Project 1: Multi-Threaded Collatz Stopping Time Generator

Ricky Gal & Emily Hightower

COP5518

July 12, 2022

Background

The Collatz Problem is a mathematical question which asks if repeating certain arithmetic operations will result in all positive integers transforming into 1 [1]. The problem uses sequences of integers and calculates a stopping time for each. In this project, this mathematical problem is used as the basis for exploring how multithreading can improve performance in a simple Java program.

Experiment

A Java class, MTCollatz, was created to be the driver program for the project. It checks the validity of the user's arguments and calls class Collatz which handles the tasks and multithreading.

On the UWF CS server, the system-wide limit for number of threads is 514727 [2]. However, each user is restricted to 32 processes each [3]. This restricts the number of threads that can be used in this project. We frequently ran into OutOfMemoryErrors when running more than 4 threads on the UWF CS server. Therefore, for the above reasons and due to high load on the server, the number of threads was limited in this experiment to a maximum of 2.

The Java class files were compiled on the UWF server using command `javac`. Next, a series of experiments were run including varying numbers of calculations and threads with locks enabled/disabled. When locks are enabled, they should prevent a variable from being accessed by multiple threads at once, and therefore prevent a race condition. When locks are disabled, race conditions may occur.

The experiment setup is outlined below.

For the duration of this experiment, values "high", "medium", and "low" were used as shorthand for the number of calculations performed. High refers to 1,000,000 calculations; medium refers to 100,000 calculations; and low refers to 1,000 calculations.

Part 1: Using Locks

Six runs were tested using locks (i.e., not using the "`-nolocks`" argument). A high number of calculations were run with 2 and 1 threads; a medium number of calculations with 2 and 1 threads; and a low number of calculations with 2 and 1 threads.

Part 2: No Locks

Six runs were tested without locks (i.e., using the "`-nolocks`" argument). A high number of calculations were run with 2 and 1 threads; a medium number of calculations with 2 and 1 threads; and a low number of calculations with 2 and 1 threads.

The data for all experiments were collected in text files on the UWF server, then imported to Microsoft Excel for visualization. The histogram text files followed the naming scheme: [num_calculations]_[num_threads]_output.txt.

Results

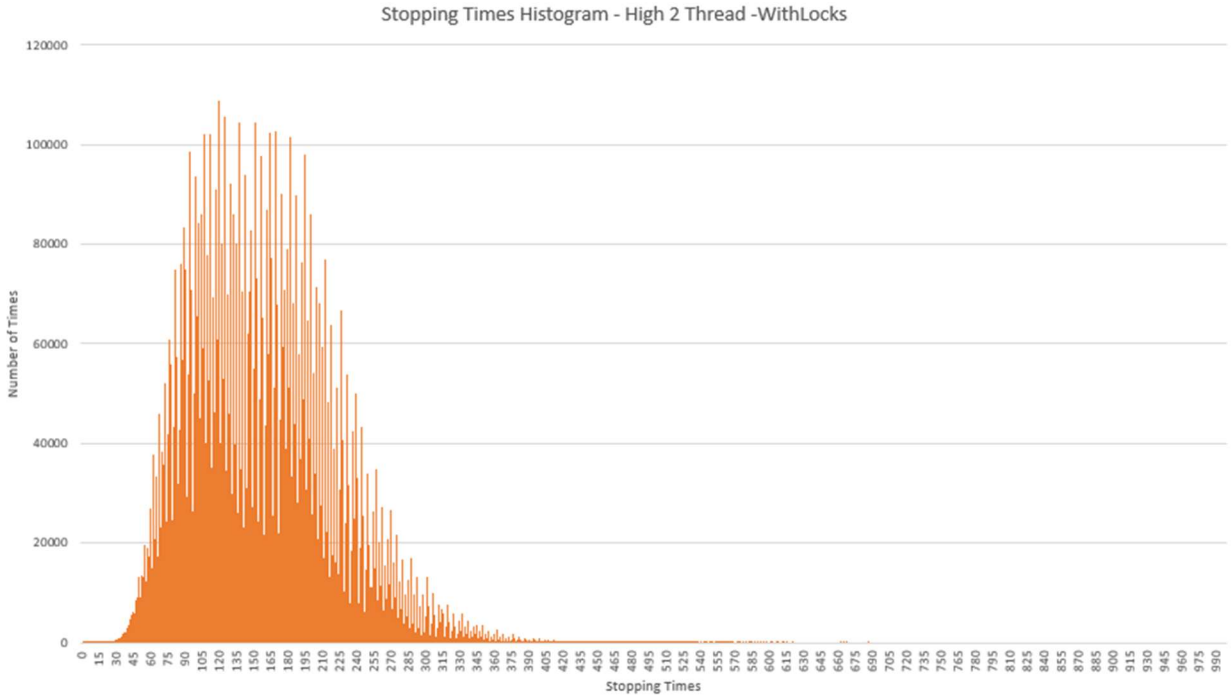
Limitations in Microsoft Excel's table creation resulted in longer datasets (medium and high) being truncated to around 10,000 rows for visualization. However, this should not affect the visualization of the results. The stopping time values past 10,000 rows are almost certainly 0 since the stopping times are so heavily weighted to the lower values.

