Collin Paiz

Professor Anastasiu

COEN 145

10 November 2022

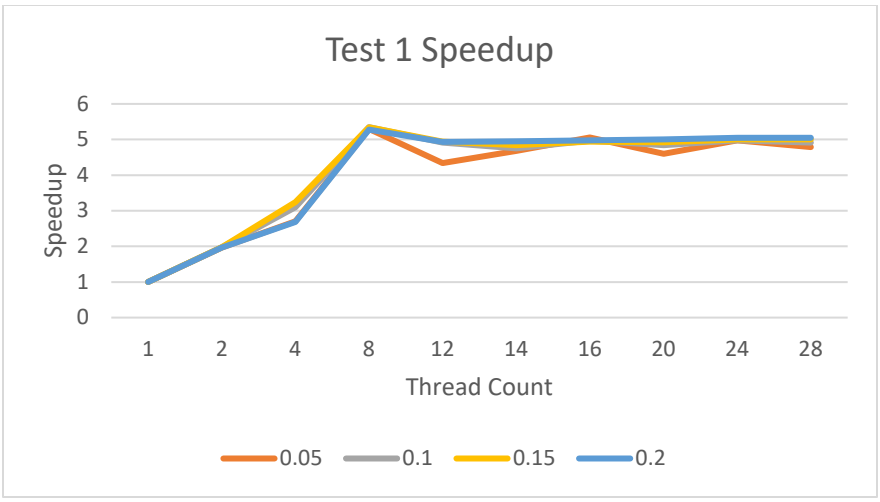
Assignment 2

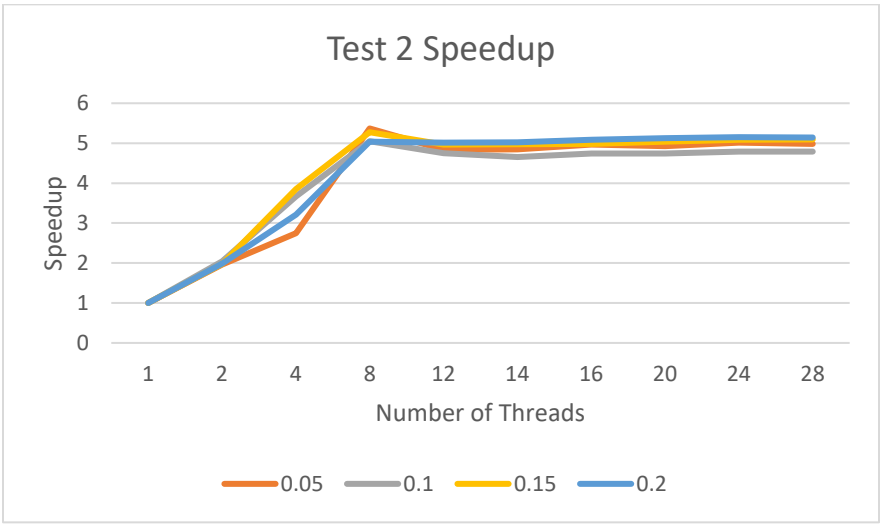
Parallelizing Sparse Matrix Multiplication

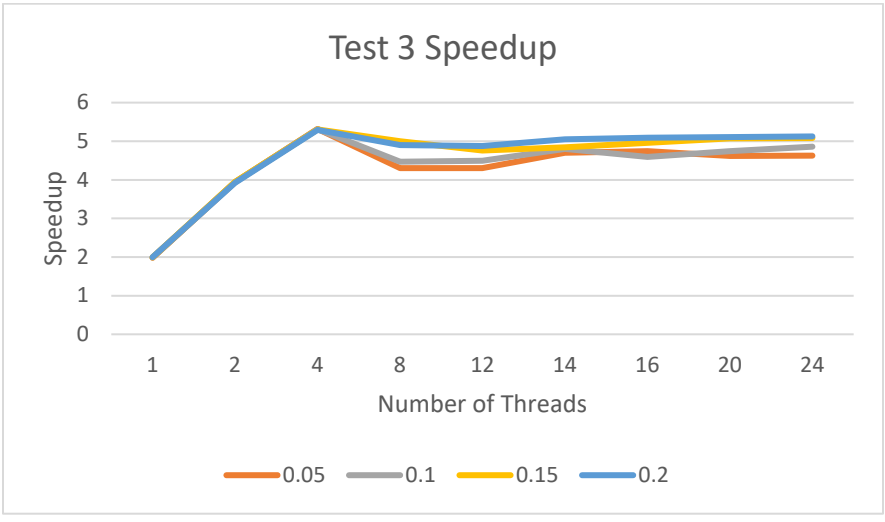
Description

To parallelize my code, I ended up focusing my parallelization on the second for-loop, which iterates through all of the rows of matrix *B* (transpose). Each thread takes a row of *B* for the current row of *A* and computes the dot product for these two vector "slices". To achieve this parallelization, I used a *#pragma omp parallel for*. I used the *schedule(static)* clause so that each thread receives the same amount of work which benefits the sum vector reduction. I used the *shared(A,B,C,cnnz,cnnz_max,nnz)* clause so that each thread shares the same matrices. This also shares *cnnz, cnnz_max.* and *nnz* which is used for keeping track of C's ptr, ind, and val arrays. *Cnnz* is used to count up the total non-zeros in *C,* while *nnz* counts up the total non-zeros in a specific column of *C.* The firstprivate*(i)* clause is used so that each thread has their own copy of *i* from the outermost for-loop and so that it obtains *i's* latest value. Each thread can keep track of its sum by having the sum declared within the loop, so it is private to each thread. To update the C (result) matrix, I used *#pragma omp critical* so that multiple threads don't try to update *C* at the same time; this did not seem to have a detrimental affect on runtime, and it ensures that *C* is free of any race conditions between the competing threads.

