Multi-threaded Hash Tree

Experimental Study on Time vs Threads on Various Hash Sizes

Created by Zaid Hilweh (zsh190000) and Joshua Cox (jac200012)

CS/SE 3377 Sys. Prog. in Unix and Other Env.

Background

This program is a multithreaded hash tree, where its objective is to compute the hash of a given file. Sizes of the files can be very large, varying from 256MB to 4GB. Using the binary tree style, a node is created for every thread and given a certain block from the file to work with in order to hash. The children feed their hash to the parent, concatenate their hashes, and rehash repeating until the root node is left with the final hash for the file. Using threads to compute the hash in parts rather than the file being hashed in one instance allows for significant speed up on the time to hash. This study will be exploring the effect of various amounts of threads and its effects on the hash speed of a file.

Design

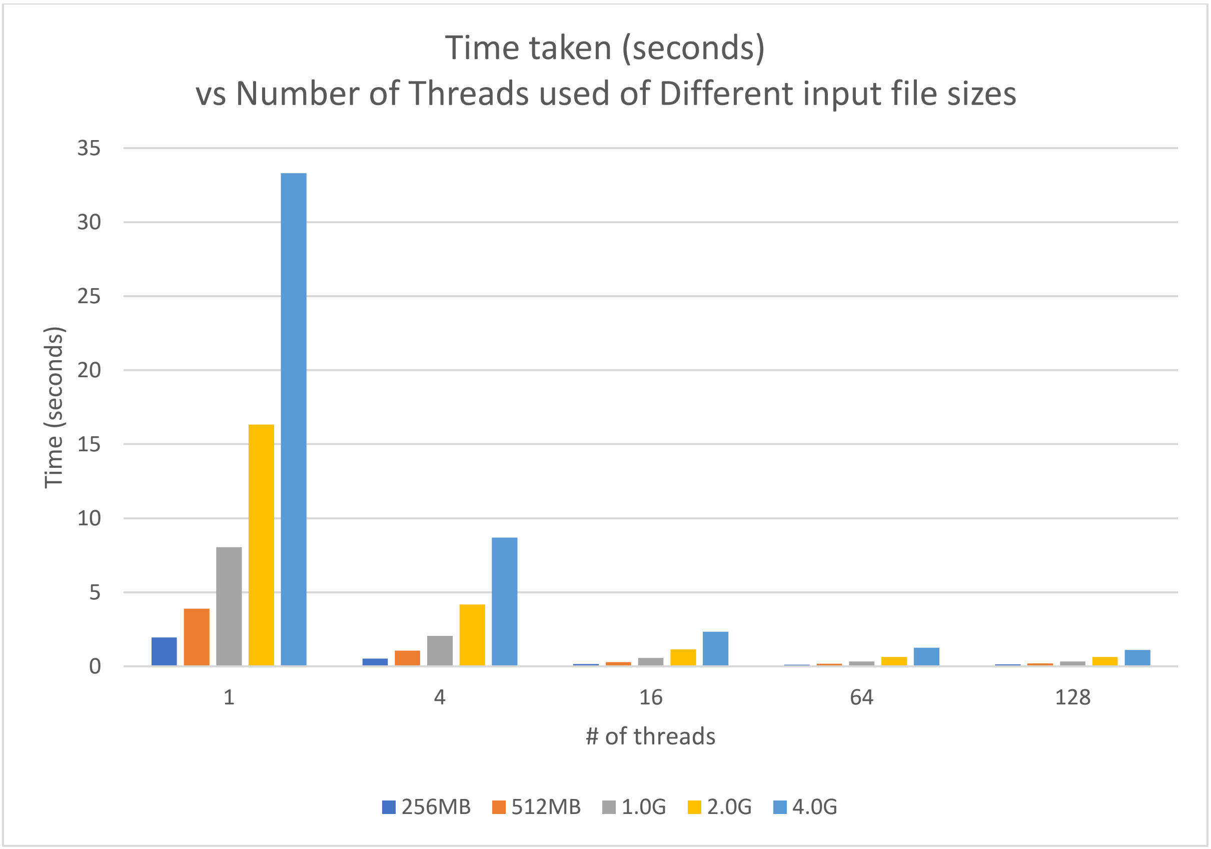
This program was created with two major concepts in mind. Hashing and thread creation. Hashing is a concept in which a set of strings are changed into a fixed length numeric value. In the design of this project the function calcHash is used to calculate the hash of the strings and is the Jenkins hash function. Information on the Jenkins hash function can be found at https://en.wikipedia.org/wiki/Jenkins_hash_function. Threads are created in a binary tree style to hash certain parts of the file, to make the process faster. The binary tree style is accomplished where each thread creates two child threads, and the process is repeated until the program reaches the given number of threads. Threads are then assigned certain consecutive blocks of data (blocks are 4096 bytes) and the thread hashes its own blocks then proceeds to concatenate its hash with its parent, and rehashes. This process repeats until there is one hash left at the root node and is returned to the user.

