“Multi-threaded Performance and Limitation Evaluation Using Collatz Stopping Times C-Program”

By Timothy L.J. Stewart, University of West Florida

Jeffery Murphy, University of West Florida

2017

Abstract:

Multi-threaded programs in general decrease the time it takes to complete a task. However, one must consider first the number of cores available and the amount of work to be done by a thread. One should expect a performance improvement ceiling where the number of threads are greater than or equal to the number of cores, assuming the work to be done by each thread is sufficiently greater than the overhead for creating and managing each thread. In the case where the work to be done by each thread is *not* sufficiently greater than the overhead for creating and managing threads, one should expect performance degradation.

Furthermore, and as expected, without thread synchronization, tasks complete significantly faster. However, race conditions are observed heavily rendering any computation incorrect. Therefore, one must use thread synchronization techniques to prevent race conditions to benefit from any performance increases to computation realized by multi-threading.

One aspect not tested in this experiment, was the three different thread models for library threads and kernel threads, namely: one-to-one, many-to-many, many-to-one. It is assumed in this experiment each library thread has an associated kernel thread.

Problem Statement:

Does a program running with more threads always correlate positively with decreasing the time it takes to complete a task?

Hypothesis:

More threads with decrease the time it takes to complete a task, but not in all situations.

Experiment:

A multi-threaded program named mt-collatz was implemented to calculate N Collatz stoppings times with T threads, where N and T are arguments input by the user. Data was collected for all cases displayed in the graphs that follow, and gnuplot was used to programmatically graph the data collected.

In Figure 1. one can observe the general trend as you increase threads up to 3 or 4, the times to complete a task is reduced. Except in one instance where N is 50, which one could conclude, the work to create threads and manage them is greater than the work to complete the task where N is 50.

