Multi-threaded Hash Tree

Experimental Study on the Time Taken to Hash Files of Various Sizes with a Varying Number of Threads

Matthew Pham (mnp190003)
CS/SE 3377 Sys. Prog. in Unix and Other Env.

1. Background

In this project a multi-threaded program is used to compute the hash value of a given file. Files of various sizes are tested, including large files up to 4GBs in size. By implementing the threads in a binary tree-like structure, each thread can compute the hash value of multiple blocks(parts of the file). The hash values returned by each thread can be combined to compute new hash values until one final hash value is returned. In doing so, the time taken to compute the hash value of a given file can be significantly reduced. This study will explore the effects of using multiple threads on the time taken of hashing files of different sizes.

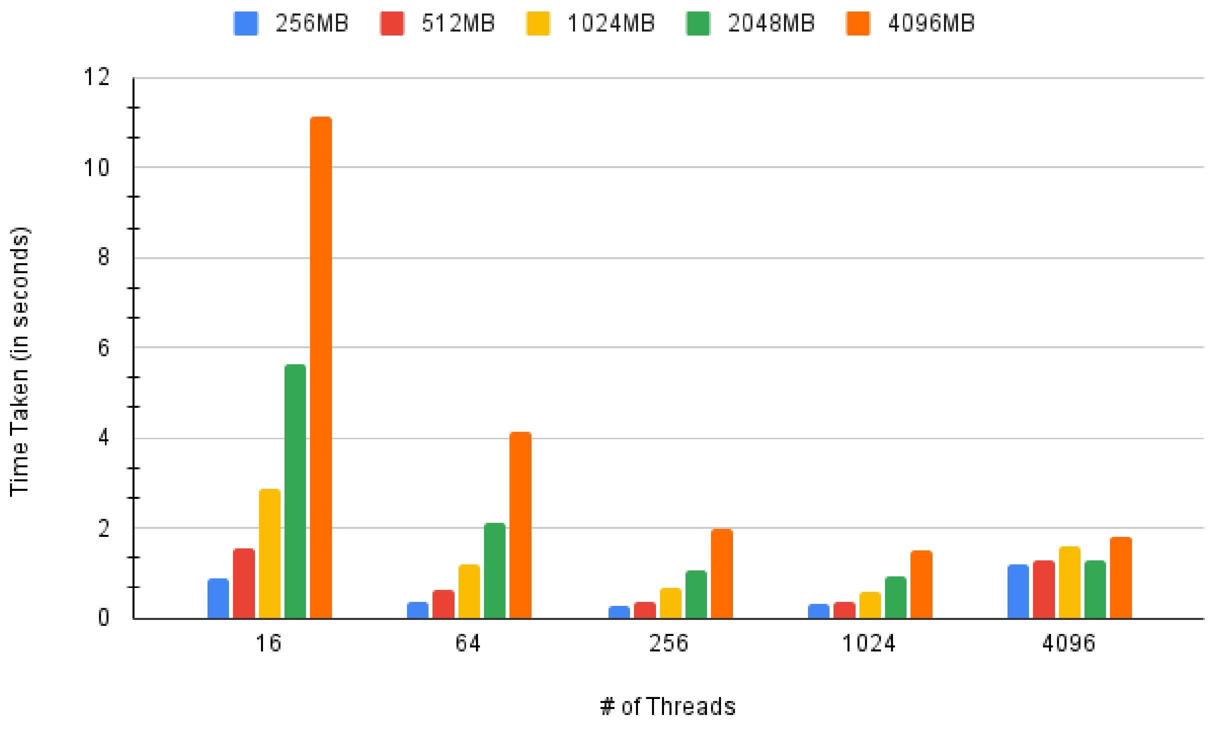
2. Design and Implementation of Multi-threaded Hash Tree

The design of the program relies on two major concepts, hashing and thread creation. Hashing refers to the transformation of strings into a fixed length numeric value. In this particular project, the Jenkins one_at_a_time hash algorithm will be used to produce hash values for the input strings of the files that are intended to be hash. Information on the algorithm is given in https://en.wikipedia.org/wiki/Jenkins_hash_function. In order to create a binary tree of threads, a root thread is created which creates two child threads, and recursively makes a tree of height n which is given in the arguments passed to the program. Threads are assigned a certain number of consecutive blocks which is calculated based on the number of threads used divided by the number of blocks in the given file (with each block being 4096 bytes in this given implementation). Leaf threads (threads without children) compute the hash value of their assigned blocks in the file. Internal threads (threads with children) are returned the computed hash values of their child threads which they concatenate with their own computed hash value. A hash value is then computed from that value and passed on until it reaches the root thread, in which the final hash value is returned.

