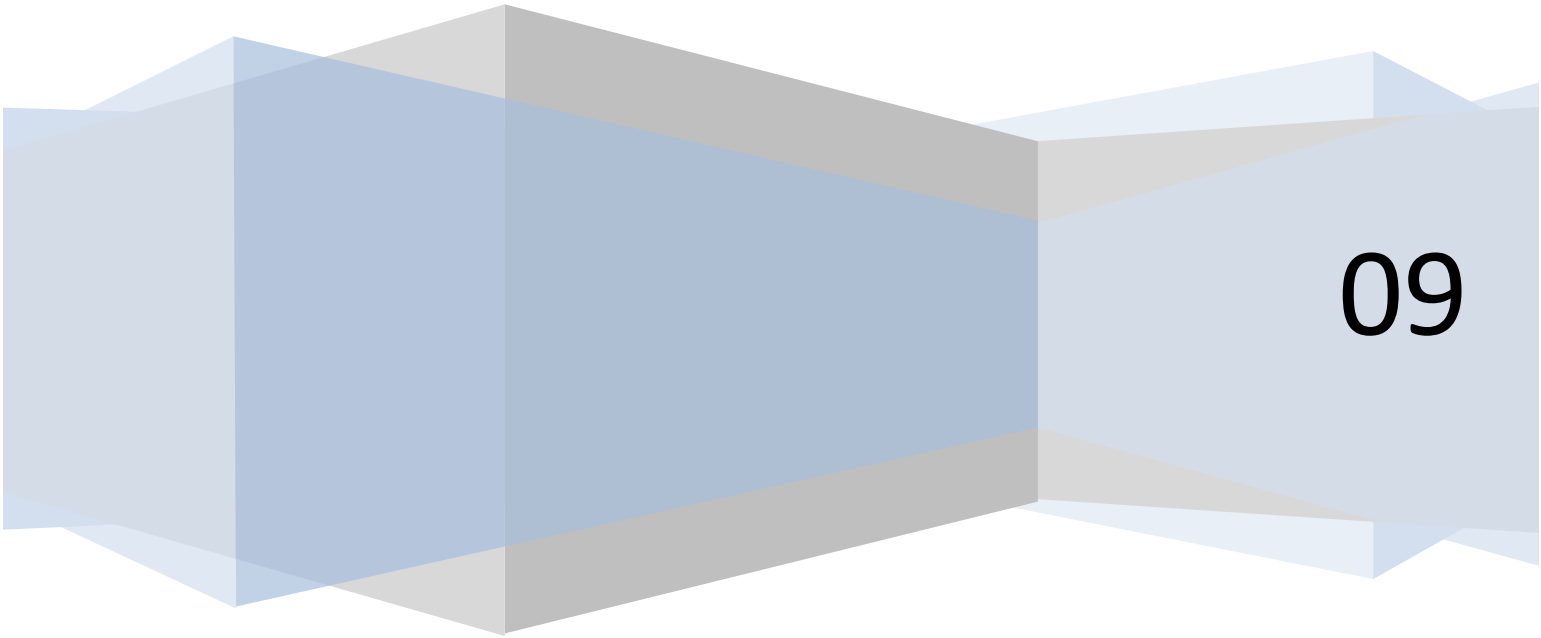
Threads Assignment

CS 732

Joshua Hegie



09

The goal of this assignment was to get a grasp of how to use threads by implementing several simple projects.

Test Bed:

All of these experiments were run on the 16 core machines on rack 4 in the

Thread Creation:

Experiment	Thread Creation
Average (us)	40.54

Table 1: Average thread creation time

The goal of this part of the project is to determine how long it takes for the test bed to create a thread. My implementation created 10,000,000 posix threads (pthreads) and determined the amount of time taken. This program was run 100 times. The average of all of these runs is shown in table 1.

Mutex Locking:

Experiment	Mutex Lock/Unlock
Average (us)	0.08

Table 2: Average mutex lock/unlock time

This part of the project is designed to test how long it takes to lock and unlock a mutually exclusive piece of code using a mutex (mutual exclusion). My implementation locked and then unlocked a mutex 100,000 times. On average, and in fact during every run, it took the system 0.08 µs, when averaged over 100 runs of the program.