Results


Test 1 Speedup
Speedup
0
1
2
3
4
5
6
1
2
4
8
12
14
16
20
24
28
Thread Count
0.05
0.1
0.15
0.2


Test 2 Speedup
Speedup
0
1
2
3
4
5
6
1
2
4
8
12
14
16
20
24
28
Number of Threads
0.05
0.1
0.15
0.2


Test 3 Speedup
Speedup
0
1
2
3
4
5
6
1
2
4
8
12
14
16
20
24
Number of Threads
0.05
0.1
0.15
0.2

Raw Data

Test 1

Thread Count	Fill 0.05 Runtime (s)	Fill 0.1 Runtime (s)	Fill 0.15 Runtime (s)	Fill 0.2 Runtime (s)
1	888.767	1745.93	2565.5	3354.73
2	452.59	883.449	1299.25	1706.21
4	328.801	562.66	790.444	1251.12
8	167.814	326.457	480.224	635.623
12	204.989	355.142	519.481	680.108
14	189.869	367.42	529.62	677.725
16	175.885	351.456	518.843	674.091
20	193.288	360.365	520.773	670.327
24	178.705	350.797	512.393	664.509
28	185.518	355.577	512.983	664.123

Test 2

Thread Count	Fill 0.05 Runtime (s)	Fill 0.1 Runtime (s)	Fill 0.15 Runtime (s)	Fill 0.2 Runtime (s)
1	4758.84	3004.21	4638.72	6108.28
2	2423.31	1472.98	2342.63	3070.36
4	1730.06	818.582	1203.14	1904.1
8	886.292	595.001	879.359	1213.3
12	982.944	632.994	933.752	1218.79
14	981.388	645.897	932.001	1217.01
16	958.431	633.164	930.528	1200.71
20	965.325	633.918	920.51	1192
24	949.205	627.377	912.662	1185.72
28	955.059	627.501	911.218	1187.55

Test 3

Thread Count	Fill 0.05 Runtime (s)	Fill 0.1 Runtime (s)	Fill 0.15 Runtime (s)	Fill 0.2 Runtime (s)
1	1604.08	3149.83	4634.39	6075.36
2	809.99	1583.11	2327.26	3043.71
4	406.862	798.84	1175.21	1547.97
8	301.64	593.076	873.541	1147.77

12	372.839	704.651	927.336	1240.53
14	372.845	700.993	974.043	1245.47
16	340.931	655.722	957.658	1202.59
20	338.136	685.287	935.439	1194.01
24	347.527	663.959	912.733	1188.97
28	346.52	648.112	911.982	1185.32

Analysis

For my tests, increasing the number of threads helped runtime up to a certain point; after 8 threads, performance of the program seemed to slightly decrease, indicated by the small increase in runtime compared to the tests using only 8 threads. Increasing the threads obviously helps the runtime/performance of the program because multiple threads can do work at the same time; for how I parallelized the code, each thread is working on some row *j* of matrix *B* simultaneously versus the serial code (one thread) which can only work on one row at a time. The reason why going past 8 threads may have been detrimental is if each thread was not receiving sufficient work to justify its existence.

The size of the problem greatly affects the runtime. As you increase the size of the dimensions linearly, the problem scales exponentially, and thus so does the runtime. Whereas a 1000x1000 matrix multiplied by another 1000x1000 matrix would yield a runtime of about 1 second on 8 threads, a 10000x10000 matrix multiplied by another of the same size would take about 300 seconds (depending on the fill factor). The number of iterations that are required as we scale up the program increases exponentially, which also results in an increase in checking to see if A->ind matches B->ind to then perform the multiplication. In addition to this, as we increase the size of the matrices for the same fill factor, we also end up with many more non-zeros simply because our matrices are larger. This leads to more calculations being required to perform the multiplication between each of the rows (vectors).

My program scales a decent amount; after 8 threads, it didn't see any benefit at all from adding more threads. This is likely due to how I parallelized the program. It would have benefited more from a block decomposition approach that broke up matrix *B* into smaller chunks and computed those chunks in parallel.

The performance in terms of speedup is consistent across different shapes of matrices. Although, performance in terms of runtime differs across different shapes of matrices. Simple square matrices, such as test 1, saw the best performance in terms of runtime, whereas tall matrices, such as test 2, saw much worse runtime performance. This is likely because of how many iterations are required for the outer for-loop (20000) which are not parallelized, and how many checks must be made within the inner while loop (50000). That being said, all of the tests

saw similar speedup of about 5-6x faster compared to the serial run, regardless of the matrices' shapes.