Analysis of Collatz Stopping Time Number Sequences Using Multi-Threaded Programming Approach

Joshua Lemon and Levi Shaffer
COP 4634 Systems and Networks I
University of West Florida, Fall 2020

***Abstract*—By utilizing multi-threaded parallel computing techniques, Collatz Number Sequences Stopping Times can be generated in far less time than what would be necessary in a serial approach. Taking an input number and dividing it into component terms a multi-threaded computer system is able to more efficiently provide the stopping time of a Collatz sequence and output a frequency distribution of such data in a way that can be visualized as a histogram. All of this comes with a cost of needing additional computational overhead in the form of context switching, thread management, and race condition protection in the form of mutually exclusive locking techniques. Ultimately, it is important to optimize the utilization of threads to match with the number of logical processing units available to a computer system.**

***Index Terms*—Collatz, Computer Science, Multi-Threading, Number Sequences, Race Conditions, Thread Performance**

I. Introduction

As software has become more and more complex, one of the major challenges facing software and hardware engineers is optimizing software performance for many hardware platforms. One of the ways to increase performance other than increasing the cycle speed of the CPU clocks is by creating multicore CPUs that are capable of processing multiple threads in parallel to take a complex computational problem and breaking it down into multiple smaller problems to solve that then get joined together at process completion.

For the Collatz project, the goal is to utilize a multi-threaded implementation to find the stopping time of the Collatz number sequences up to a passed argument, N. The Collatz conjecture states that for a number N, if the number is even, divide by two. If the number is odd, multiply N by three and add one. The result of this piecewise calculation will then become the next number N to be evaluated. When the number to be evaluated reaches a power of two, the number will continue to be divided by two until reaching one, at which point the sequence is complete. The time complexity of this reduction from a power of two down to one is logarithmic time.

Taking the Collatz process and breaking it down into multiple threads should ideally allow for a faster computation of the Collatz number sequence, and allow us to calculate the stopping time deltas between processes that kick off different numbers of threads. Each thread will take the next number in sequence as it finishes the task at hand and tally the number of times that stopping time occurs per number N. From this tally of stopping times ranging from one and one-thousand, the numbers can then be converted into a histogram which will show the frequency that each end time is reached. In addition, the process turnaround time is calculated which will indicate how much quicker the Collatz sequence can be calculated in parallel instead of sequentially using a single-threaded implementation.

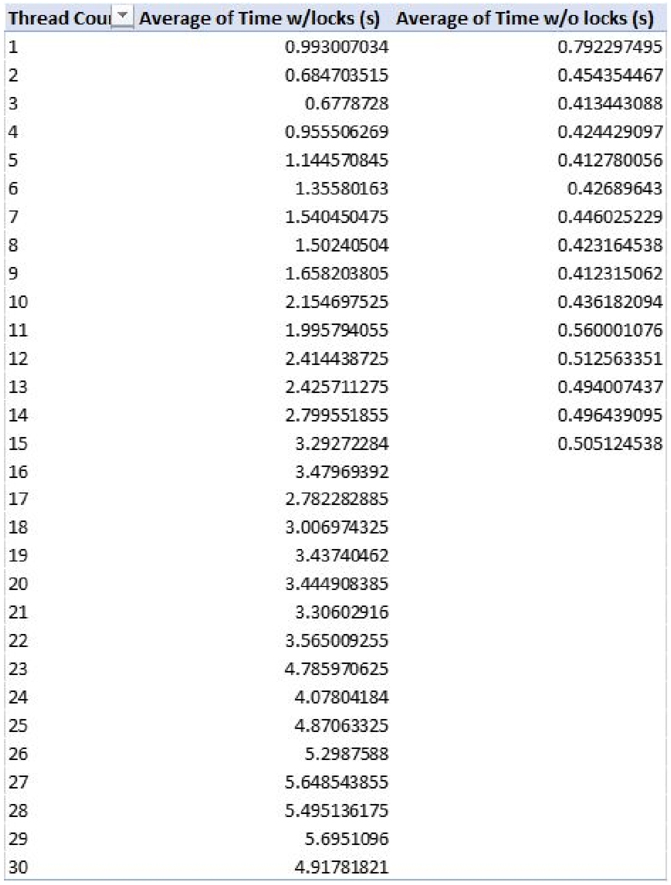
II. Implementation

The user supplies 2 arguments to the program: N, which is the range of numbers for which a Collatz sequence must be computed, and T, which is the number of threads the program will create and use to calculate the Collatz sequences from 2 up to N. There is a thread manager that creates the threads and keeps track of the current number being calculated in a counter, and the map of Collatz values to stopping times. When a thread is ready to work, the thread will try and use a mutex to lock the counter. This allows it to get the value of and increment the counter without worrying if another thread will try to increment the counter while the thread is reading the value of the counter. Once the thread has calculated the value for the Collatz conjecture based on the value of the counter, it uses another mutex lock to prevent other threads from updating the map of values. Once it has exclusive access to the map of values, the thread will update the map, and is ready to work on the next value if there is another one.