Ping Pong Test:

Experiment	Ping Pong
Average Time (us)	97.997

Table 3: Average Ping Pong test time

This section is the time required to make 2 threads communicate. Due to an inability yo understand how message passing between threads works at this time, the communication is achieved vi a global variable that both threads watch for changes. On average this took 97.997 µs, as shown in table 3, which includes time to lock the variable to ensure that only one thread was ever editing it at any given time. The message was passed 10,000 times, and the code was run 100 times

Count 3's Test:

#of Threads	Time (us)
0	493.457
1	1501.218

2	785.187
3	559.911
4	451.232
5	388.521
6	350.594
7	334.945
8	326.319
9	589.522
10	600.667
11	494.948
12	484.482
13	486.941
14	501.638
15	488.517
16	478.926

Table 4: Average time for count 3's with the given number of threads, 0 threads is sequential

#of Threads	Time (us)
1	0.32870443
2	0.62845793
3	0.88131328
4	1.09357714
5	1.27009093
6	1.40748843
7	1.47324785
8	1.51219206
9	0.83704595
10	0.82151508
11	0.99698756
12	1.01852494
13	1.0133815
14	0.98369143
15	1.01011224
16	1.0303408

Table 5: Speedup for Count 3's

#of Threads	Time (us)
1	0.32870443
2	0.31422897
3	0.29377109
4	0.27339428
5	0.25401819
6	0.23458141
7	0.21046398
8	0.18902401
9	0.09300511
10	0.08215151
11	0.09063523
12	0.08487708
13	0.07795242
14	0.07026367
15	0.06734082
16	0.0643963

Table 6: Efficiency for Count 3's

This code was provided in the book, as a means of showing how the cache sizes on a CPU can affect how parallel code is run. For this code an integer array of 1,000,000 integers 0 to 9, inclusive, is processed by the given number of threads. In the case of 0 threads the code is being run as a single process with no threads. This actually did better than the single thread case, which can be attributed to the extra overhead in creating and tearing down the thread. There is an increase when a second thread is added, cutting the time down, which is expected with two threads working on the same data set. When the third thread is increased there is a loss in performance. When the fourth thread is added there is a gain in performance, but this does not ever get nearly as good of performance as the two thread case. It is likely that in these cases all of the threads finish at about the same time and have to wait on the global count variable longer. To offset for the width of the cache, the data that is being modified is a struct that contains an integer and 64 characters, to force all data onto separate cache lines. To get the averages the program was run 10,000 times and the completion times were averaged. Table 1 shows the actual values of observed absolute speedup, while chart 3 shows this data graphically. Likewise table 6 and chart 2 show this information for efficiency.