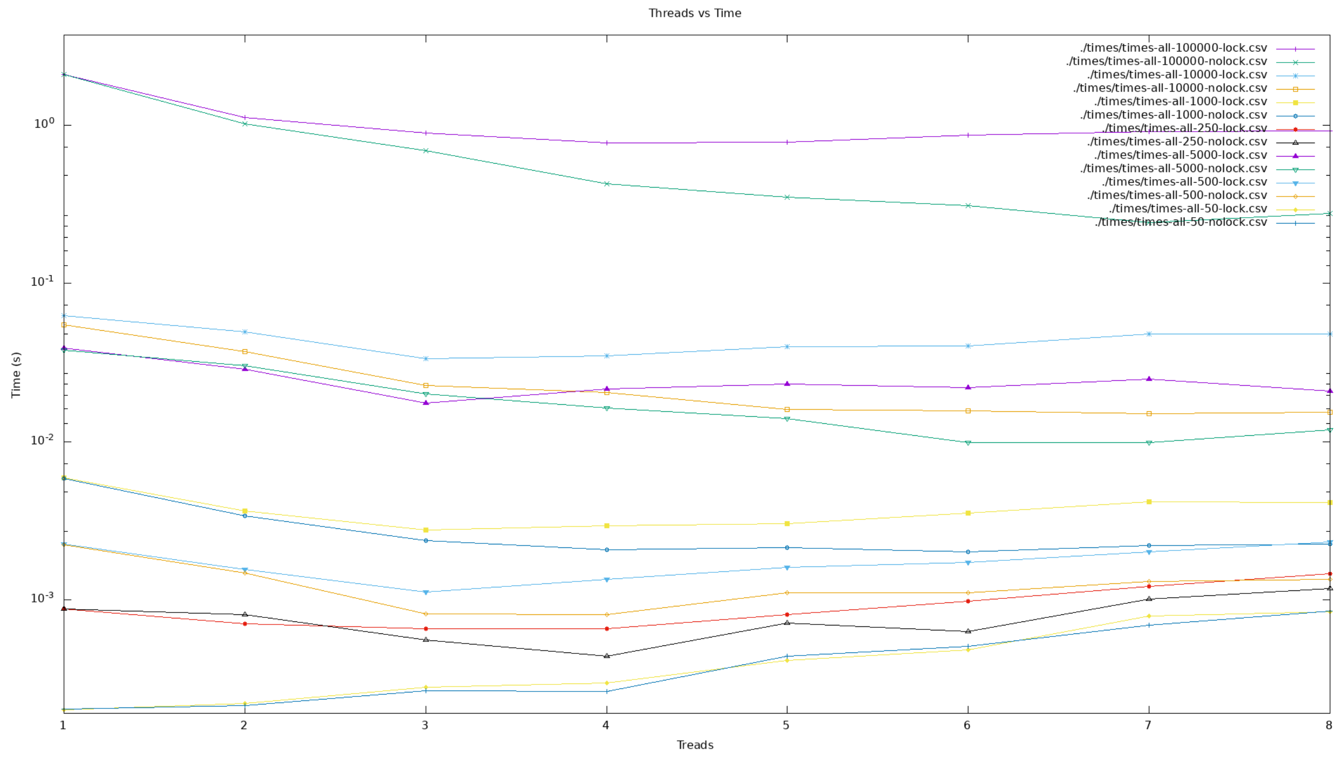

Figure 1. Threads vs. Time

Table 1. Threads vs. Time

N, Numbers to Compute On	T, Threads Used	Time to Complete
50	1	0.002606472
50	2	0.002768263
50	3	0.003285352
50	4	0.003414808
50	5	0.004302096
50	6	0.004807353
50	7	0.00675036
50	8	0.007041173
250	1	0.007250994
250	2	0.006257255
250	3	0.005966364
250	4	0.00596417
250	5	0.006885115
250	6	0.007868083
250	7	0.009124357
250	8	0.010411276
500	1	0.014116835
500	2	0.010881625
500	3	0.008658021
500	4	0.00982571
500	5	0.011104558
500	6	0.011678247
500	7	0.013033762
500	8	0.014383747
1000	1	0.027671125
1000	2	0.01977445
1000	3	0.016279938
1000	4	0.017017864
1000	5	0.017315531
1000	6	0.019273559
1000	7	0.021577084
1000	8	0.02156621
5000	1	0.103754595
5000	2	0.083371006
5000	3	0.058985304
5000	4	0.06809409
5000	5	0.071685806
5000	6	0.069106556
5000	7	0.075680301
5000	8	0.066707052
100000	1	1.679028988
100000	2	1.08254838
100000	3	0.923686504
100000	4	0.839252234
100000	5	0.844363928
100000	6	0.906730831
100000	7	0.939739764
100000	8	0.949224651

In Figure 2. one can observe, that when mutex locks are used for thread synchronization, no race conditions are experienced. This is determined by the number of times any number n is computed on is strictly once. Comparing this result to Figure 3. And Figure 4. where one can observe race conditions and frequency on said conditions when no lock is used. Some number's n, were computed on up to 5 times and some as low as 0.

