REPORT ON
MULTI-THREADED COLLATZ STOPPING TIME

By Dae Sung & Mukesh Rathore

Abstract

The Idea of the project is to explore the multi-threaded programs and how they can improve the performance of our computation.

Description

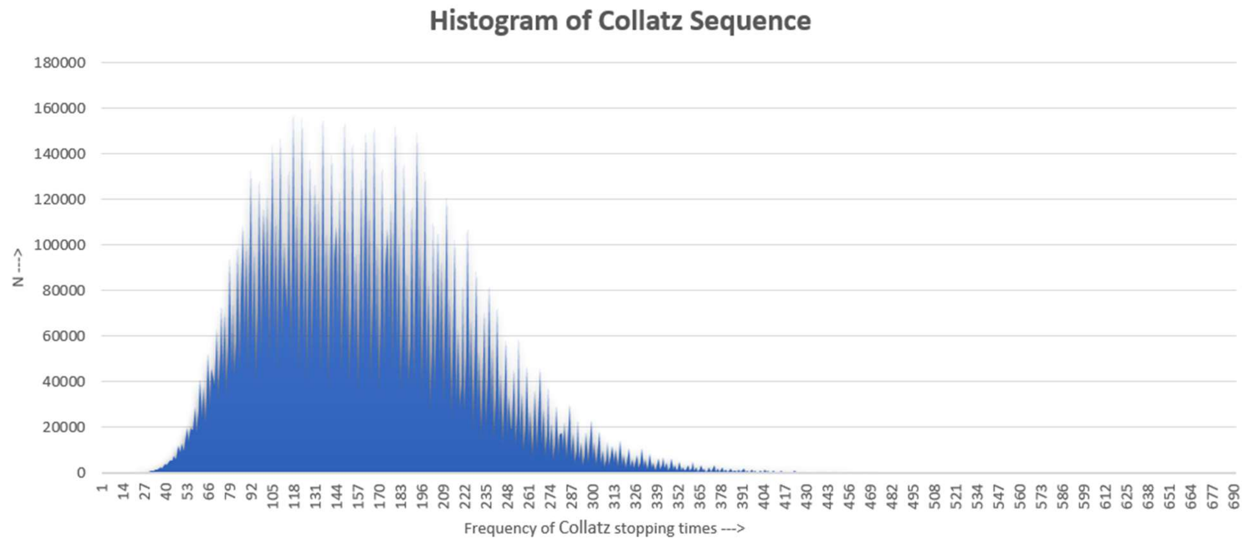
There is a mathematical conjecture that says that whenever you have a number and the number is even and you divide it by 2 and when the number is odd, you multiple it by 3 and add 1 to compute the sequence of numbers here you will end up eventually at 1. And it turns out that for any number sequence 1……….N, the result will always crunch down to 1. This mathematical conjecture is known as **Collatz Conjecture,** and it can be represented in mathematical formula as below**:**

$$f(n) = \begin{cases} \frac{n}{2}, & n \text{ is even} \\ 3n+1, & n \text{ is odd} \end{cases}$$

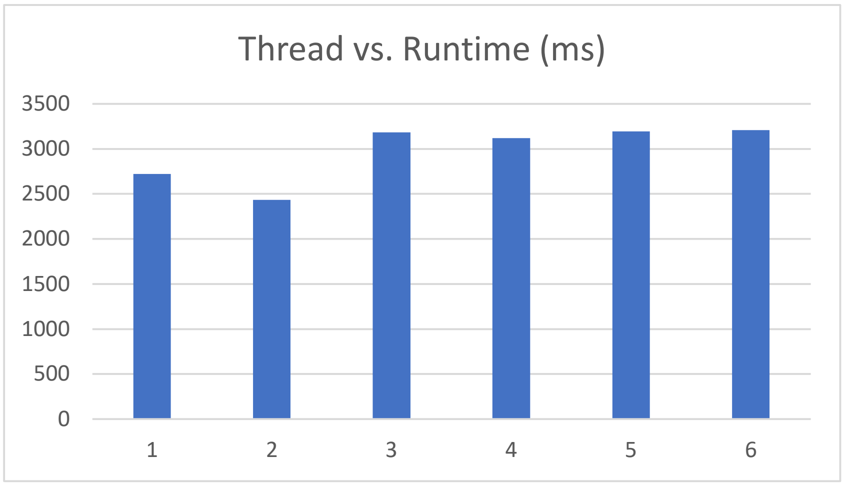
Figure 1: Collatz Conjecture Formula

Analysis

The histogram shows how often a particular stopping time occurs for the range of provided numbers N. Below histogram statistic is for the numbers(N) from 1 to 15,000,000 with 2 threads under race condition, showing the total stopping time of the Collatz sequence on the x-axis against the number of occurrences on the y-axis.