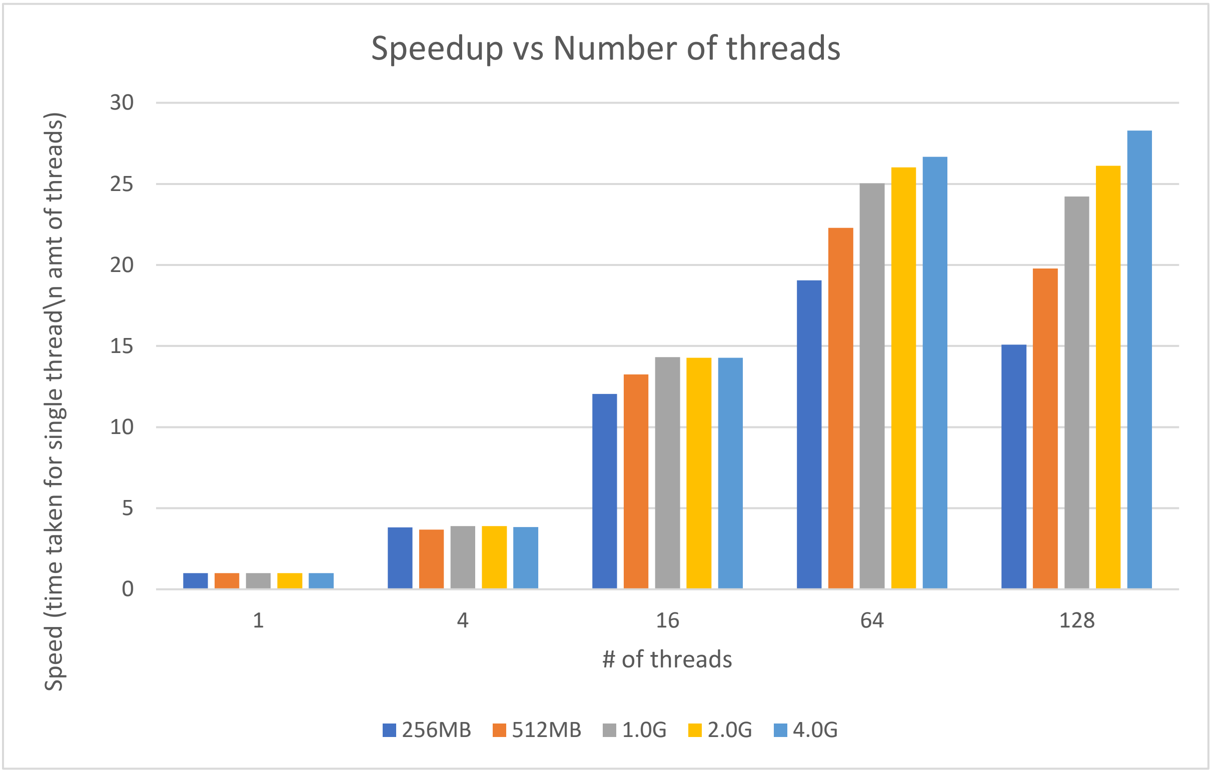
Observations

Hash time is recorded across the different thread counts, to see how long each variation of thread counts, 1, 4, 16, 64, and 128, would differ in execution time to give a final hash of the file. As well as how the speed up, time taken for a single thread divided by time taken for n threads, compares across thread counts.

Data was taken on cs3.utdallas.edu server. Contains 48 CPU’s (2 cores 12 threads) running at 3.1GHz. Data was internally measured within the program, while running on the server, using the clock_gettime() function given by the time.h library. Given different input files, time was recorded to hash, and done ten times, to give an average run time.

Observation of averages is listed below in two different graphs, the first one containing the run time for hashing of different file sizes and thread counts. While the second one is the ratio of time for one thread count run time, to 4, 16, 64, and 128 threads, across multiple files, and is referred to as speedup.


Time taken (seconds)
vs Number of Threads used of Different input file sizes
35
30
25
20
15
10
5
0
Time (seconds)
1
4
16
64
128
# of threads
256MB
512MB
1.0G
2.0G
4.0G


Speedup vs Number of threads
30
25
20
15
10
5
0
Speed (time taken for single thread\n amt of threads)
1
4
16
64
128
# of threads
256MB
512MB
1.0G
2.0G
4.0G

Conclusion

The reason for implementing the multi-threaded hash tree is to decrease the amount of work each instance must do, and therefore decreasing the time to hash. It is assumed that as one has more threads, doing less work each that the decrease in time to hash across multiple file sizes will be sped up proportionally. However, this isn't the case. Initially with the 256mb file, there is a proportional speed up as thread count increases from 1, 4, 16, and finally to 64 threads. However, when we run the program using 128 threads, its performance is less than that of 64 threads, and is even comparable to using 16 threads to hash the file. As the file sizes increases to 512mb, 1gb, and 2gb, 128 threads do outperform the use of 16 threads, however its performance is still lackluster compared to the time to hash of 64 threads. 128 threads do outperform 64 threads on the 4gb file, but only by a difference of 6 percent. This shows that as the thread count increases exponentially the difference in time to hash won't be proportional, and sometimes even worse, depending on the file size.

Why do 128 threads only outperform 64 threads when the file size reaches 4gb? It's shown that the performance of 128 threads is hindered in smaller file sizes, and only sees better performance when it reaches a file size of 4gb. It can be inferred that performance is hurt with smaller file sizes, because there is more time to wait on the threads, due to there being a larger quantity, than it does to compute the hash initially, giving a longer turnaround time to return to parents and rehash, compared to hash times of smaller threads.

In conclusion, more threads don't always imply that the work will be done at a faster rate, as seen from this experiment. Both times to hash and speed up aren't proportional to the thread count and can even be hurt due to an excessive number of threads as seen with the use of 128 threads. The increase in thread counts leads to a decrease in the growth of speedup, implying that for any given file as the thread count increases, there will be a point in which speedup will not increase, and could even decrease.