III. Runtime Testing Results

We tested this program by deciding on a constant number (in our case we chose two-million) to use for each test, and ran each test twenty times. We tested thread counts from one to thirty for locked mutex protection, and one to fifteen for unlocked mutex protection to get a wide range of multithreaded results. This resulted in six hundred total tests for the case of utilizing mutex locks as well as three hundred when leaving them unlocked to allow for race conditions. The following is the average time for each of the twenty tests to execute using from one to thirty or one to fifteen threads for locked and unlocked mutex protection.

Our test virtual machine has a Ryzen 7 3800x CPU with 4 logical threads available, running at a CPU clock speed of 4.2 GHz. This is in addition to 16GB DDR4 RAM running at 3200MHz.



Thread Cou	Average of Time w/locks (s)	Average of Time w/o locks (s)
1	0.993007034	0.792297495
2	0.684703515	0.454354467
3	0.6778728	0.413443088
4	0.955506269	0.424429097
5	1.144570845	0.412780056
6	1.35580163	0.42689643
7	1.540450475	0.446025229
8	1.50240504	0.423164538
9	1.658203805	0.412315062
10	2.154697525	0.436182094
11	1.995794055	0.560001076
12	2.414438725	0.512563351
13	2.425711275	0.494007437
14	2.799551855	0.496439095
15	3.29272284	0.505124538
16	3.47969392	
17	2.782282885	
18	3.006974325	
19	3.43740462	
20	3.444908385	
21	3.30602916	
22	3.565009255	
23	4.785970625	
24	4.07804184	
25	4.87063325	
26	5.2987588	
27	5.648543855	
28	5.495136175	
29	5.6951096	
30	4.91781821	

Fig. 1. Average execution time of Collatz sequence with N = 2,000,000 in seconds for threads from locked and unlocked mutex protection in place.

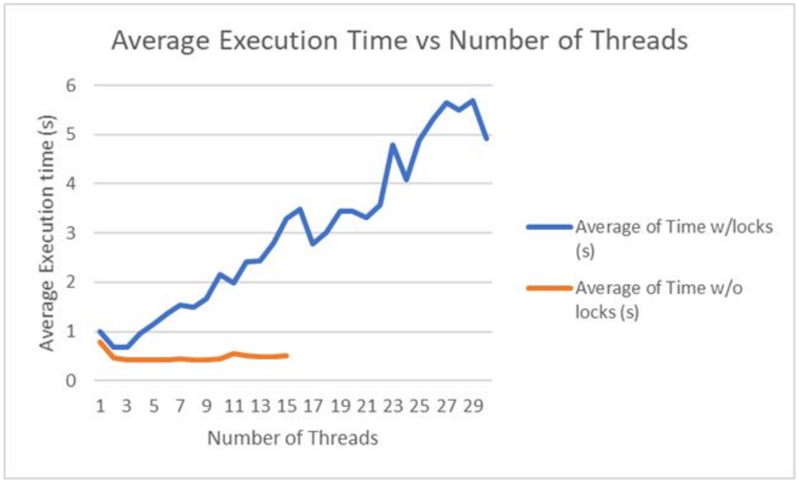


Fig. 2. Graph of Average execution time in seconds for the Collatz threads (N = 2,000,000) running with locked and unlocked mutex protection.

IV. Analysis

As evidenced by the graph in Figure 2, we see a noticeable drop in execution time for both the locked and unlocked mutex protection trial runs as the thread count increases from one and approaches the two-to-four thread range. It is at this point that the execution time begins to rise in the mutex locked process as more and more threads are spawned to solve the Collatz number sequence. This is due to the additional overhead presented with managing more threads than are necessary to optimize the computation of the problem at hand, given the limited CPU core count available on the machine we tested on. The context switch time of the CPU we believe played a large role in the increase in execution time of our Collatz sequence generator.

Another discrepancy we recorded is the fact that far more time is needed when running with the mutex lock protections enabled than when they are disabled. This is likely due to when a running thread normally encountered a lock preventing the update of a global variable, and forcing the thread to wait for the ability to proceed. In the case of no mutex locking enabled, the threads are free to continue execution without waiting for other threads to update. This results in far faster running time, while resulting in race conditions, especially at higher total thread counts.