Graph 1: Histogram of range 15,000,000



Graph 2: Compiled and run on UWF CS server

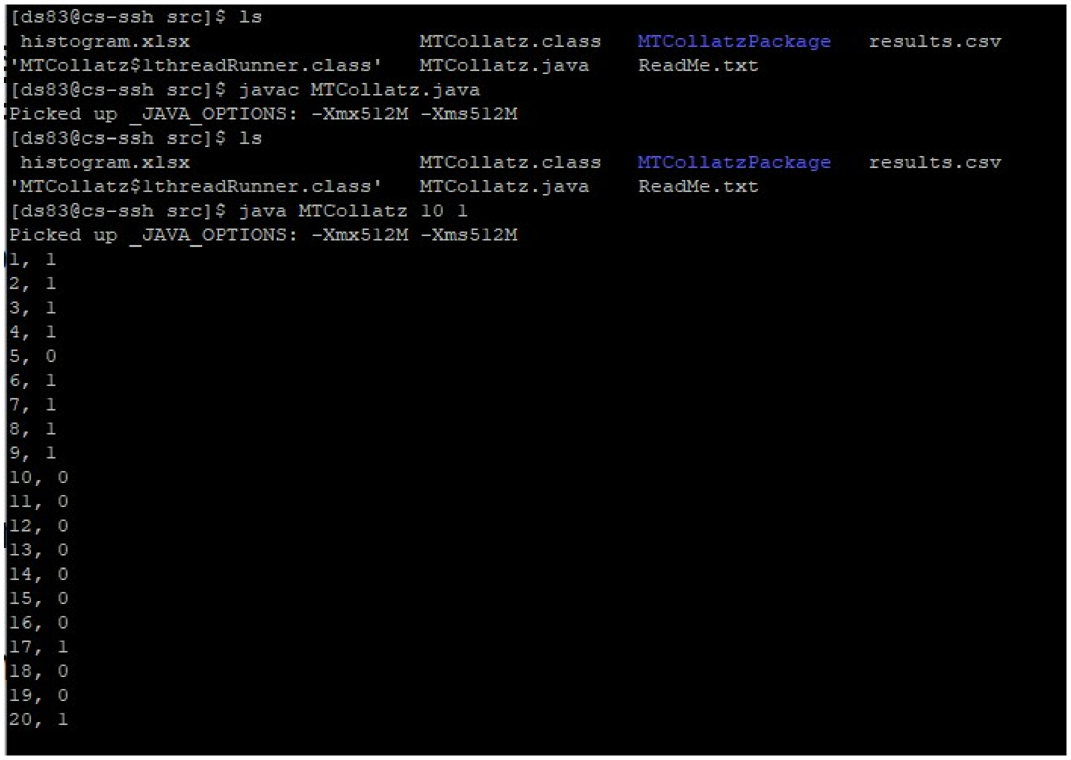
Unfortunately, we only see a consistent speed gain going from 1 thread to 2 threads. Beyond that, the time with more threads actually takes longer than the single thread. The theory is that because the calculations are fairly simple and run very quickly, the threads do not have to switch to run more efficiently, or to state otherwise, the single thread process is already close to perfectly efficient for the work, only beat by two threads. We tried printing out the thread names using command *Thread.CurrentThread().GetName()* and that showed that with multithreaded runs, the threads didn't switch often, rather, the CPU chooses to run one thread for a while, then switch to another thread, then another, each running for several iterations before switching. Therefore, we are lead to believe the code is working correctly, and that in a scenario where the calculations require more time outside the critical code section, then we would see more meaningful and consistent time decrease with thread increase.

Code Review

Some notable coding decisions were that the array length that stores frequency of stopping time is of size 1000 as instructed. The largest stopping time for values ranging to 5 million is less than 600, so quite reasonable array size. In conjunction, the values are accepted as type int, then converted to long, before the Collatz formula is applied to allow for the large potential growth before settling to value 1.

Another aspect of the code is that we separate the calculation and heavy lifting to a class called DataSetHelper, with each thread having one instance, such that the shared memory class DataSet is only locked for incrementing either the frequency of stopping time or the counter / index, and unlocked right after. Locking is done in a try encapsulation, and the unlock is contained in the finally section in order to ensure unlocking and avoid issues with deadlock.

The timing of the completion is measure from opening of first thread to closing of first thread, where the first thread will close after N passes the upper limit set by the user.



```
[ds83@cs-ssh src]$ ls
 histogram.xlsx                  MTCollatz.class   MTCollatzPackage   results.csv
'MTCollatz$1threadRunner.class'  MTCollatz.java    ReadMe.txt
[ds83@cs-ssh src]$ javac MTCollatz.java
Picked up _JAVA_OPTIONS: -Xmx512M -Xms512M
[ds83@cs-ssh src]$ ls
 histogram.xlsx                  MTCollatz.class   MTCollatzPackage   results.csv
'MTCollatz$1threadRunner.class'  MTCollatz.java    ReadMe.txt
[ds83@cs-ssh src]$ java MTCollatz 10 1
Picked up _JAVA_OPTIONS: -Xmx512M -Xms512M
1, 1
2, 1
3, 1
4, 1
5, 0
6, 1
7, 1
8, 1
9, 1
10, 0
11, 0
12, 0
13, 0
14, 0
15, 0
16, 0
17, 1
18, 0
19, 0
20, 1
```

Graph 3: Compiled and run on UWF CS server

Challenges

One of the biggest challenges we faced were a race condition issue, where we perform a while loop in the run() method implementing Runnable, outside of the lock.lock() and lock.unlock() safe space. Because of this, the counter was incrementing in the short time between the while conditional check and the lock function running, so we had to place a second conditional check after the lock so that there was no way the counter would increment from another thread which did have it locked.

Conclusion

The conclusion from above experiment turned out to be that there is a direct solution of Collatz´s conjecture for even numbers and finite number of computations for odd numbers. Computing the Collatz conjecture formula for all numbers N through single thread will always take comparable more time than multi-threaded approach.

The “Counter”, “Thread” and “Computation” should be mutually exclusive while performing the computation using multi-threaded approach. The probability of encountering the race condition in multi-threaded approach is high and thread computations should use locks.