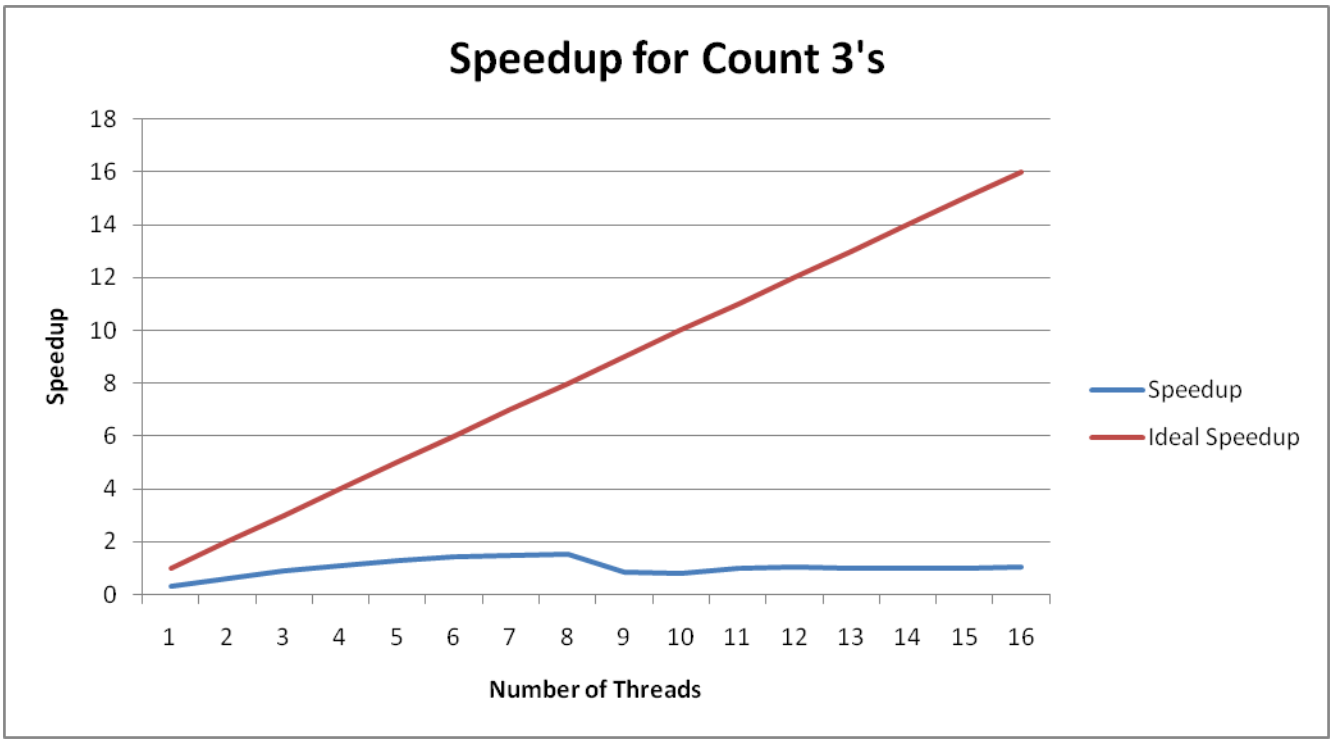


Chart 1: Speedup for the count 3's

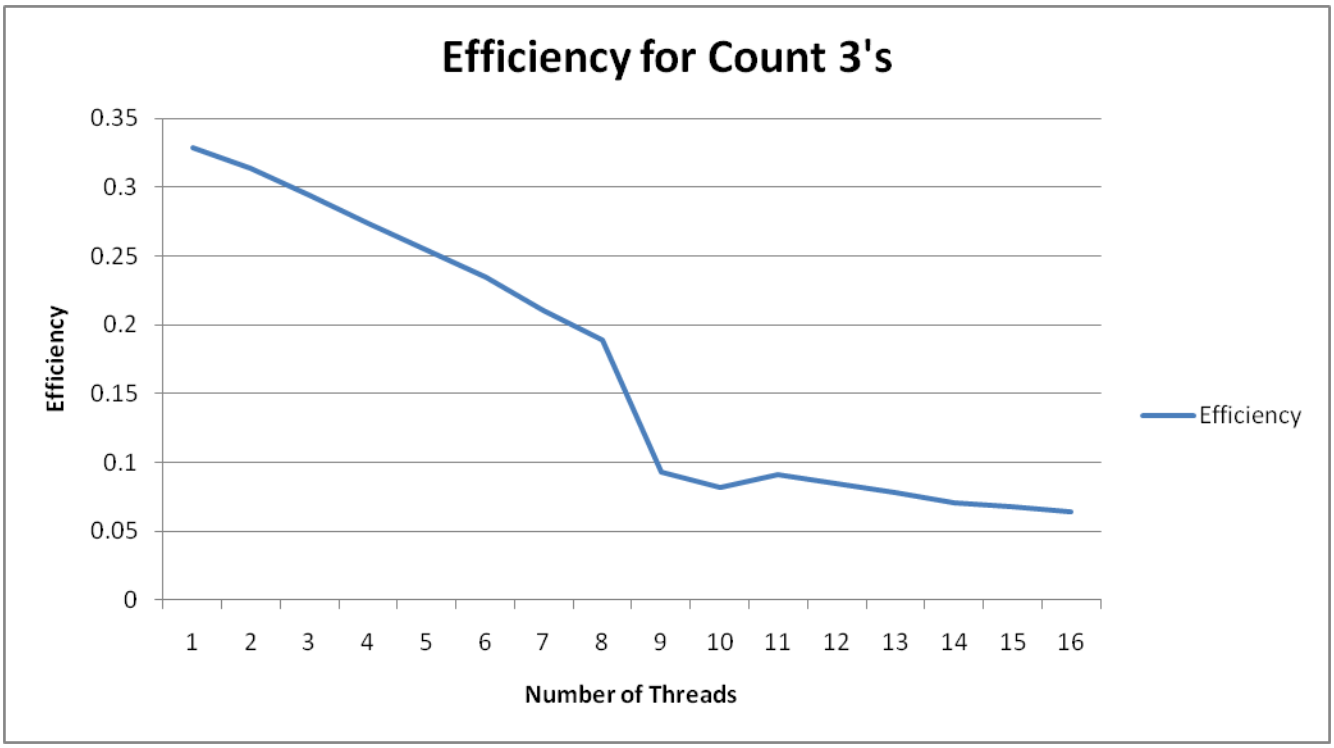


Chart 2: Efficiency of the count 3's program

Matrix Multiplication:

# of Threads	Time (us)
0	115207.264
1	124495.945
2	64746.112
3	43703.156
4	32740.433
5	30113.789
6	33273.281
7	26697.947
8	15808.76
9	14770.941
10	21157.388
11	27269.575
12	28244.973
13	28958.256
14	29014.491
15	27416.394
16	24132.753

Table 7: Average time for 200x200 matrix multiplication

# of Threads	Time (us)
1	0.92538969
2	1.779369609
3	2.63613145
4	3.518806975
5	3.825731262
6	3.462455776
7	4.315210604
8	7.287558543
9	7.799588665
10	5.445249858
11	4.224754658
12	4.078859059
13	3.978390964
14	3.970680168
15	4.202130448
16	4.773896455

Table 8: Speedup for matrix multiplication

# of Threads	Time (us)
1	0.92538969
2	0.8896848
3	0.87871048
4	0.87970174
5	0.76514625
6	0.57707596
7	0.61645866
8	0.91094482
9	0.86662096
10	0.54452499
11	0.38406861
12	0.33990492
13	0.30603007
14	0.28362001
15	0.28014203
16	0.29836853

Table 9: Efficiency for matrix multiplication

This project was a matrix multiplication program. For the purposes of this project, a random pair of 200 by 200 matrices are randomly generated and operated on. Table 4 shows the average time required for the computations in the sequential case and when using between one and four threads. The sequential and single thread case were basically the same, with the thread overhead probably accounting for the extra time required. When a second thread is added there is a large gain in performance, cutting execution time almost in half. There is additional gain in performance by adding in the third and even the fourth threads. The code does not use locks, the data is entered into the destination array in row major order, to take advantage of how C/C++ stores the data in memory. This code was run 10,000 times and the times were averaged. This implementation did not use any tricks, but instead used the three imbedded loops implementation. When the threads spawn they divvy up the target array, and work through the data. Table 8 shows the values for the absolute speedup observed in running matrix multiplication on the test machine, Chart 3 shows the graph of this data. Likewise table 9 and chart 4 shows this data for the efficiency of the code.