Part 1: Using Locks

A high number of calculations were run with 2 and 1 threads. The high number of calculations took 4087.0 milliseconds with 2 threads, and 4375.0 milliseconds with 1 thread. The improved speed is expected as a benefit of multithreading when calculations can be shared between threads.

Table 1 shows the histogram for the 2 and 1 thread experiments with a high number of calculations.

Table



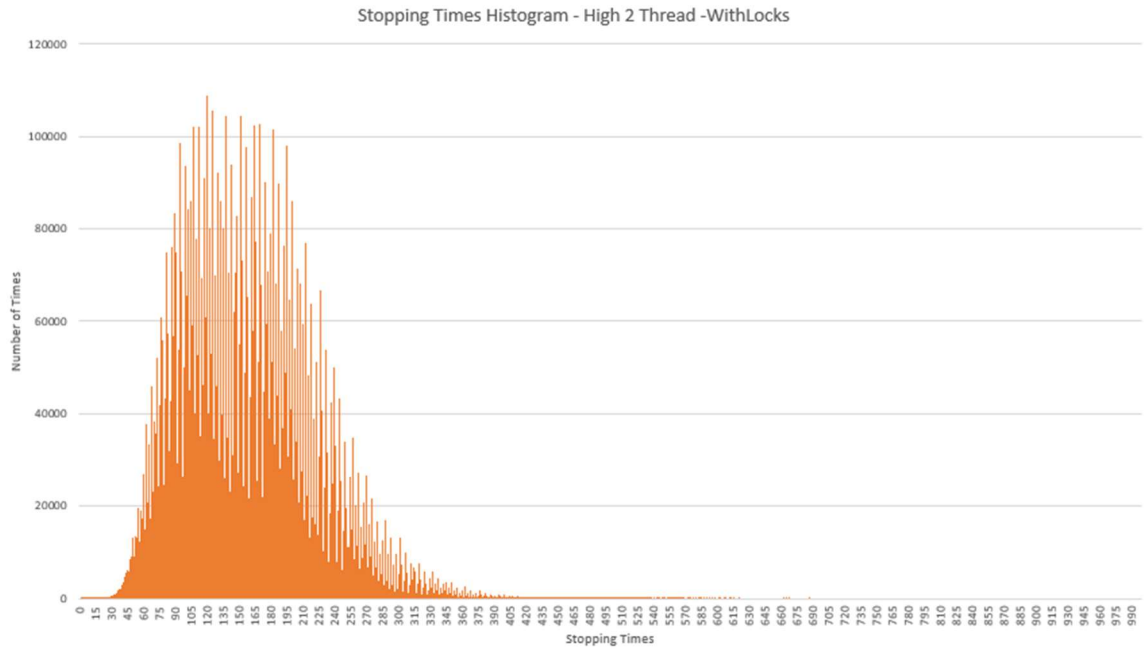
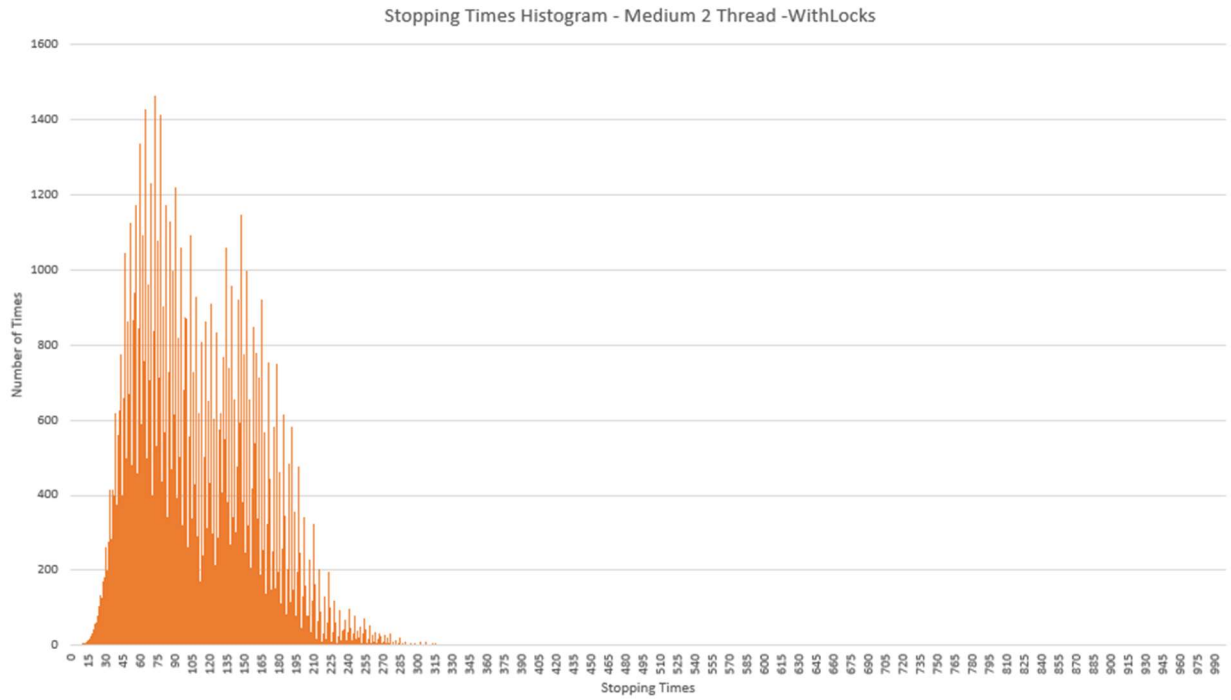


