Multithreaded Hash Tree Project Report

Introduction

Multi-threaded trees effectively decrease execution time by dividing a task amongst a number of threads. We designed and wrote a program using multi-threaded hash trees to compute the hash value of a given file. The goal of this experimental study was to find the speedup achieved when the number of threads is increased and determine if a higher number of threads decreases the time taken to compute the hash value.

Setup

The program creates a multi-threaded hash tree that assigns each thread a number and divides each file into multiple blocks based on file size. Each thread is given a number of blocks and it computes the hash value of those assigned blocks. Each thread is also responsible for concatenating the hash values of its child threads with its own hash value. This new value is rehashed and sent to the parent and this process repeats until the root thread is reached. The final hash value is then computed and displayed.

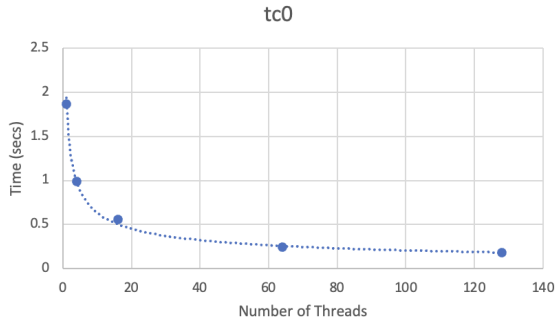
The finished program was run on the cs3 server with 1, 4, 16, 64, and 128 threads to observe the time taken to compute at each thread number. The cs3 server has 48 CPUs and 2 threads per core which is more than cs1 and cs2, which have 1 thread per core and have 12 and 16 CPUs, respectively. The higher number of CPUs and threads per core allows the program to run more efficiently, decreasing the time it takes to compute the hash value.

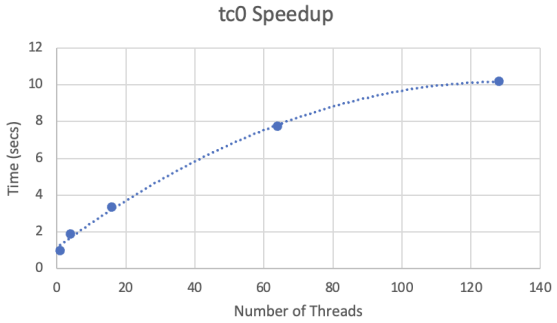
Results

It is expected that when the number of threads increases, the time taken to compute the hash value decreases, and our results support this. Looking at both graphs for each file, we see the time decrease and the speedup increase proportionally to the number of threads. The time decrease is exponential. Based on our observations, the time taken to compute hash values continuously decreases as the number of threads increases up to 128 threads. However, this may not happen at a higher number of threads if the machine used to run the program has fewer CPUs than 48. The CPUs and the threads allow for parallelization to occur making the program time efficient. The speedup occurs until 128 threads but afterwards it becomes more constant. The results for all five files follow the same trend. If the file was bigger, it took longer for the hash value to be computed compared to a smaller file but the trend of time decrease and speedup is something that can be seen in all five files regardless of size.

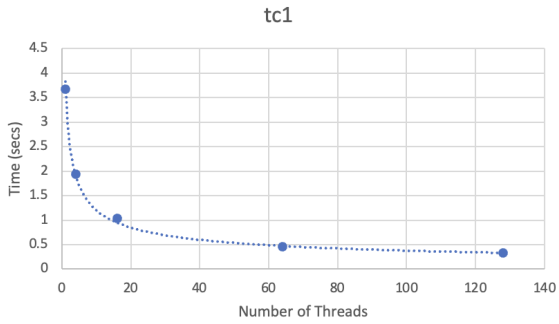
Experimental Data

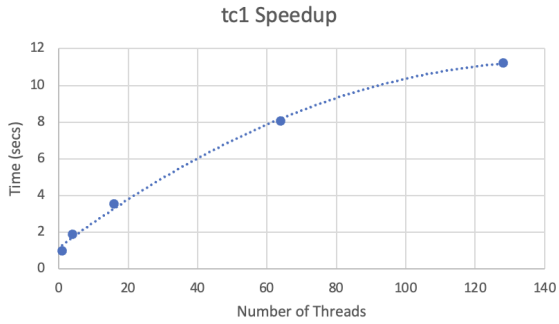
Graphs for p2_tc0:



