OpenMP

CS 732

Joshua Hegie

The goal of this assignment was to get a grasp of how to use OpenMP. OpenMP is an interface that does threading for the programmer without any explicit threading. Unfortunately OpenMP is not simple to use, and the results are incredibly sporadic. The first task was to count the number of 3's in an array, and the second task was to multiply matrices.

Test Bed:

This code was all run on the 16 core rack 4 machines.

Count 3's Test:

#of Threads	Time (us)
0	473.348
1	482.932
2	258.596
3	174.046
4	132.219
5	108.942
6	94.202
7	85.645
8	80.952
9	75.901
10	75.971
11	79.987
12	78.722
13	84.125
14	82.081
15	82.325
16	113.828

Table 1: Average time for count 3's with the given number of threads, 0 threads is sequential

#of Threads	Time (us)
1	0.98015456
2	1.83045368
3	2.71967181
4	3.5800301
5	4.3449542
6	5.02481901
7	5.52686088
8	5.84726752
9	6.23638687
10	6.23064064
11	5.91781164
12	6.01290618
13	5.62672214
14	5.76684007
15	5.74974795
16	4.15844959

Table 2: Speedup for Count 3's

#of Threads	Time (us)
1	0.98015456
2	0.91522684
3	0.90655727
4	0.89500753
5	0.86899084
6	0.83746983
7	0.78955155
8	0.73090844
9	0.69293187
10	0.62306406
11	0.53798288
12	0.50107551
13	0.43282478
14	0.41191715
15	0.38331653
16	0.2599031

Table 3: 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. This trend continues up until 12 threads. After this there is a loss in performance, with the greatest loss

being when there are 16 threads present, likely because there are in fact 17 threads in the system, one being the thread that is responsible for all of the creation processing. 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 2 shows the actual values of observed absolute speedup, while chart 1 shows this data graphically. Likewise table 3 and chart 2 show this information for efficiency. This is apparently to be expected from OpenMP, and unfortunately there are not many resources to help figure out this issue.

