Alexander N. Chin

Dr. Jey Veerasamy

CS 3377.005

16 November 2022

Multi-threaded Hashing Report

Program Background

The targeted program was a multi-threaded hashing program that generates unique hash codes for large files of Unicode characters. The hashing function was built from the Jenkins-one-at-a-time algorithm, which creates a unique hash code given an array of bytes. The multi-threading was handled by a binary tree organization of threads that are assigned specific parts of the file to compute hash codes and concatenate the hash codes of their children. The size of the tree is dictated by the user, thus, allowing for the performance analysis below. The full description of this behavior can be found in the project assignment file.

Prediction

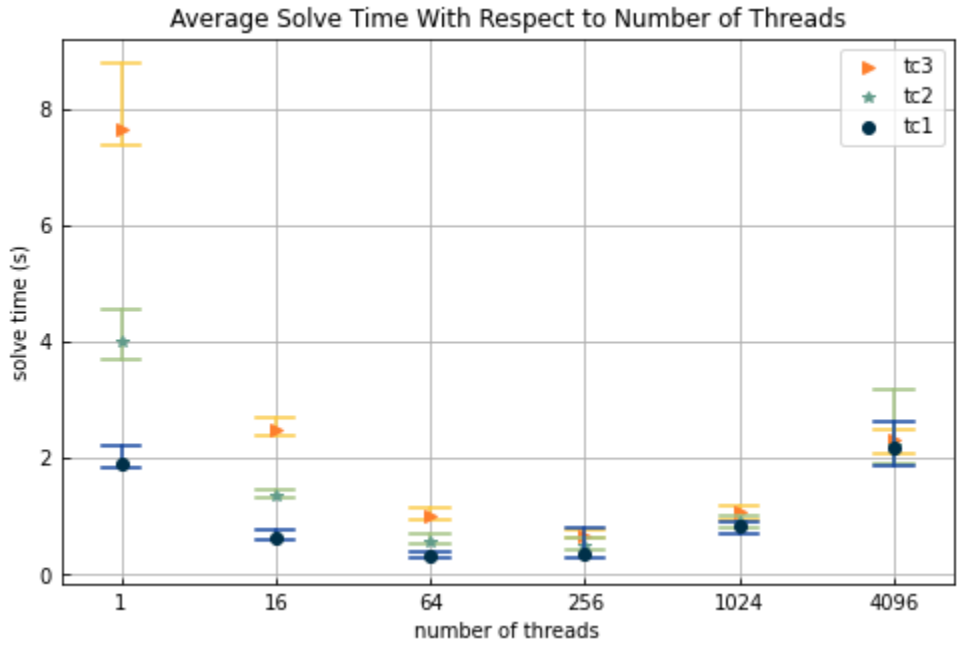
We carried out this experiment in order to observe the effect that an increasing number of threads processing in parallel would have on the real solve time of the program. The prediction was that *an increase in threads would correspond to a linear decrease in solve time*.