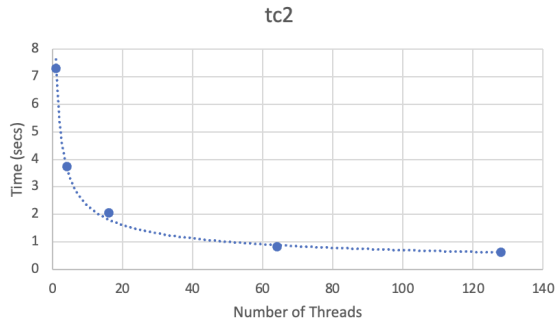


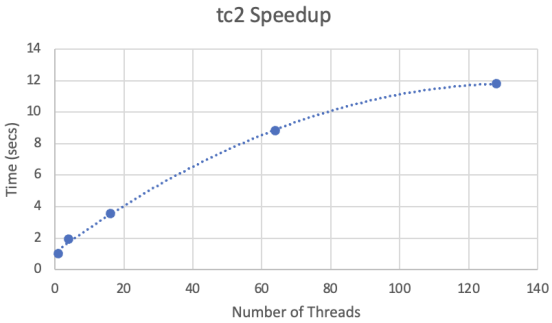
Graphs for p2_tc1:



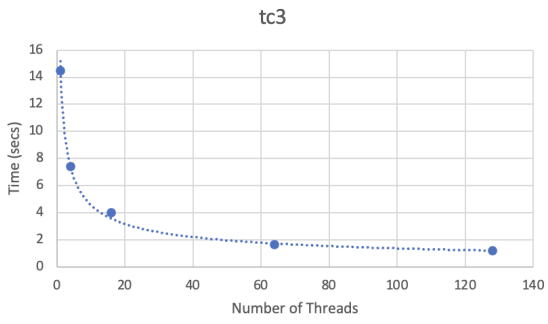


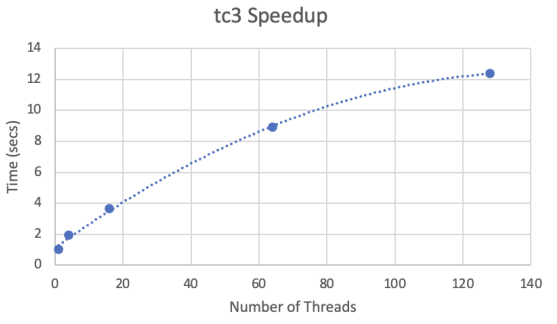
Graphs for p2_tc2:



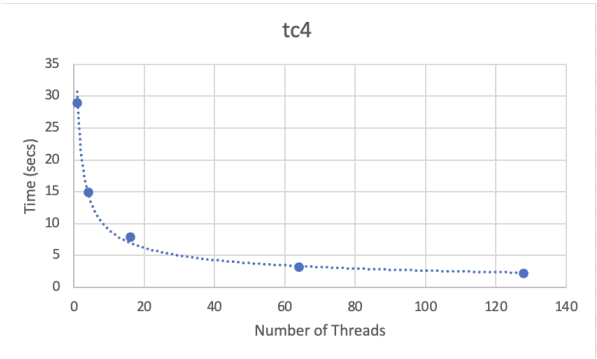


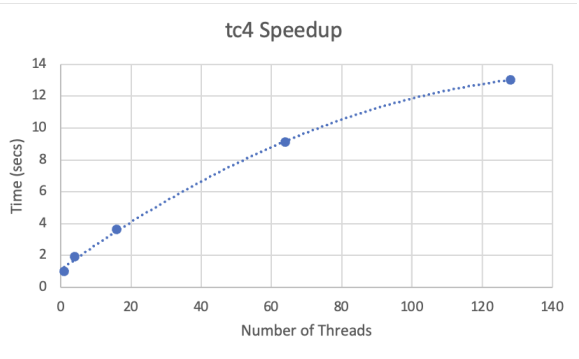
Graphs for p2_tc3:





Graphs for p2_tc4:





The table below holds the experimental data collected for each file with a different number of threads. This data is also depicted in the graphs.

Threads	p2_tc0	p2_tc1	p2_tc2	p2_tc3	p2_tc4
1	1.865794	3.669006	7.295486	14.497977	28.977028
4	0.990227	1.944111	3.730383	7.411216	14.868894
16	0.553812	1.033045	2.050331	3.982831	7.91848
64	0.240856	0.454397	0.824918	1.625606	3.18245
128	0.183071	0.327327	0.617794	1.170324	2.223955
256	0.208728	0.318067	0.553344	1.114919	2.036651

The table below holds the speedup calculated using the experimental data collected for each file with a different number of threads. This data is also depicted in the graphs.

Threads	p2_tc0	p2_tc1	p2_tc2	p2_tc3	p2_tc4
1	1	1	1	1	1
4	1.88420837	1.88724101	1.95569356	1.95622108	1.94883547
16	3.36900248	3.55164199	3.55819914	3.64011855	3.65941797
64	7.74651244	8.07445032	8.84389236	8.91850608	9.1052579
128	10.1916415	11.2089928	11.8089298	12.3880028	13.0295028
256	8.93887739	11.5353243	13.1843591	13.0036146	14.2277828