Cameron Wallace
6/7/2020
CSCI 347 Spring 20

Assignment 6 Report

In order to implement pthreads to speed up matrix multiplications I started by creating a struct that would contain the necessary variables that would be passed to the thread body. The struct contains pointers to the matrices being used, the dimensions of these matrices, the indices of the resultant matrix that the given thread will calculate, along with several other variables that store the thread number and barrier. Once passed to the thread body each of these threads containing specific values in their struct would calculate the final matrix cell by cell by first calculating which row and column the final cell belonged to and using that row and column from the original matrices to calculate.

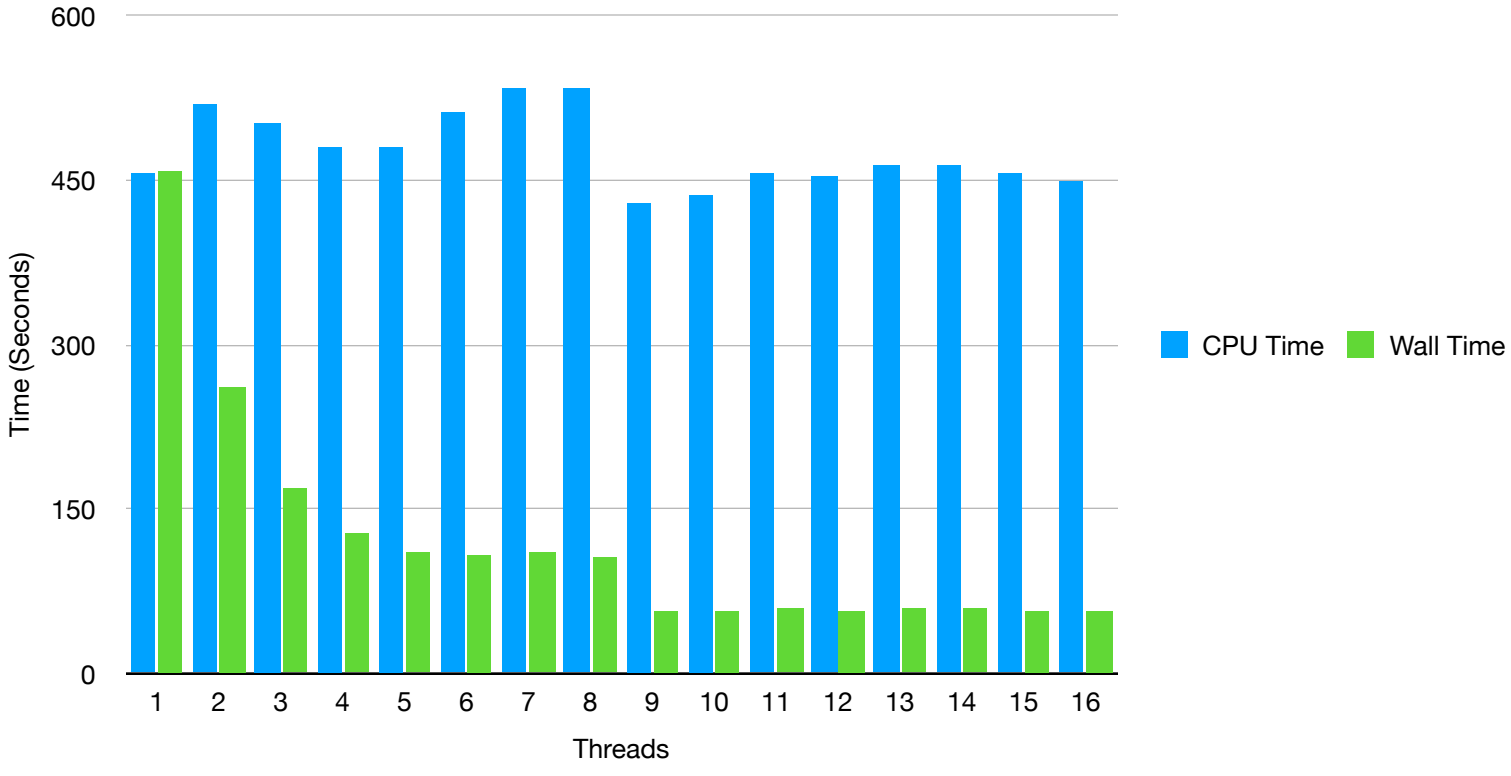
For my experiments I was interested in comparing speeds for a small, medium and large matrix multiplication. The dimensions used for MatSquare were 50 by 50, 500 by 500, and 2000 by 2000, with each matrix being squared 6 times. For MatMultiply the chosen dimensions were 25 x 50 and 50 x 25, 250 by 500 and 500 by 250, and finally 1000 by 2000 and 2000 by 1000.

Dimensions	Square operations	CPU Time	Wall Time	Threads
50 x 50	6	0.006698	0.006672	1
500 x 500	6	2.073123	2.073161	1
1000 x 1000	6	21.366532	21.367059	1
2000 x 2000	6	254.510489	254.523707	1
50 x 50	6	0.007319	0.002010	8
500 x 500	6	4.142180	0.534803	8
1000 x 1000	6	45.846051	5.744855	8
2000 x 2000	6	462.953724	57.967534	8