Looking at the histogram in Figure 3, the frequency distribution of the stopping time for the Collatz sequence N = 2,000,000 the graph correlates with what we expect to see for this Collatz sequence.

V. Summary and Conclusion

There is a definite correlation between thread count and time to execute; however, in the case of this implementation of the Collatz conjecture, it is much worse to have more threads than it is to have fewer threads. The program spends so much time switching context between threads that it actually hurts performance, as the time the program takes to switch contexts is time that could have been spent calculating values. The sweet-spot seems to be less than the amount of system threads available, but more than just a few. The best performance seems to be approximately half of the available system threads, as that is where our program had the best time.

Running without data race protection with mutex locks makes the program much faster. This is because when a thread tries to lock a mutex that has already been locked, it has to sit and wait until it is able to get exclusive access to the mutex. This can lead to resource starvation if there are many threads contending over the same resource, which is another avenue for program slowdown. However, without the data race protection, all of the threads are able to access the data whenever they want at the cost of data integrity. This makes the program significantly faster.

VI. Acknowledgment

The authors would like to thank Dr. Thomas Reichherzer for his instruction in the course lectures about these topics. Dr. Reichherzer provided us with the knowledge necessary for us to setup and conduct the experiments presented in this document.

VII. Appendix

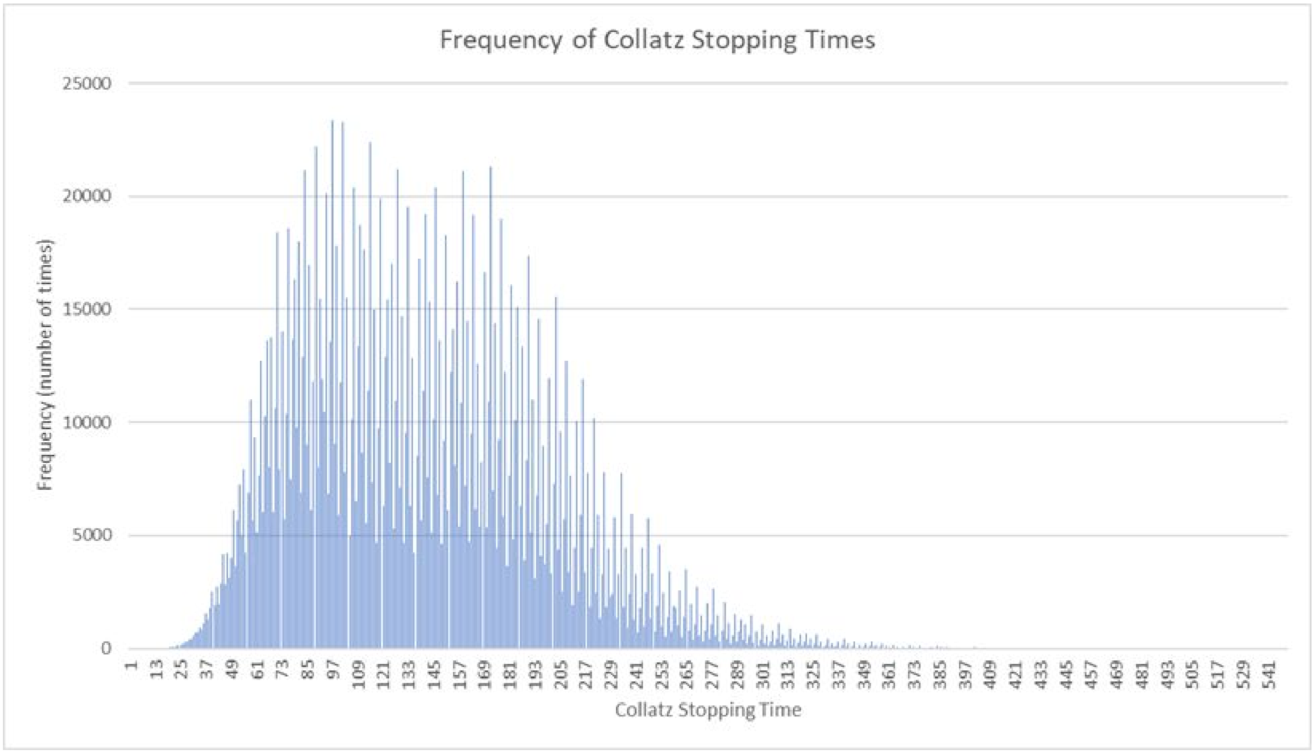


Fig. 3. Histogram of the frequency of Collatz stopping times for N = 2,000,000.