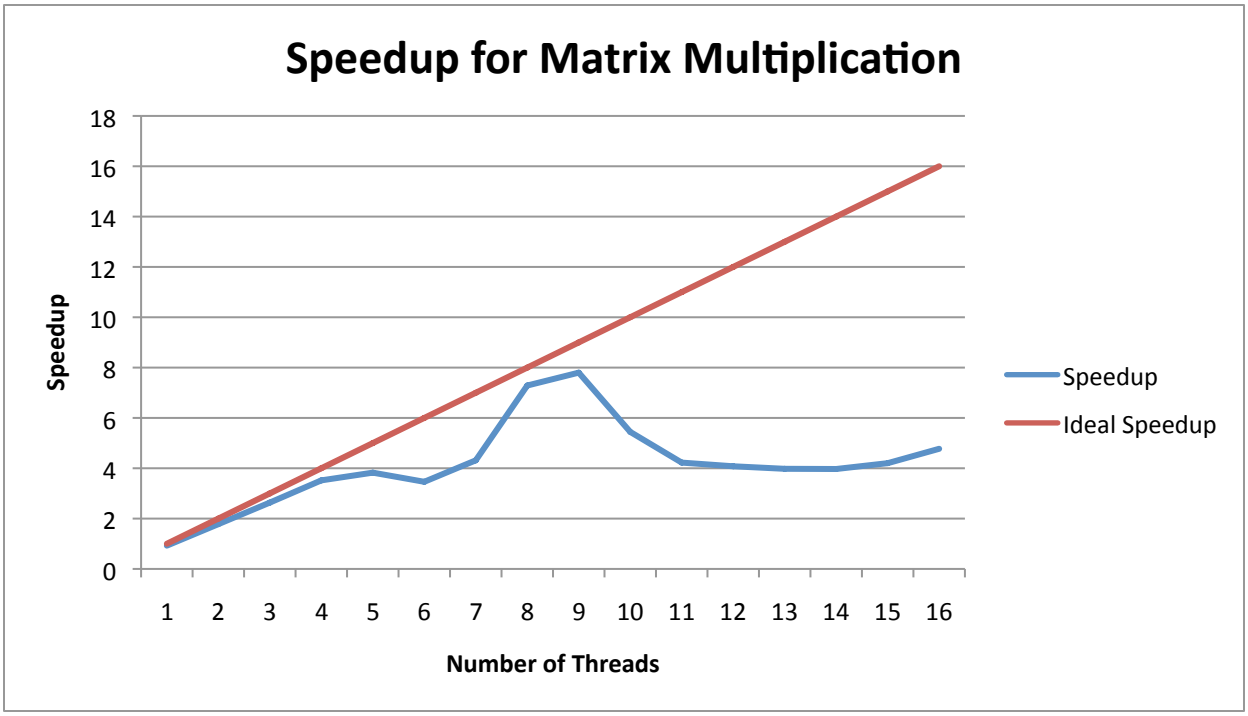


Chart 3: Speedup for 200 x 200 matrix multiplication

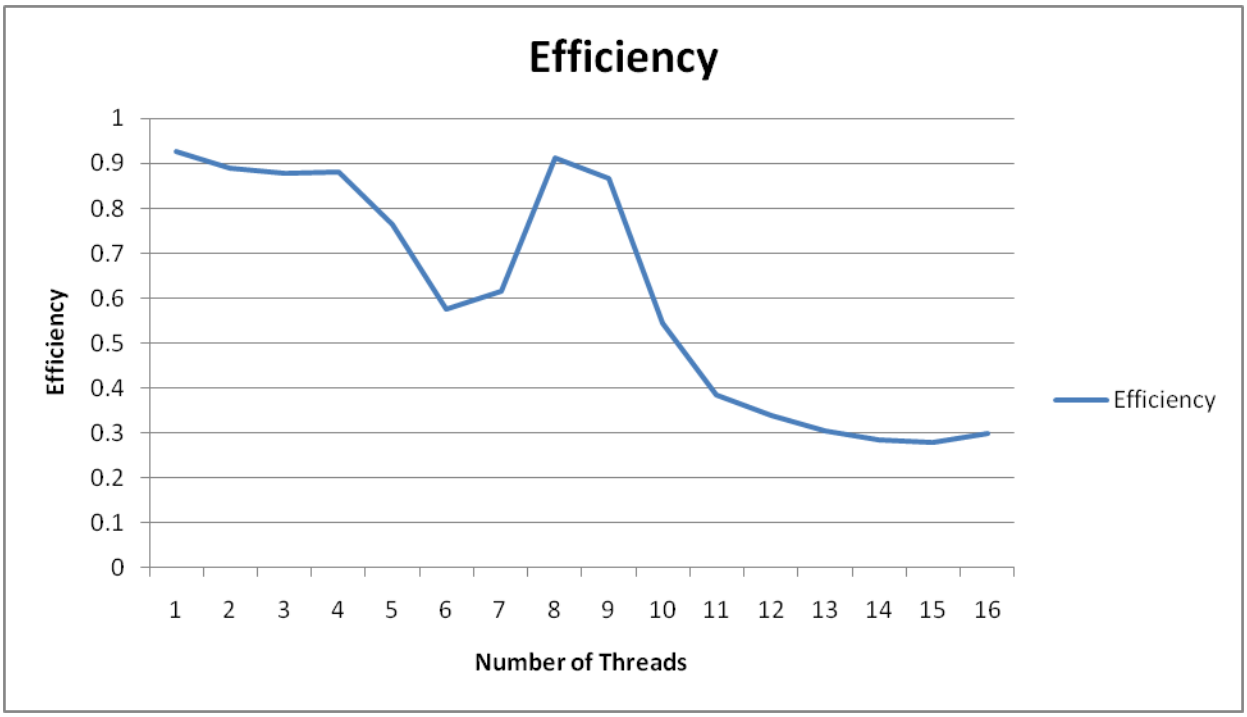


Chart 4: efficiency for matrix multiplication