COEN 319 - Assignment 2 Report

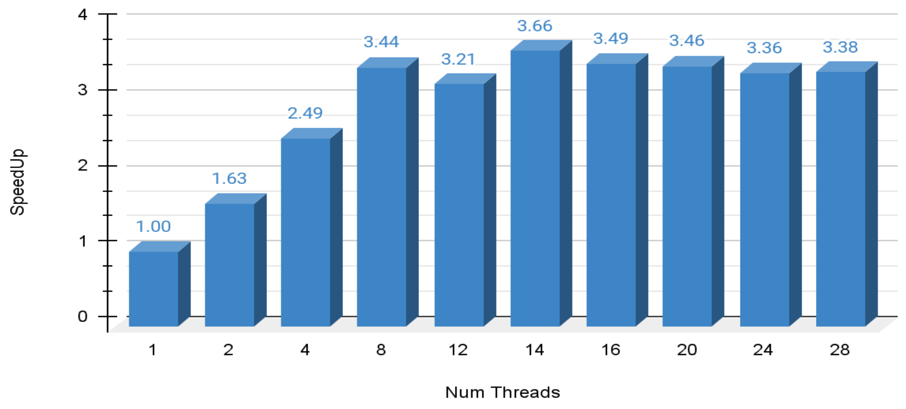
Bhanu Prakash Naredla
1630571

I. Approach

- As the given B is already transposed, I have computed the original B by computing the inverse index of B.
- **Serial approach:** I have created a temporary ans which is an array of pointers, where each pointer points to the final elements of each row in matrix C. Computing each row of C by multiplying each row of A with the whole matrix B and storing that in temporary ans.
- For each row of A, I have traversed only those rows of B where we have a corresponding element in row A and while computing I have used an accumulator to store the results. Once a thread is done computing a row of C, we'll add this row to the temporary answer.
- Finally, we traverse through the temporary ans and compute CSR matrix C.
- **To parallelize this strategy**, Each thread computes each row of C and adds it to temporary ans. Once a thread is done computing a row of matrix C, It will begin computing the next row which is not yet computed and adds it to temporary ans. And so on.. Until we finish computing every row of matrix C. (achieved this using OMP parallel for dynamic schedule)
- But here we need a separate accumulator for each thread. So, I have created an array of accumulators with the size of numThreads and each thread uses its own accumulator to compute a row of C and zero's it out once finished to use the same accumulator for computing any other row of matrix C.
- Once we are done computing every row of matrix C, we shall first compute the ptr array in CSR representation of matrix C. Now, each thread fills a row in CSR matrix C and begins filling the next row which is not yet filled.
- Once all threads finish, we have a final CSR representation of matrix C.

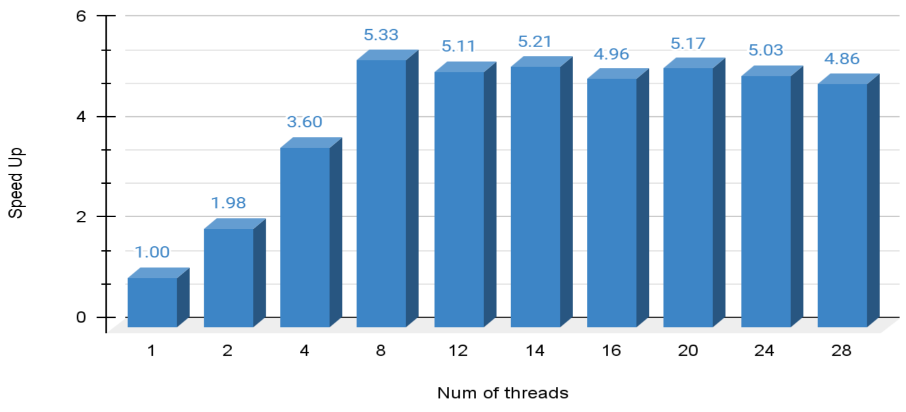
II. SpeedUp Results:

A[10000, 10000] X B[10000, 10000], f = 0.05



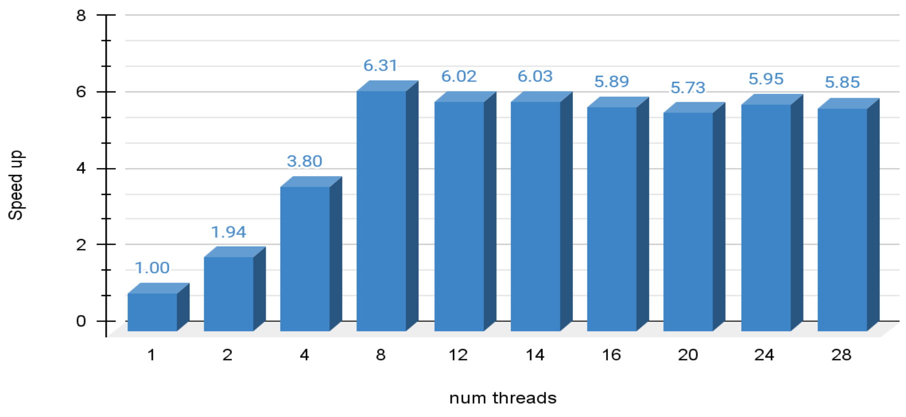
- Time taken with 1 thread : 37.12 sec
- Achieved best speed up with 14 threads: 10.15 sec