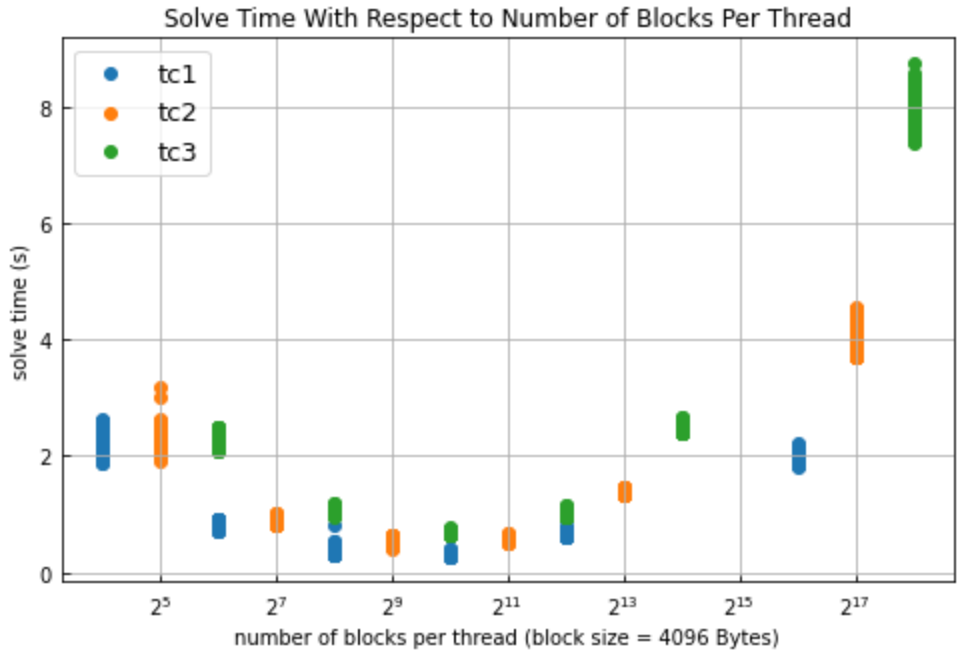
Execution

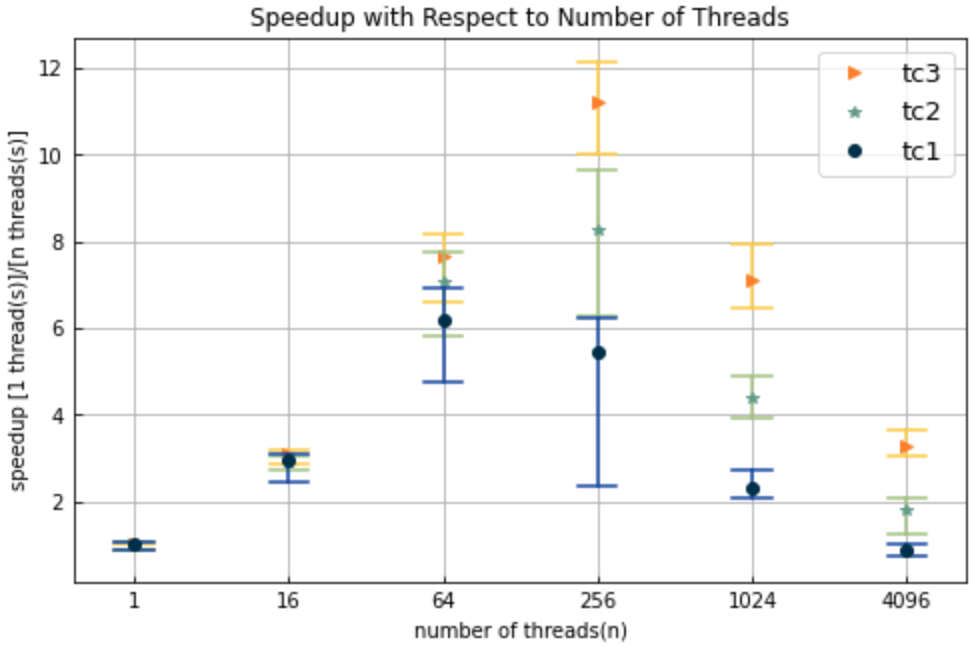
The following analysis has been carried out by executing the hashing program over 4 test files (tc1, tc2, tc3) with varying amounts of threads (1-4096) on one of the University of Texas at Dallas's servers (CS3), which boasts 46 cores to enable parallelism in the execution of threads.

Each thread amount was tested 1000 times to account for variance in solve times. Although the program has been programmed to run n threads in parallel, it ultimately comes down to the server scheduler to execute the threads in parallel. This results in a large source of variance within the dataset.

Graphs







Analysis

The data shows a quadratic correlation between the solve time and the number of threads. The same can be said about the correlation between the solve time and the number of blocks per thread. This disproves our original prediction regarding the linear speed up, but it gives insight

into the practical application of multi-threading: although initially, an increase in threads results in a decrease in solve time, as the number of threads passes a certain threshold (around 256 threads in this experiment marked by the maximum of the curve) the corresponding speed up is bottlenecked by the overhead of creating, calculating, and passing the values between each thread. Graph 3 visualizes this drop in speedup past the aforementioned threshold by displaying the ratio between the solve time with a single thread and the solve time with n threads working in parallel.

Conclusion

Although multi-threading can be implemented to speed up runtimes by large factors (more than 12 times as observed in this experiment), at a certain threshold of threads, the overhead of creating these threads outweighs the computation time of the thread, which slows down the program. Thus, the number of threads used has rapidly diminishing returns on solve time and speedup.