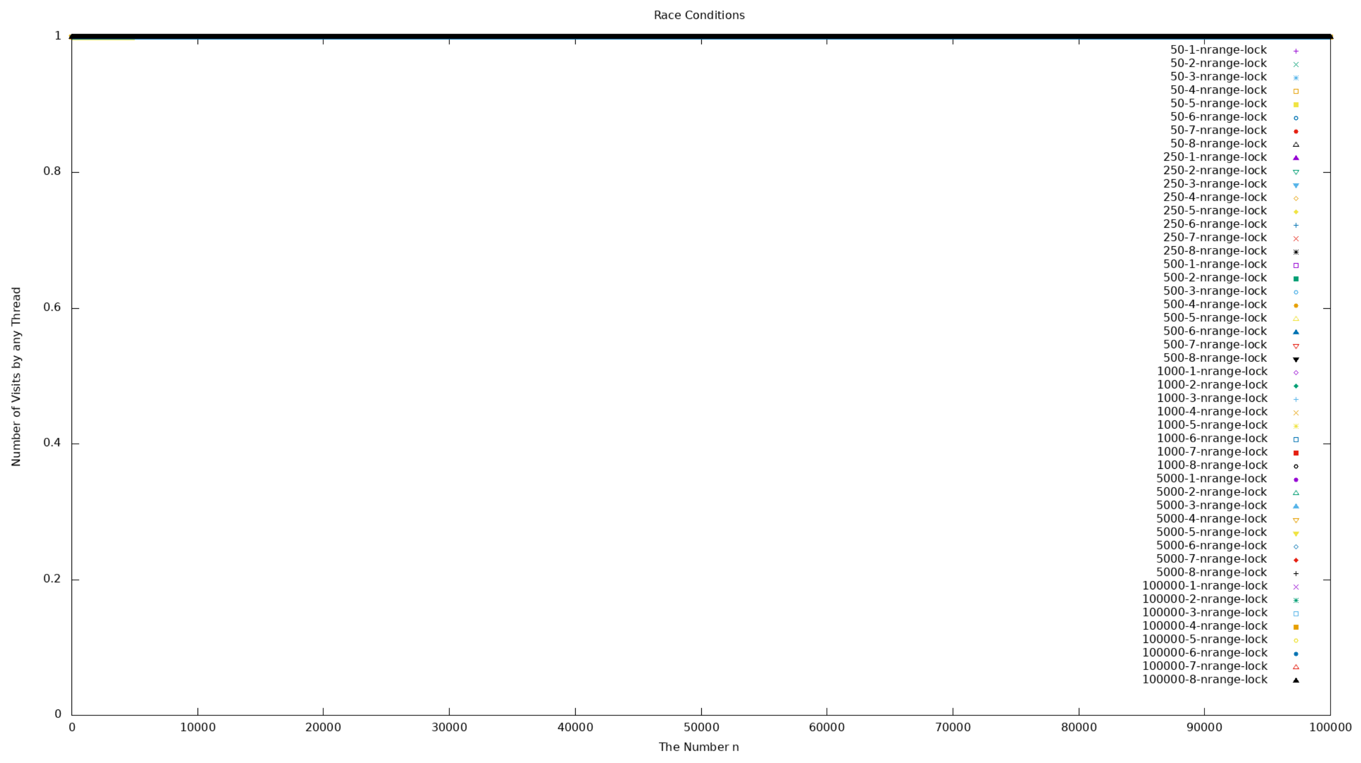

Figure 2. Race Conditions for Synchronized Threads

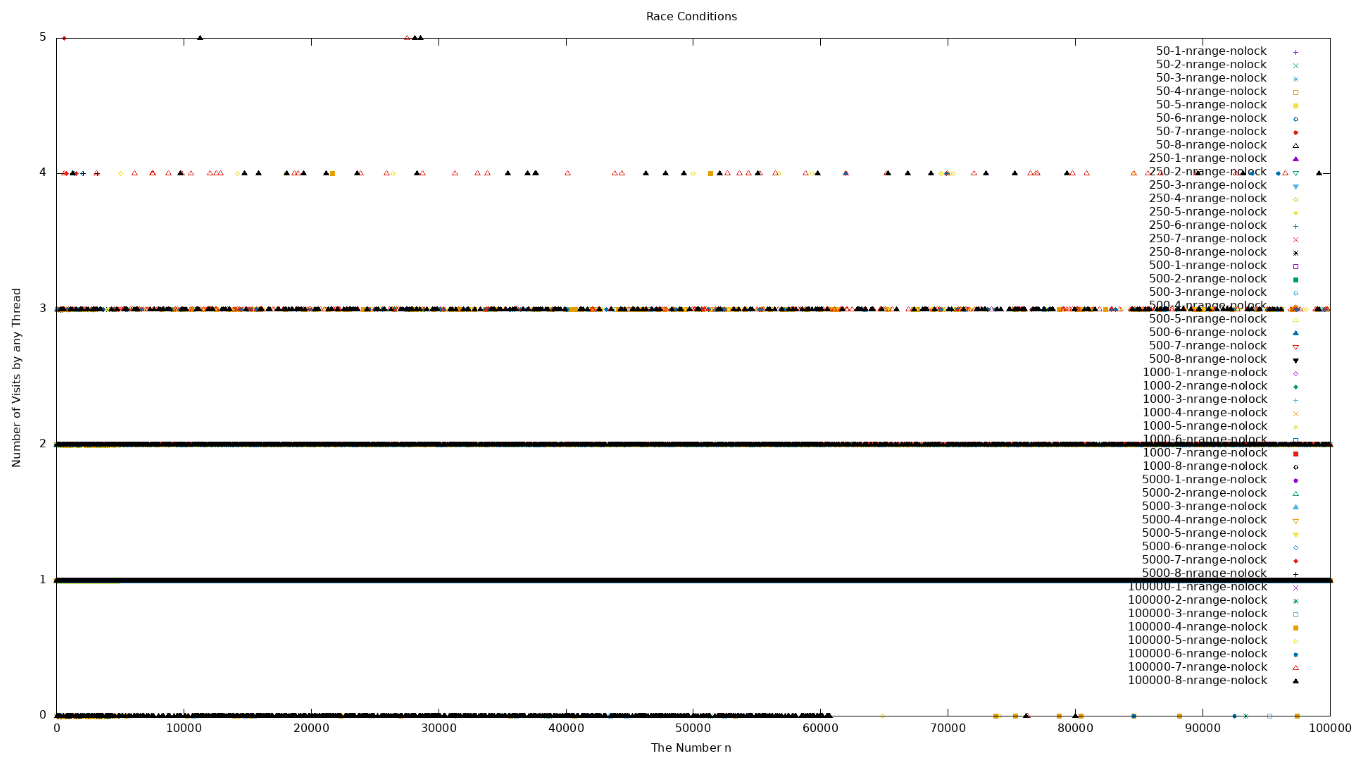

Figure 3. Race Conditions for Non-Synchronized Threads

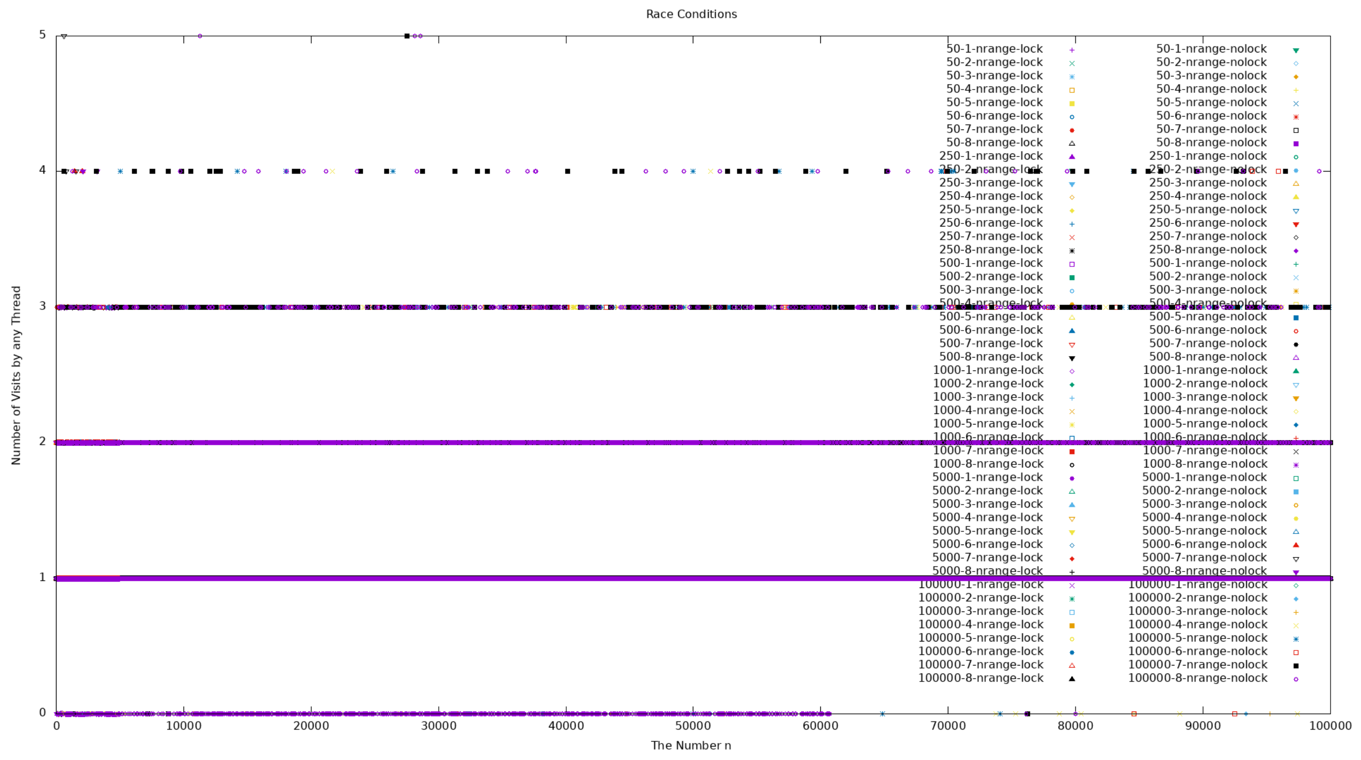

Figure 4. Race Conditions for Non-Synchronized and Synchronized Threads

In Figure 5. through Figure 7. one can observe the histogram plots for the collatz stopping times for locks and no locks. Correlating these result with the race conditions observed in Figure 4. one can reconcile the contrast observed between Figure 6. where no locks were implemented and Figure 5. where locks were implemented. The superimposed plots can be observed in Figure 7.