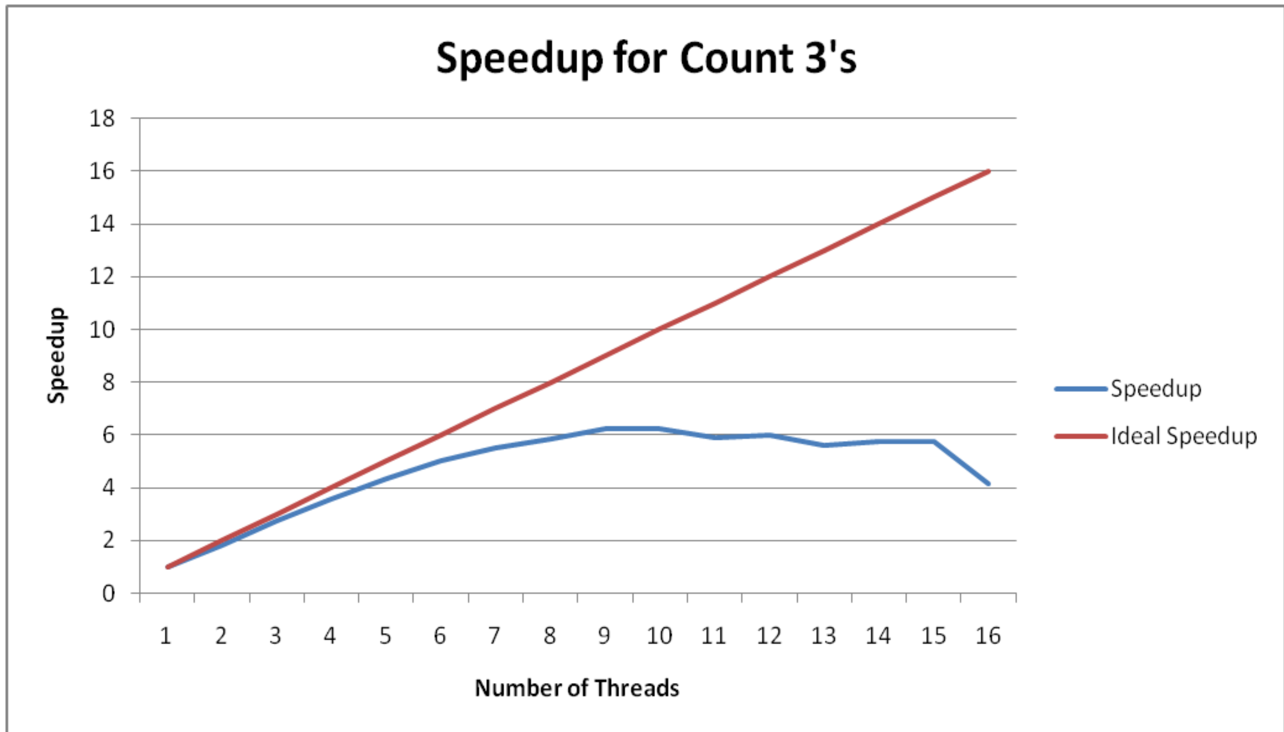

Chart 1: Speedup for the count 3's

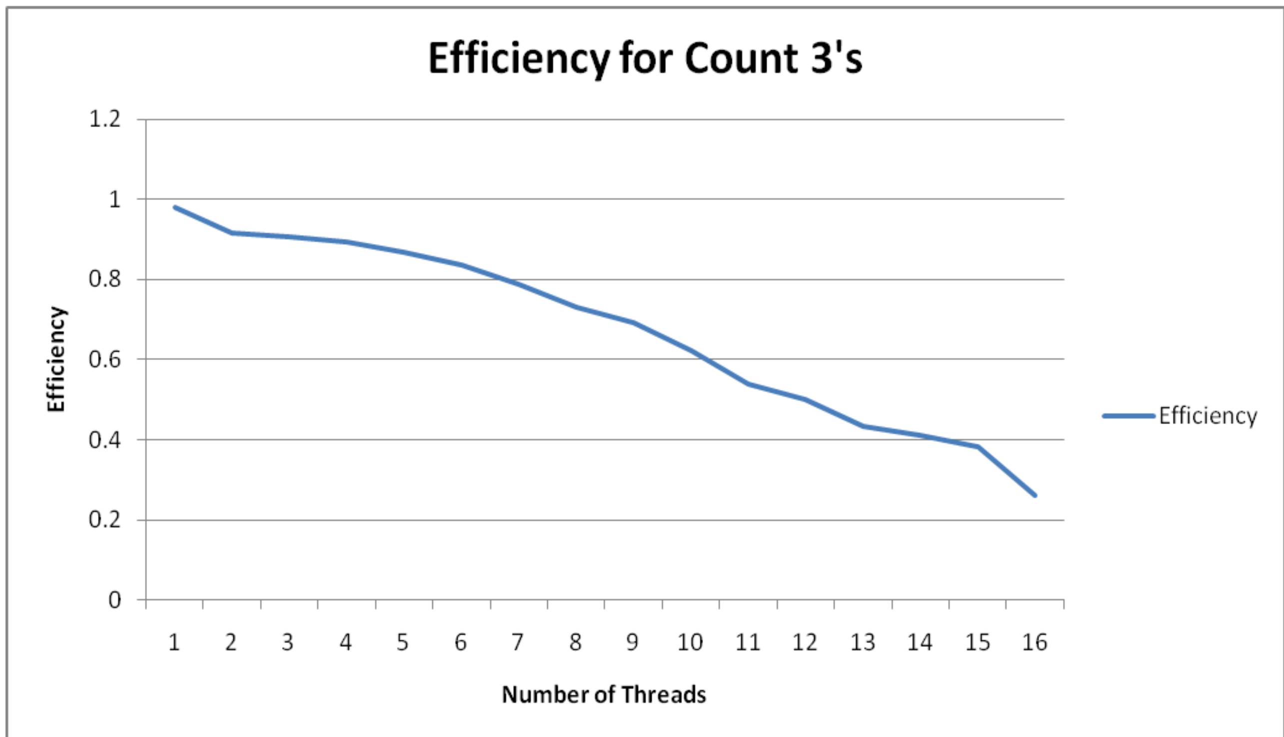

Chart 2: Efficiency of the count 3's program

Matrix Multiplication:

# of Threads	Time (us)
0	84915.029
1	116695.303
2	70940.392
3	49956.6
4	40075.496
5	34120.781
6	30439.982
7	27063.024
8	24672.554
9	23149.688
10	22474.46
11	22178.967
12	21827.122
13	21892.259
14	22081.742
15	22383.993
16	22397.619

Table 4: Average time for Matrix Multiplication with the given number of threads, 0 threads is sequential

# of Threads	Time (us)
1	0.727664497
2	1.196991257
3	1.699775986
4	2.118876557
5	2.48866018
6	2.789588673
7	3.137677039
8	3.44167973
9	3.668085246
10	3.778290068
11	3.828628673
12	3.890344728
13	3.878769614
14	3.845485968
15	3.79356038
16	3.791252499

Table 5: Speedup for Matrix Multiplication

# of Threads	Time (us)
1	0.7276645
2	0.59849563
3	0.566592
4	0.52971914
5	0.49773204
6	0.46493145
7	0.44823958
8	0.43020997
9	0.40756503
10	0.37782901
11	0.34805715
12	0.32419539
13	0.29836689
14	0.27467757
15	0.25290403
16	0.23695328

Table 6: Efficiency for Count 3's

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 5 shows the values for the absolute speedup observed in running matrix multiplication on the test machine, Chart 5 shows the graph of this data. Likewise table 6 and chart 6 shows this data for the efficiency of the code.

