Effect of Multiple Threads on Hashing a File

Kristopher Pally

Intro:

When hashing a file, a program must go through each byte in a file and pass the information to a hashing function. For small files, this takes under a few seconds on modern processors. For sufficiently large files, however, this can take minutes to complete on a single thread. This can be reduced by using a divide-and-conquer paradigm to split the file into blocks and assign those to multiple threads for hashing. Multithreading the hash for a file will let each thread hash simultaneously. The structure we are using to handle the hash is a Binary Tree where each node is assigned a block to hash and child nodes pass their hash to the parents who then combine their own hash with their children's to create a final hash to send to their parent.

Hypothesis:

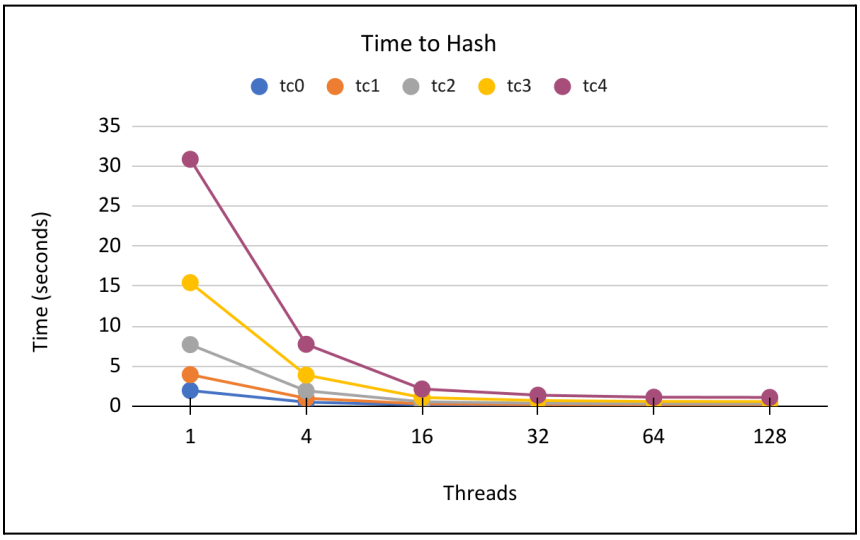
In this project, we divide a file into blocks of 4096 bytes and assign those blocks to threads for hashing. The expected result is that we will see a speedup until the data each thread has to hash is small enough that there will be negligible gains in performance. In other words, we will see diminishing returns as we add more threads.

Experiment:

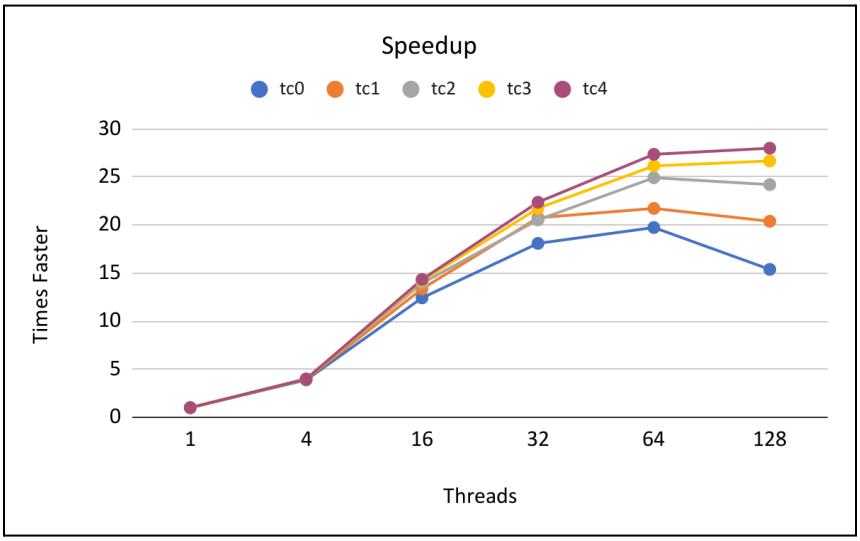
The test files used in this experiment were of sizes 256 MB, 512 MB, 1 GB, 2 GB, and 4 GB for tc0-4 respectively. The number of threads used for each file were 1, 4, 16, 32, 64, and 128. I ran each thread count 5 times and used the average of those runs to construct my graphs. The speedup per thread used the time taken for 1 thread to hash the file. The system I ran my experiment on was a server with 2 CPUs, each with 24 threads making a total of 48 threads. The hashing function used was Jenkins' one-at-a-time hash.

Results:

The time needed to hash a file goes down as the number of threads used goes up. Looking at the graph below, we can see that at 32 threads the time to hash each file is within a couple of seconds of each other despite the large size difference between tc0 and tc4. 16 threads appear to be the limit at which the time to hash each file reaches a point where adding more threads does not affect the time in a substantial way.



The speedup graph shows how adding more threads affects performance directly. This graph was made using the time needed for 1 thread to hash as a base and comparing it to the times for the remaining threads. There is an increase in performance as we add up to 64 threads for each file, but the smaller files (256 MB, 512 MB, and 1 GB) see a loss in performance at 128



threads. This could be because creating the threads and combining their hashes takes longer than an individual thread to hash. For the 2 GB and 4 GB files, however, there is a very slight improvement in performance. From this, we can conclude that larger files benefit more from creating more threads compared to smaller files.