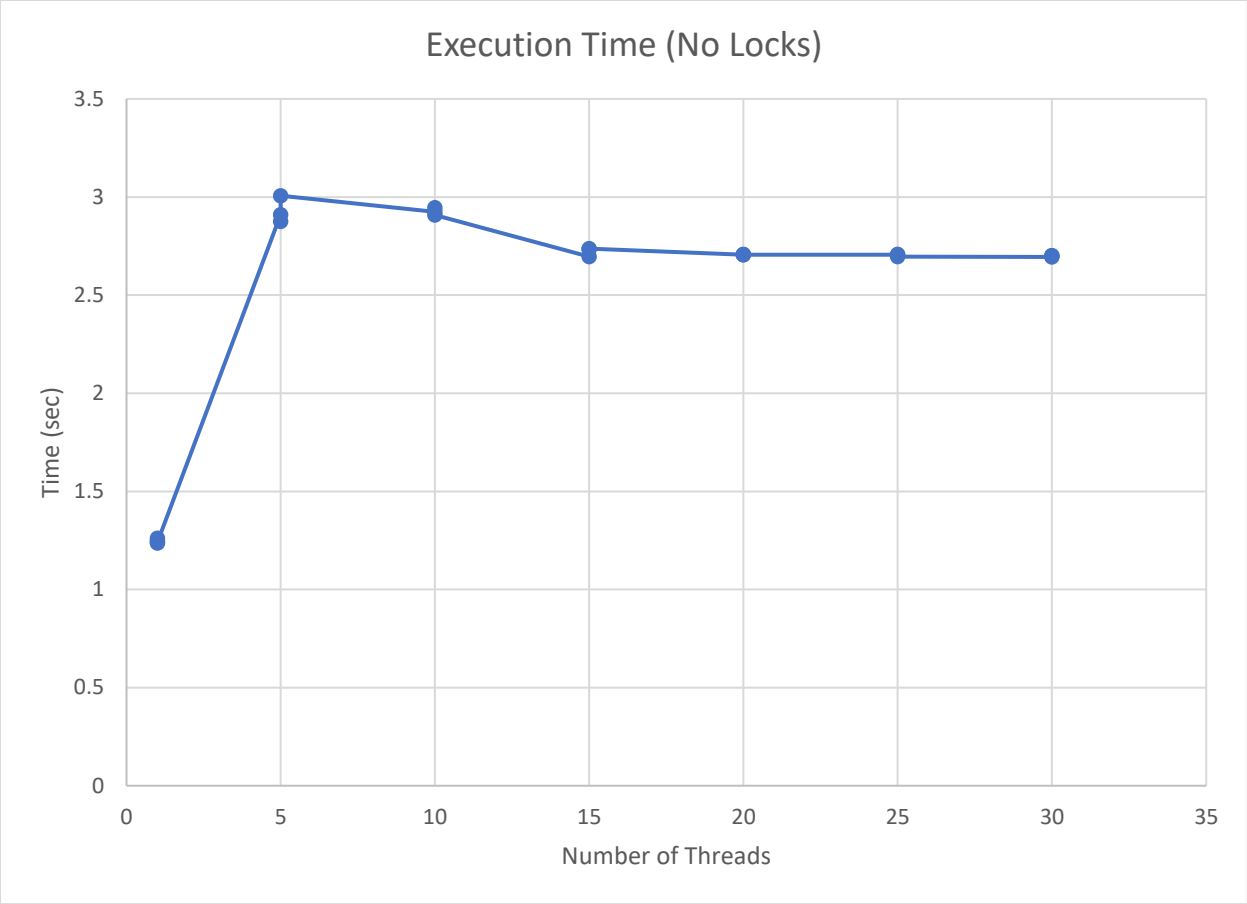
The histogram that is recorded appears to illustrate a nice bell curve pattern. Most of the values populate the 76 to 196 range for the highest Collatz stopping time frequencies.



The results from recording the execution time for a locked multi-threaded process are very interesting. It is very difficult to understand why but going from 1 thread to 5 threads shows a massive spike in execution time. Then, the process execution time slowly tapers off as more and more threads are used. The program runs much slower than without any locks.



The results from the no-lock multi-threaded process execution time experiment are equally interesting and difficult to understand. Again, the program has a significant increase in execution time going from 1 thread to 5 threads. However, the execution time only slightly decreases as the thread count increases.



Discussion

The results of this project ended up being very strange. The program our group designed had issues that could not be resolved. When running this program on a Windows based system the program would execute quickly solving a range of 100 million in a Collatz sequence in around 60 seconds with 12 threads and when increasing thread counts we were getting better results each time until around 16 threads where time gains seemed to level off, most likely due to CPU usage. The issues started when we tried to run it on multiple GNU/Linux platforms. As seen in the below graphs you can see that at first when increasing threads execution time increased by a large amount. Going from one to 5 threads increased the execution time by almost 30 seconds. This along with extremely long execution times compared to what we were getting on the Windows based system. For instance when testing it on a group members high powered Linux machine at a range of 100 million with 12 threads we received an execution time of about 600 seconds. We did not save these results and did not rerun that range again. 1 million was about all we could do in a reasonable time. After some research online we saw that many people have had this issue and it can be caused by coding issues but this code seems fairly simple and we could not find an easy way to optimize the code. One very interesting this we ran across was the idea of potential false sharing, The below quote explains it as we do not currently have the knowledge to fully understand it but it seems to be a fairly common occurrence on Linux systems when doing multi-threading.

“False Sharing:

The problem here is that each thread is accessing the result variable at adjacent memory locations. It's likely that they fall on the same cacheline so each time a thread accesses it, it will bounce the cacheline between the cores.” (Mystical)

Source:

https://stackoverflow.com/questions/17348228/code-runs-6-times-slower-with-2-threads-than-with-1

The specifications for the system used for execution testing and experimentation are as follows:

OS: Manjaro Linux (Arch based) 64 bit

CPU: Ryzen 7 2700X 8-Core

GPU: Nvidia GTX 1070

RAM: 32GB DDR4 3600MHZ

Conclusions

The experiment cannot draw any concrete conclusions about the nature of our program and why it seems to perform overall worse with multi-threading versus single threaded tasks rather than better as expected. The process of multi-threading seems to need further optimization for increases in performance to be shown. The overhead required for multi-threading seems to outweigh it’s benefit in this implementation of the technology.