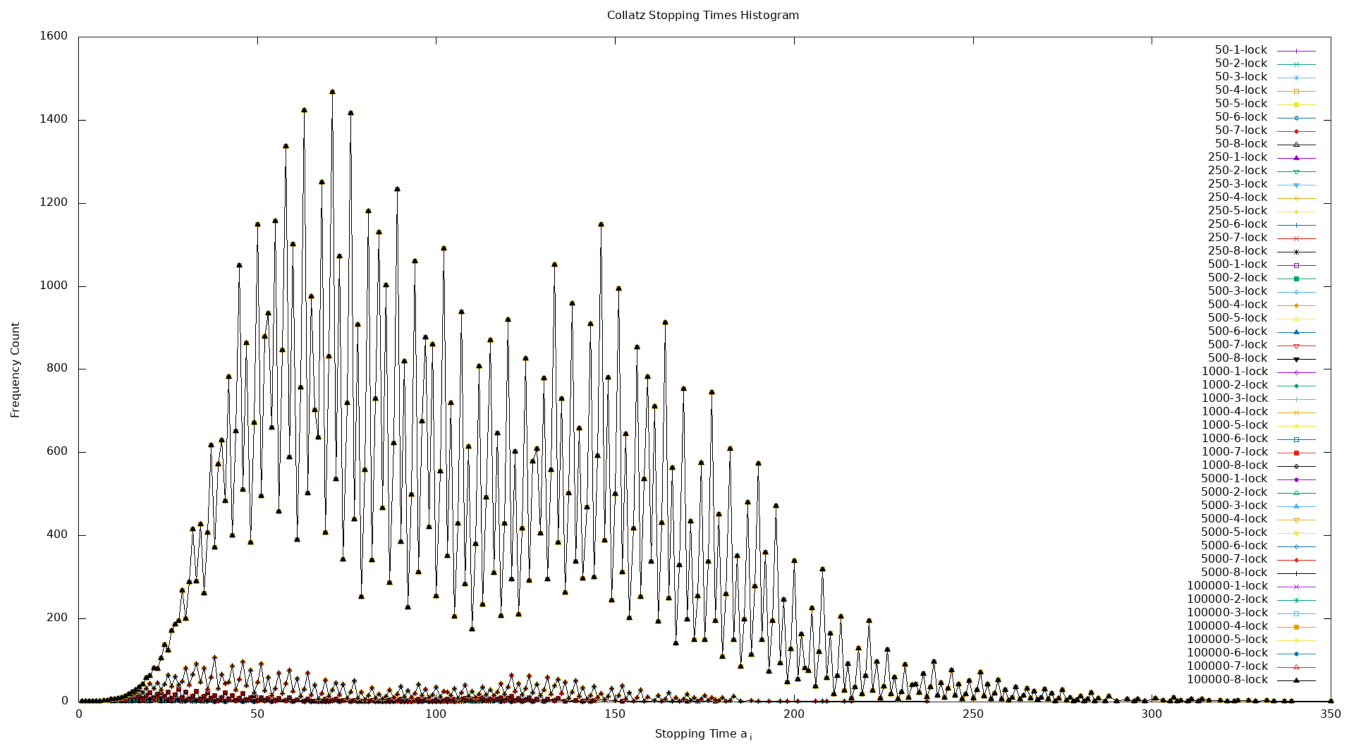

Figure 5. Collatz Stopping Times for Synchronized Threads

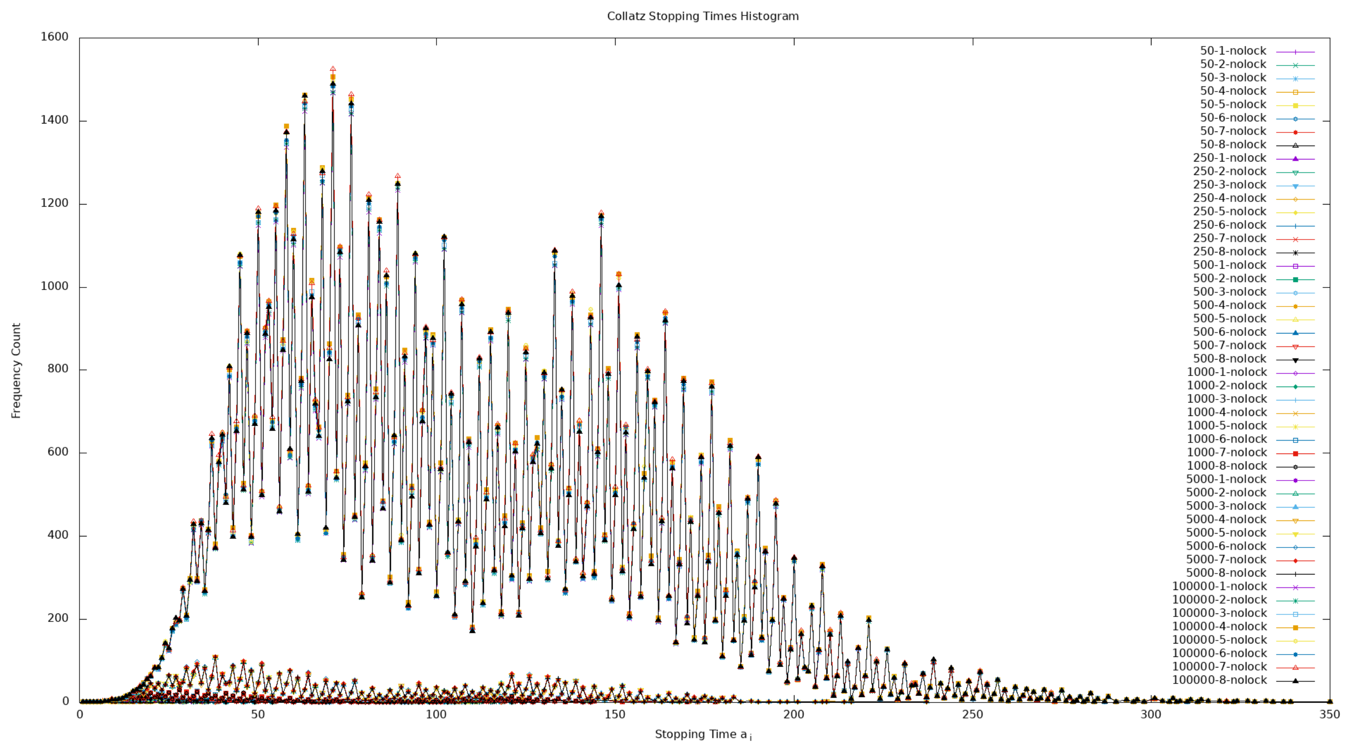

Figure 6. Collatz Stopping Times for Non-Synchronized Threads

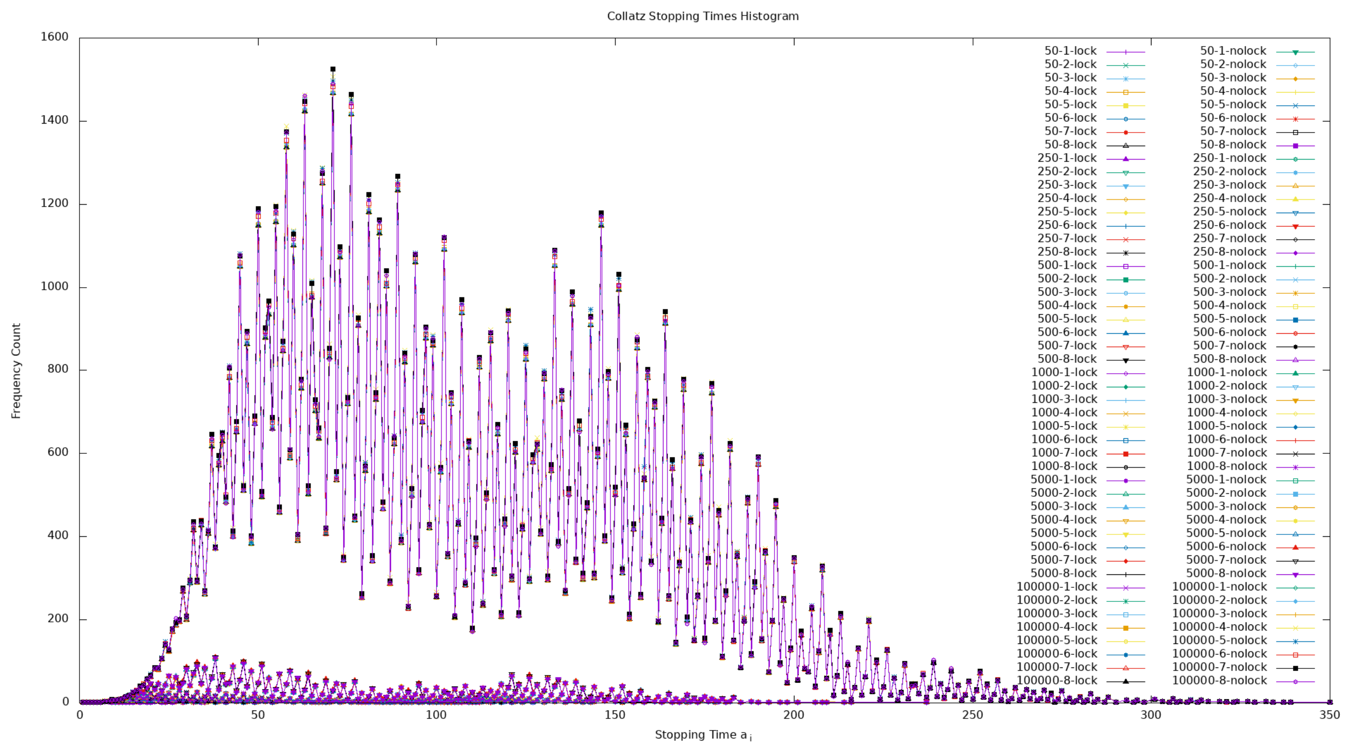

Figure 7. Collatz Stopping Times for Non-Synchronized and Synchronized Threads

In Figure 8. for numberphila sake, one can observe the characteristics of the collatz stopping times sequence in reference to each number n the collatz times were computed on. In this case N was chosen to be 500. The result is, one could conjecture, a periodic waveform.