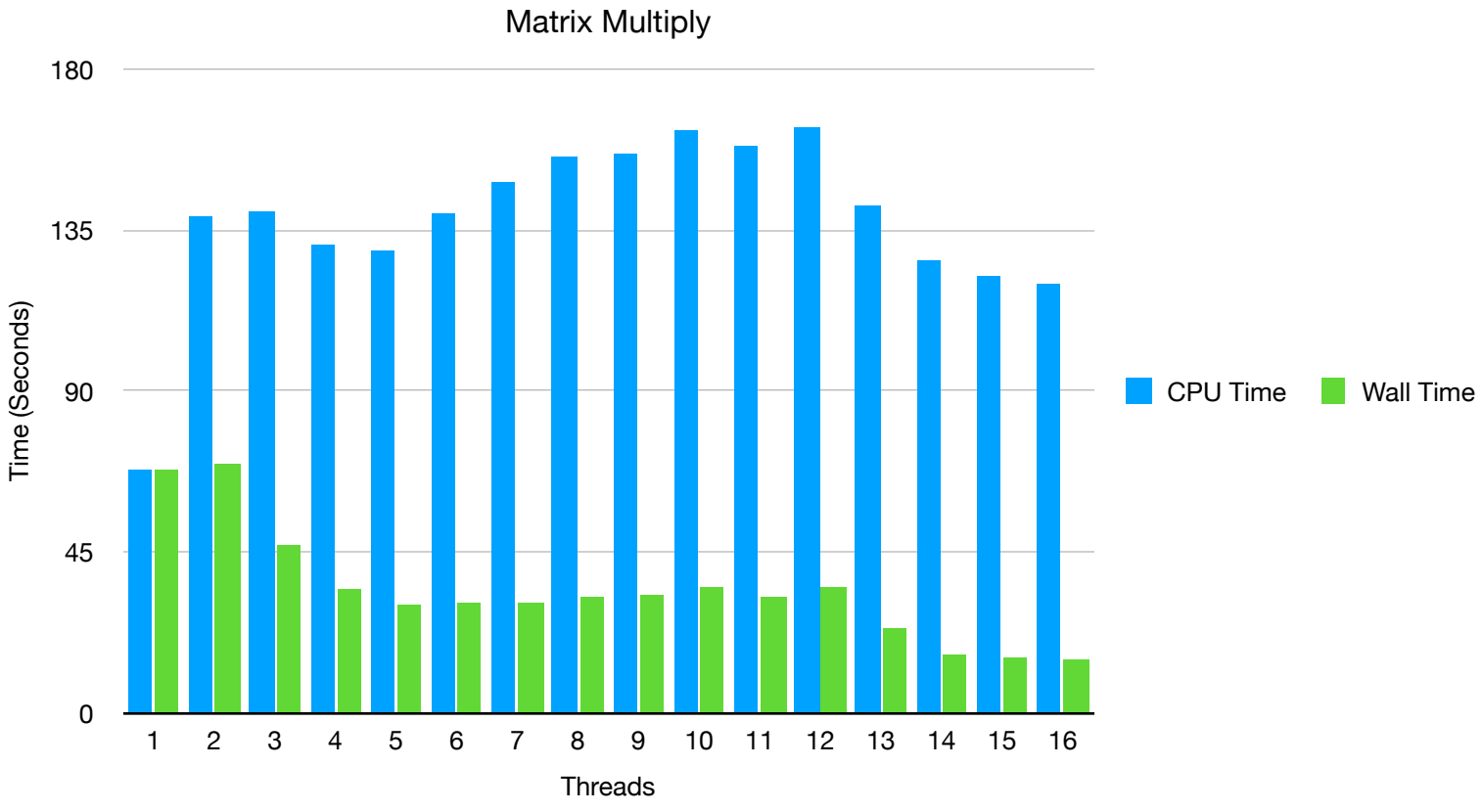
Dimensions	CPU Time	Wall Time	Threads
25 x 50 x 25	0.000945	0.000915	1
250 x 500 x 250	0.089994	0.089986	1
2000 x 3000 x 2000	67.329129	67.330701	1
25 x 50, 50 x 25	0.000919	0.000339	8
250 x 500 x 250	0.174816	0.022808	8
2000 x 3000 x 2000	118.925234	14.914037	8

I was also interested in seeing how a different number of threads changed the times for the largest matrices in both situations as this seems to more accurately and effectively represent the speed up times for increased threads.

Square Matrix: 2000 x 2000, 6 Square Operations		
Threads	CPU Time	Wall Time
1	456.650003	457.480981
2	520.260645	262.399993
3	501.449951	170.314076
4	480.074446	128.592922
5	481.500856	110.504349
6	512.037149	109.354642
7	533.389674	112.115787
8	532.92345	107.143605
9	429.499923	57.0449
10	437.808489	57.825691
11	455.330006	59.100208
12	453.273552	58.681016
13	462.898122	59.345899
14	463.092082	59.166252
15	456.562454	57.800487
16	449.510289	56.788859



Two Matrices: 2000 x 3000, 3000 x 2000		
Threads	CPU Time	Wall Time
1	68.33982	68.36255
2	138.7213	69.68295
3	140.1203	47.0557
4	131.0579	34.72495
5	129.4808	30.20143
6	139.6464	30.69364
7	148.0897	31.21643
8	155.8286	32.65512
9	156.2663	33.22205
10	163.2428	35.02637
11	158.416	32.1996
12	163.5061	35.23697
13	142.0733	23.5662
14	126.3639	16.11825
15	122.0041	15.59316
16	119.8164	15.13889



My results for squaring the matrices were closer to my expectations than the results from multiplying two matrices together. While the wall time decreased as more threads were used the CPU time increased from 5 to 12 threads. For squaring matrices I was not expecting the performance to plateau once more than 9 threads were being used to perform the calculations.

In conclusion it can be seen that increasing the number of threads used to perform calculations can provide a significant increase in speed, depending on the calculations being performed and the way in which the load is split. My implementation of threads improved squaring a matrix 6 times by ~805%, and it improved multiplying one matrix by another by ~452%.