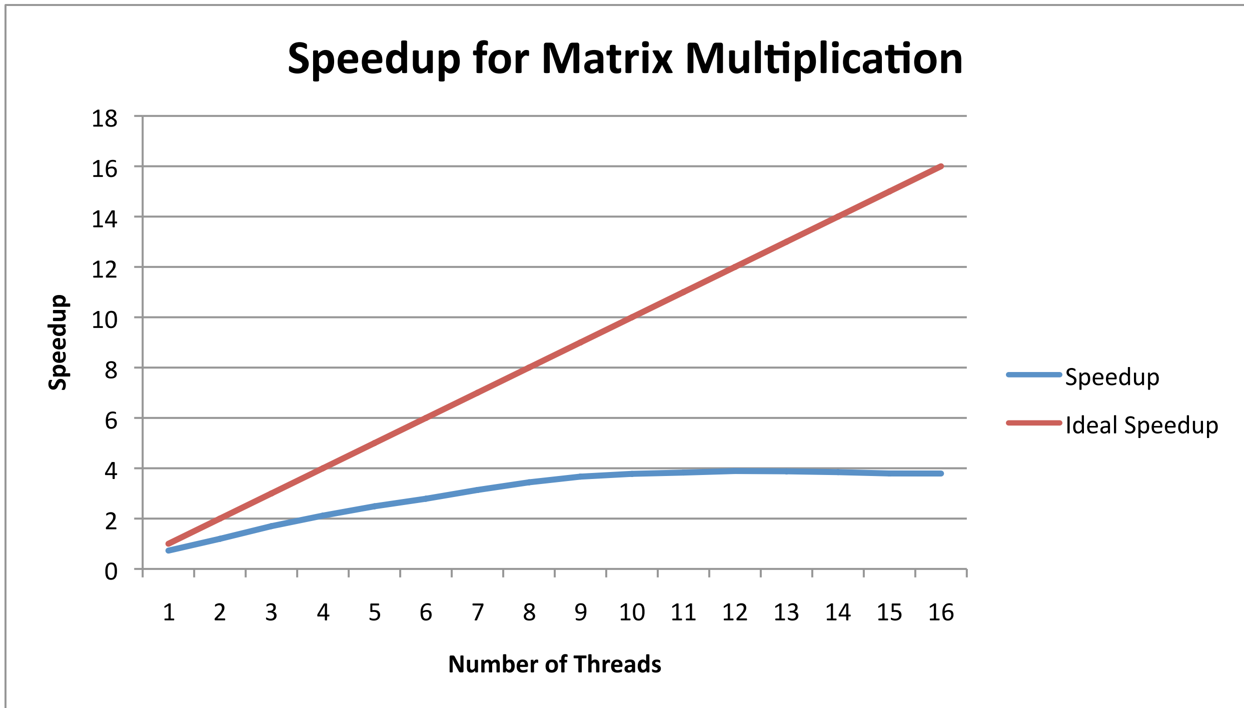

Chart 3: Speedup for 200 x 200 matrix multiplication

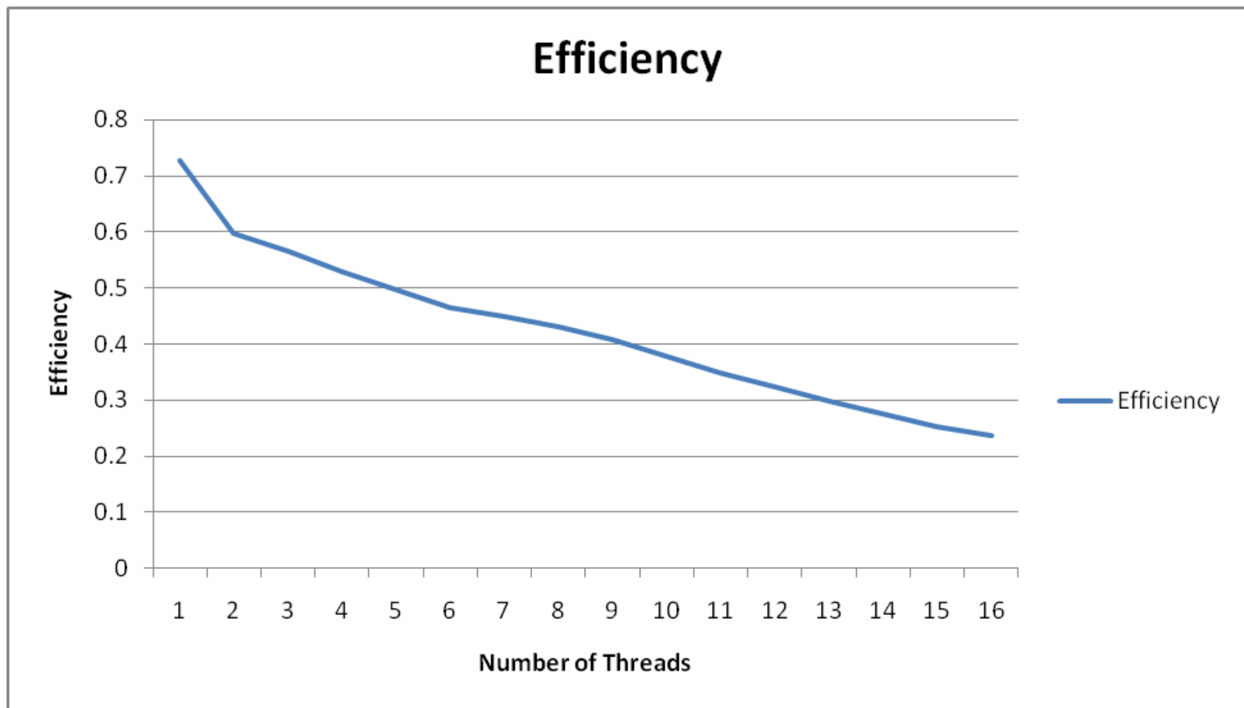

Chart 4: efficiency for matrix multiplication