Table 2 shows the histogram for the 2 and 1 thread experiments with a medium number of calculations. The medium number of calculations took 55 milliseconds with 2 threads, and 41 milliseconds with 1 thread.

Table 2:



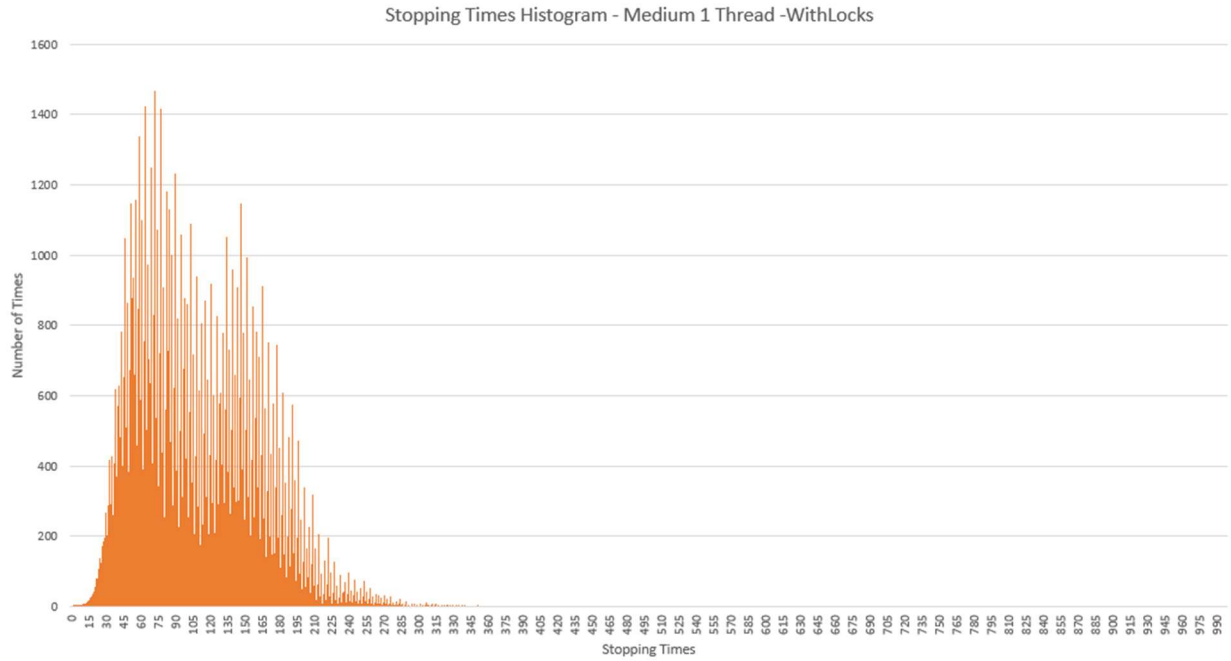
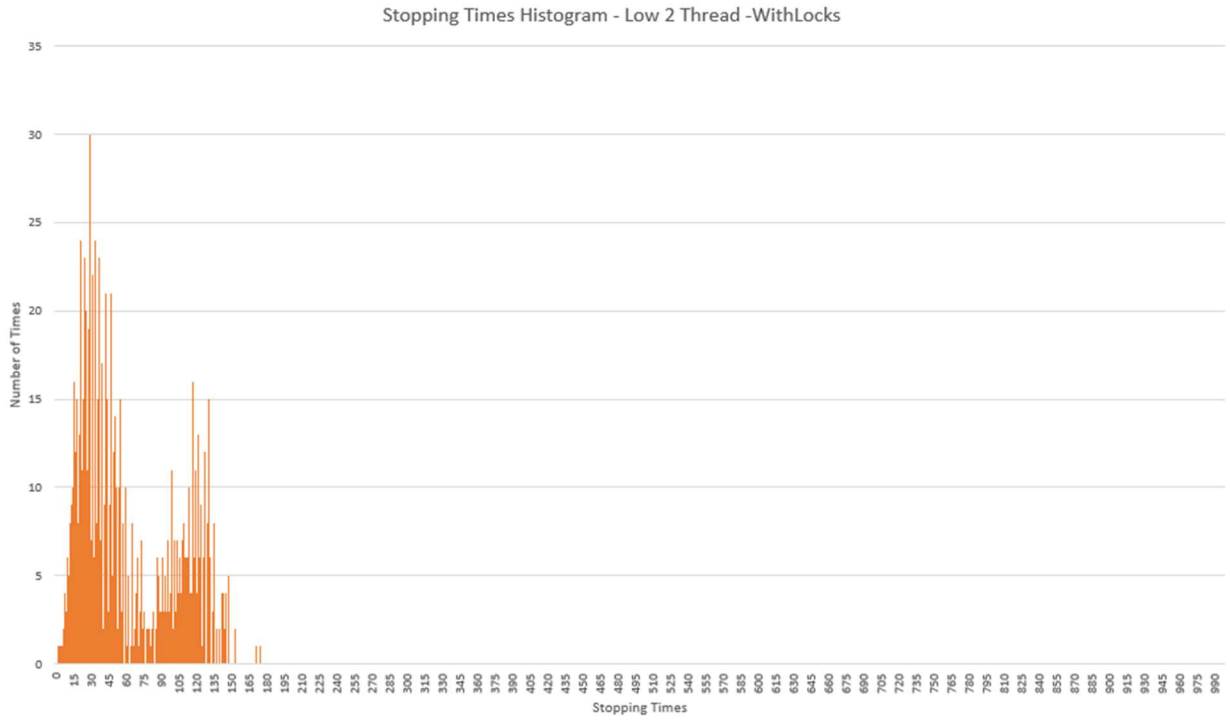


Table 3 shows the histogram for the 2 and 1 thread experiments with a low number of calculations. The low number of calculations took 3 milliseconds with 2 threads, and 4 milliseconds with 1 thread.

Table 3:

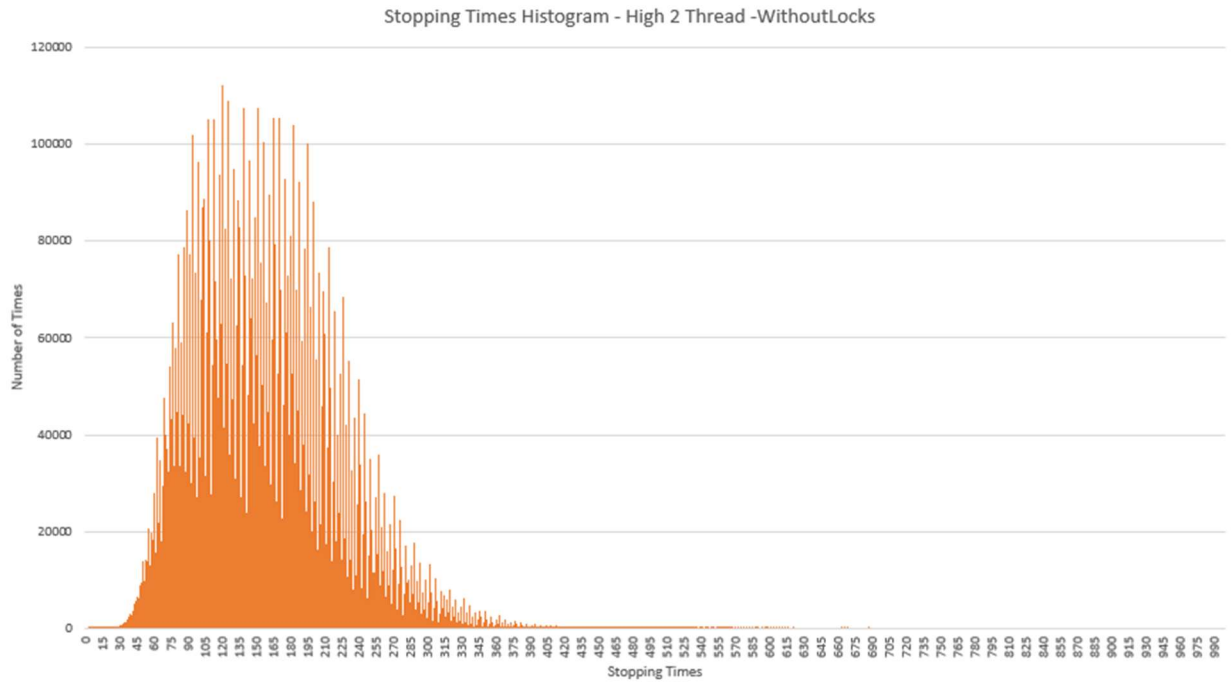


Part 2: No Locks

A high number of calculations were run with 2 and 1 threads with locks disabled. The high number of calculations took 2892 milliseconds with 2 threads, and 4824 milliseconds with 1 thread.

Table 4 shows the histogram for the 2 and 1 thread experiments with a high number of calculations.

Table 4:



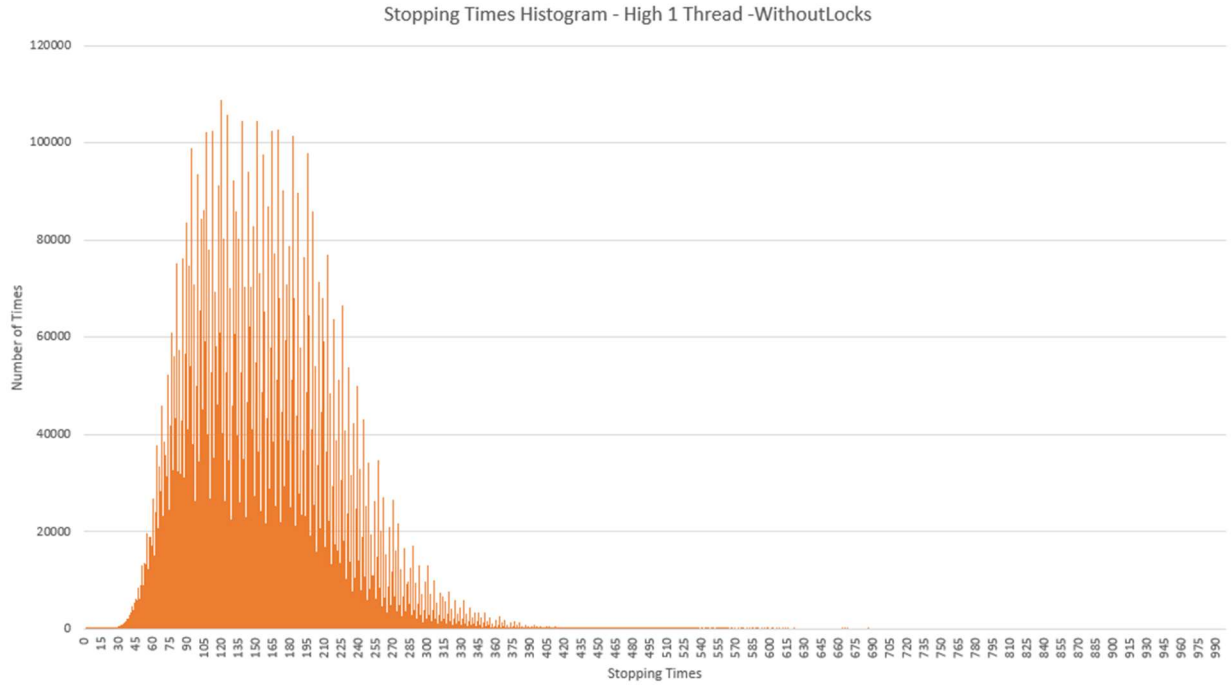
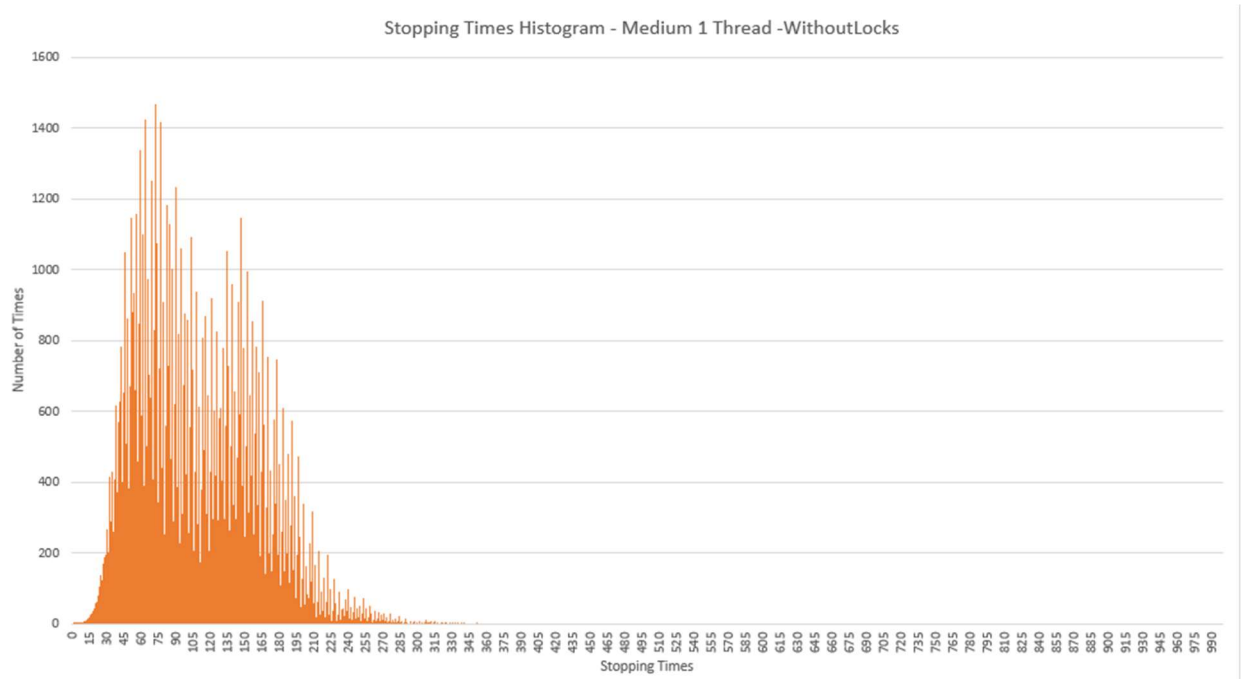


Table 5 shows the histogram for the 2 and 1 thread experiments with a medium number of calculations. The medium number of calculations took 29 seconds with 2 threads, and 40 seconds with 1 thread.

Table 5:



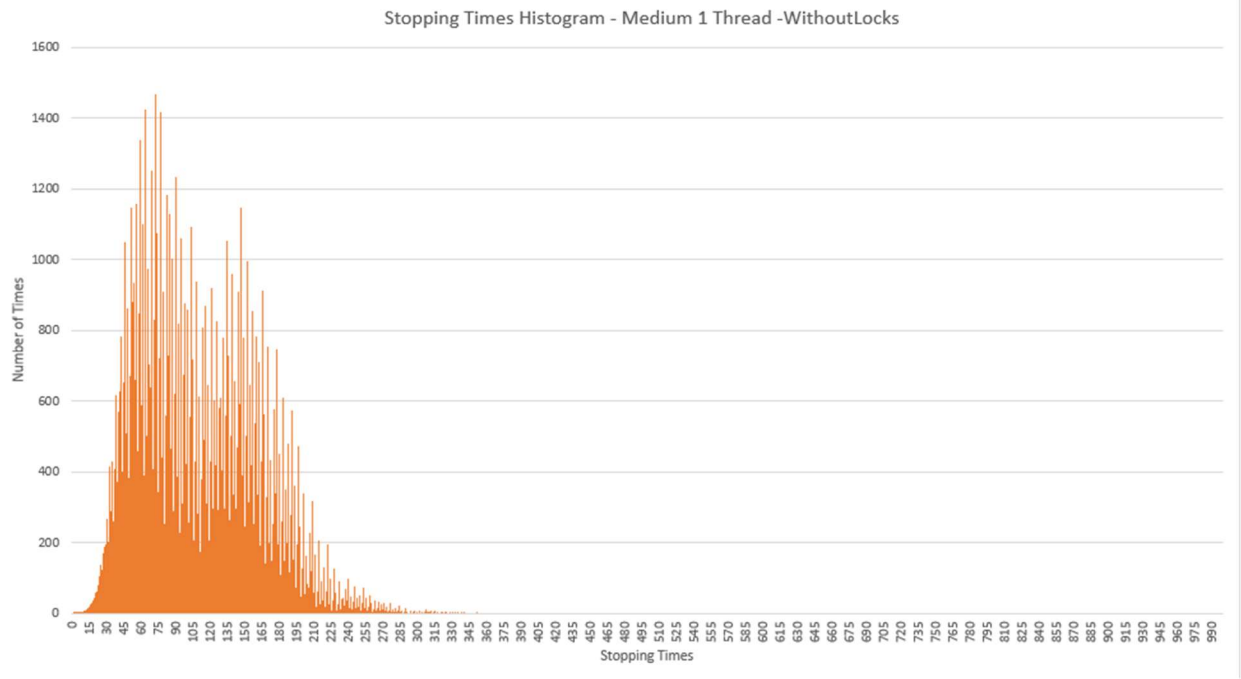
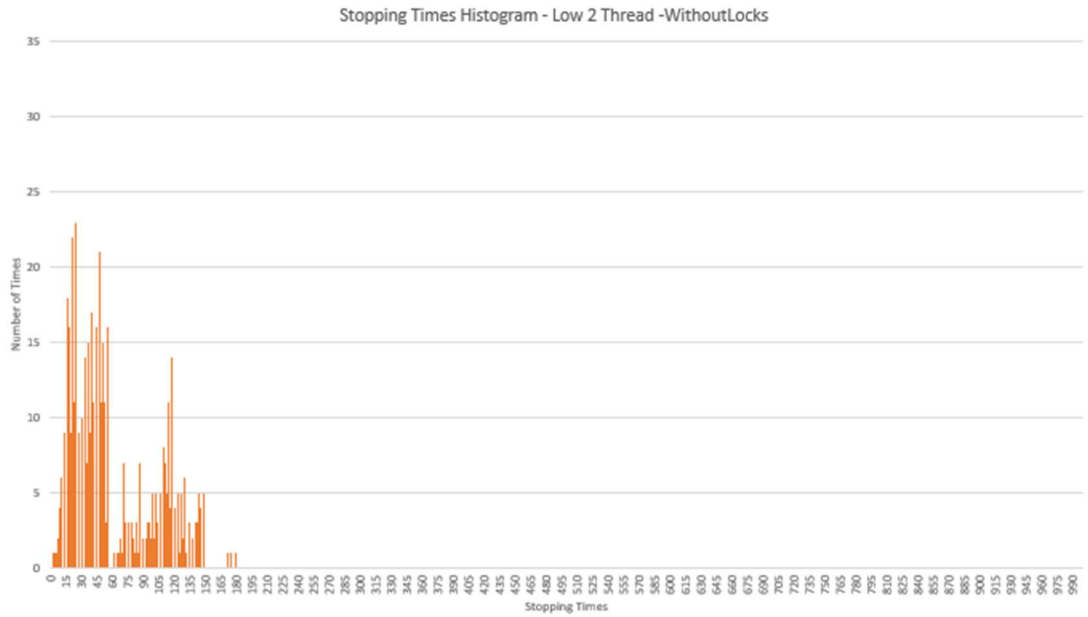