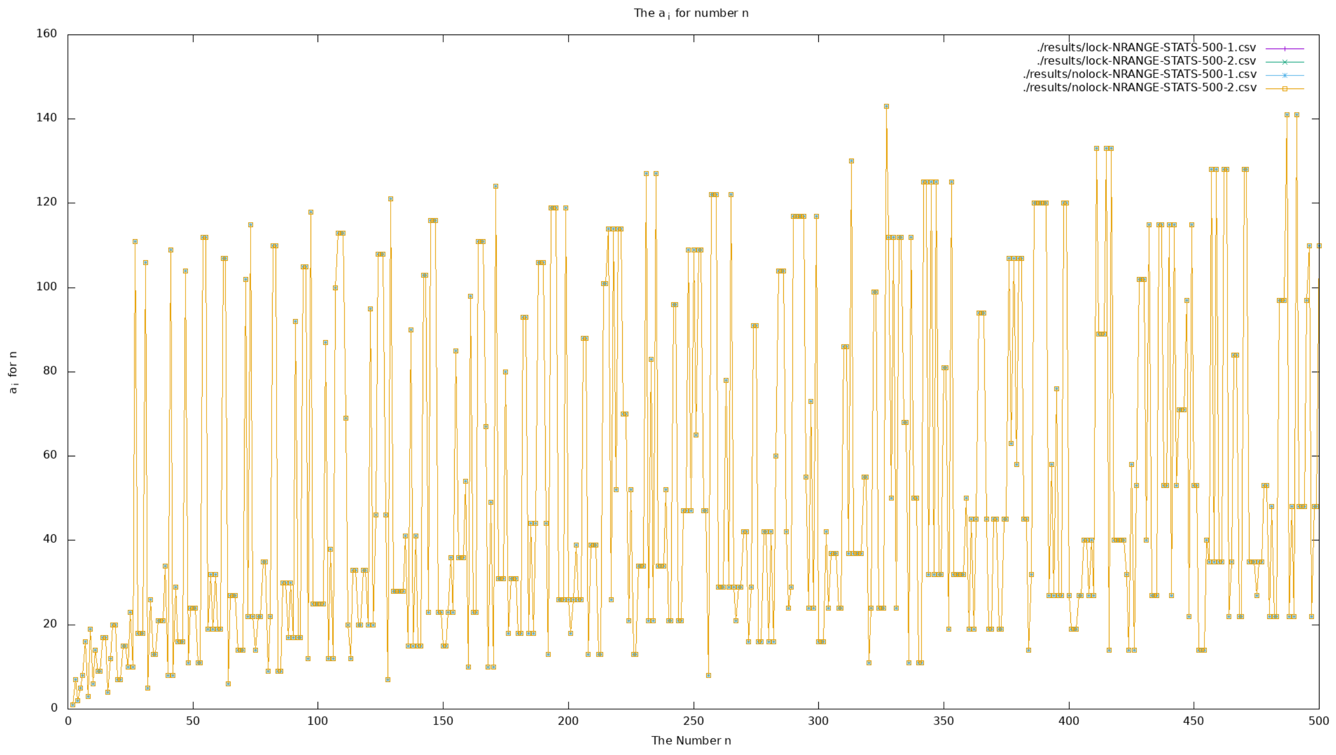

Figure 8. Collatz Stopping Time for Each Number n from 0-500

Conclusion:

Multi-threaded programs in general decrease the time it takes to complete a task. However, one must consider first the number of cores available and the amount of work to be done by a thread. One should expect a performance improvement ceiling where the number of threads are greater than or equal to the number of cores, assuming the work to be done by each thread is sufficiently greater than the overhead for creating and managing each thread. In the case where the work to be done by each thread is *not* sufficiently greater than the overhead for creating and managing threads, one should expect performance degradation.

Furthermore, and as expected, without thread synchronization, tasks complete significantly faster. However, race conditions are observed heavily rendering any computation incorrect. Therefore, one must use thread synchronization techniques to prevent race conditions to benefit from any performance increases to computation realized by multi-threading.

One aspect not tested in this experiment, was the three different thread models for library threads and kernel threads, namely: one-to-one, many-to-many, many-to-one. It is assumed in this experiment each library thread has an associated kernel thread.