3. Observation of Results and Efficiency

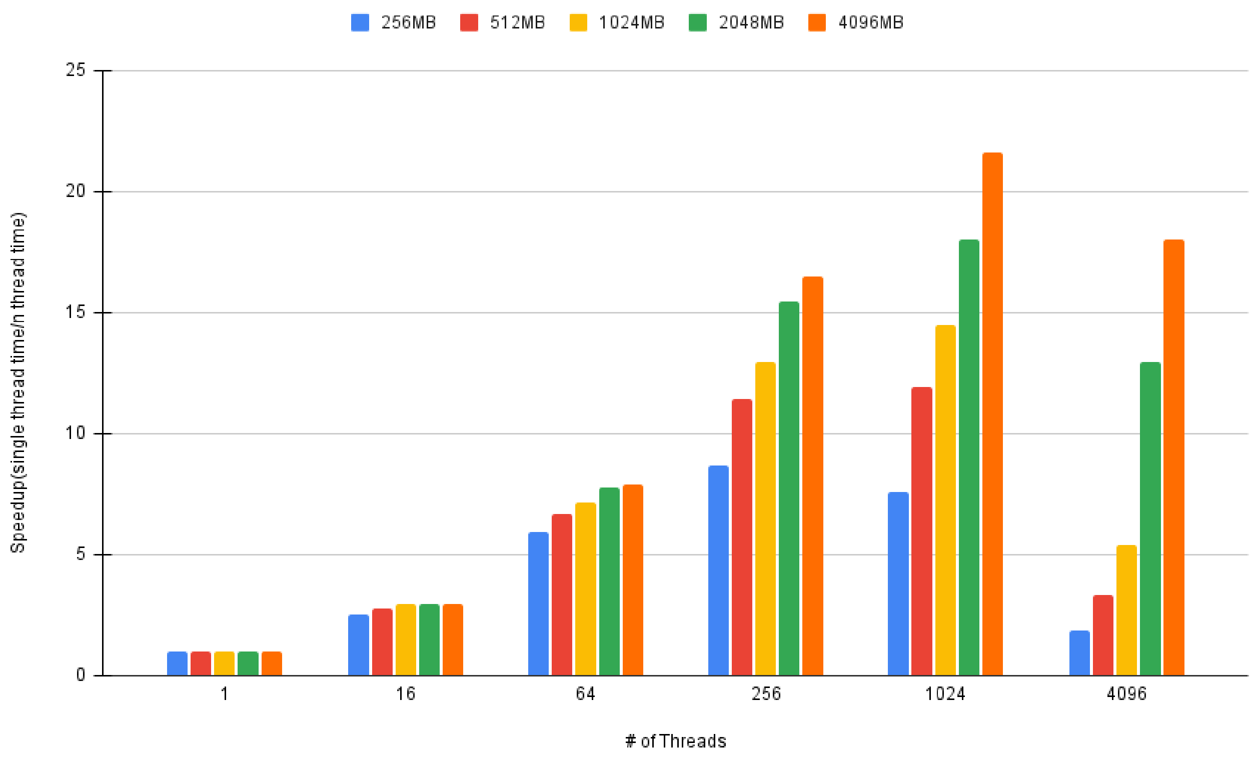
Using various files with file sizes of 256MB, 512MB, 1024MB, 2048MB, and 4096MB, the total time taken to compute the hash values of each file at thread counts of 1, 16, 64, 256, 1024, and 4096 is measured and documented in the bar graph below.

Graph 1: Time Taken (in seconds) vs Number of Threads used in Files of Various Sizes



In addition, the speed up which can be defined as the time taken for a single thread divided by the time taken for n threads, was calculated and plotted against the number of threads used in the bar graph below.

Graph 2: Speed Up vs Number of Threads used in Files of Various Sizes



4. Conclusion and Analysis

The purpose of implementing the multi-threaded hash tree was to decrease the time taken to compute the hash value of a given file. As observed in the experiment, the time taken to compute the hash value saw a significant drop off from 1 thread to 16 threads and with marginally smaller drop offs as the number of threads increased. Another important observation is that the drop off in time taken is more significant as the size of the file increases, which makes the use of threads more valuable in larger file sizes. Speed up in each case peaks around 1024 threads except for in the 256MB file which peaks at 256 threads. Surprisingly, in all cases increasing the number of threads from 1024 to 4096 causes an increase in time taken which is a reduction in speedup. And in the file of size 256MB (the smallest file) the time taken is actually greater than the time taken with 1 thread. It can be inferred that this occurrence may be due to the fact that internal threads must wait for child threads to compute their hash values before they can compute their own hash value from concatenation. It might be possible that using an excessive amount of threads can cause slower performance because of this induced wait time, which may have outweighed the decrease in time taken from using multiple threads initially.

In conclusion, the number of threads used is not necessarily proportional to speed up nor a reduction in time taken. The decrease in speedup from 1024 to 4096 threads and the max speed up being achieved around 1024 threads shows that the optimal amount of threads to use in this particular experiment is around 1024 threads. In this experiment speed up was shown to be from around 7 times faster to as high as 21 times faster with 1024 threads than with 1 thread. The increase in speed up at 1024 threads is higher in files with larger sizes. Per the results of the experiment, hashing files, particularly large files, with around 1024 threads can lead to a large reduction in time taken to compute a hash value.