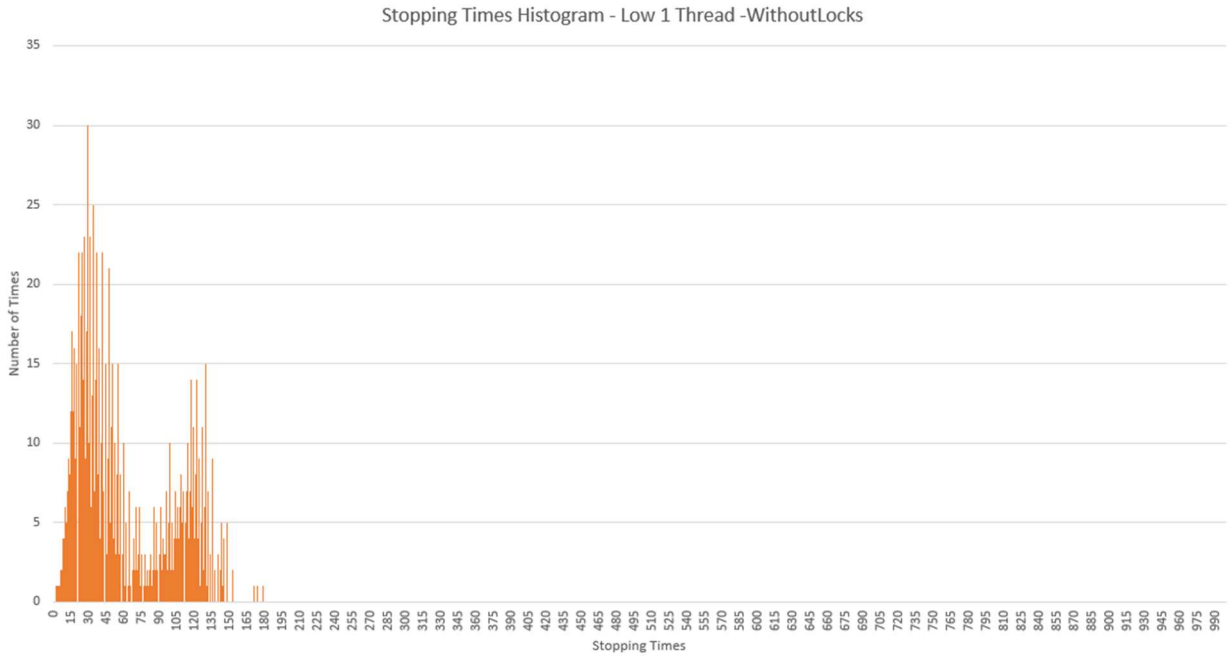


Table 6 shows the histogram for the 2 and 1 thread experiments with a low number of calculations. The low number of calculations took 3 milliseconds with 2 threads and 3 milliseconds with 1 thread.

Table 6:





Discussion

Part 1: Using Locks

The improved speed with 2 threads for a high number of calculations is expected as a benefit of multithreading when calculations can be shared between threads. As the number of calculations decreased, it appeared that multithreading diminished in value and even became a liability. This is likely due to the overhead required for multithreading, which takes enough time to offset any benefits gained

with a medium or low number of calculations. Although the multithreading time was technically lower with 2 threads vs. 1 thread, the time difference (1 millisecond) is so small as to probably not be useful.

Part 2: Without Locks

Without using locks, the time advantage with multithreading was much more significant. In particular, the high number of calculations took about half the time with 2 threads as with 1 thread. However, this is likely at the expense of fine-grained accuracy as race conditions are almost certainly occurring.

Both with and without locks, it is clear that all of the histograms look very similar. More detail is visible with lower number of calculations, however this is probably due to the smaller horizontal axis scale. This suggests that despite the presence of race conditions, the big-picture results are generally similar between the lock and no-lock experiments. There are certainly minor differences between the outcomes, however the overall charts are roughly the same.

Conclusions

Our experiment showed that when it is important to avoid race conditions, multithreading gives a slight advantage over single threading. However, over a high number of averages it may not be as important to avoid race conditions. In those cases, allowing race conditions can give an extremely significant time benefit. It appears that there is a trade-off between fine-grained accuracy and speed gained by using multithreading.

Sources

[1] “Collatz Conjecture.” *Wikipedia*, Wikimedia Foundation, 30 June 2022, https://en.wikipedia.org/wiki/Collatz_conjecture.

[2] Value found in /proc/sys/kernel/threads-max.

[3] Value found by viewing the max user processes with the `ulimit` command.