A[10000, 10000] X B[10000, 10000], f = 0.1



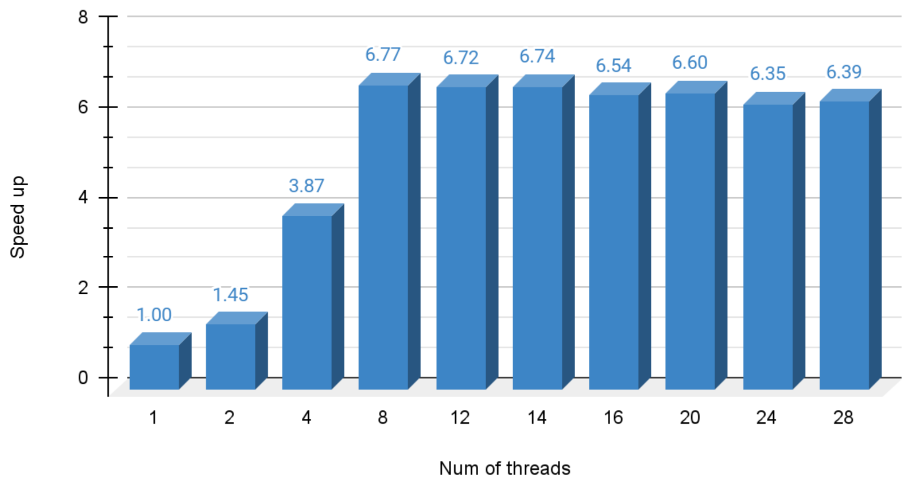
- Time taken with 1 thread : 126.96 sec
- Achieved best speed up with 8 threads: 23.82 sec

A[10000, 10000] X B[10000, 10000], f = 0.15



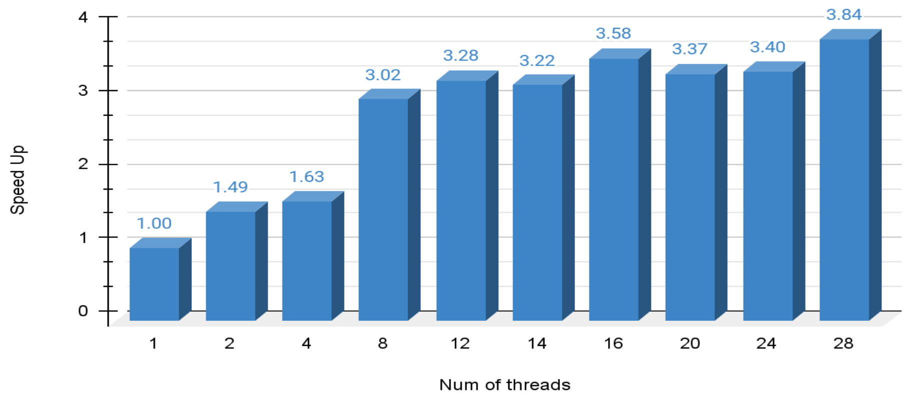
- Time taken with 1 thread : 270.36 sec
- Achieved best speed up with 8 threads: 42.86 sec

A[10000, 10000] X B[10000, 10000], f = 0.2



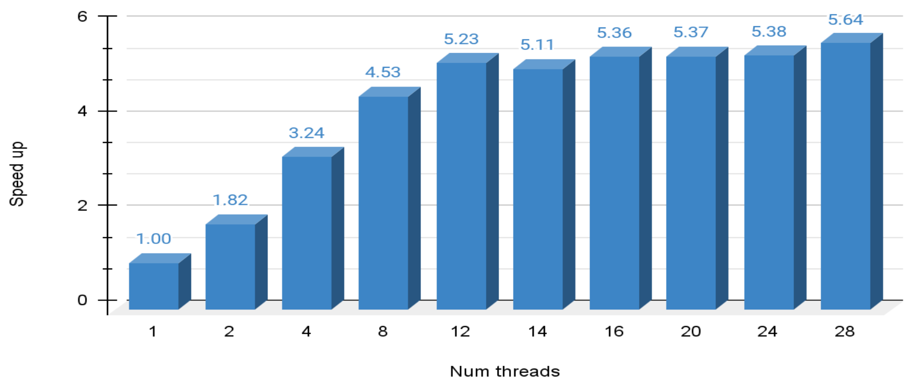
- Time taken with 1 thread : 467.4 sec
-Achieved best speed up with 8 threads: 71.1 sec

A[20000, 5300] X B[5300, 50000], f = 0.05



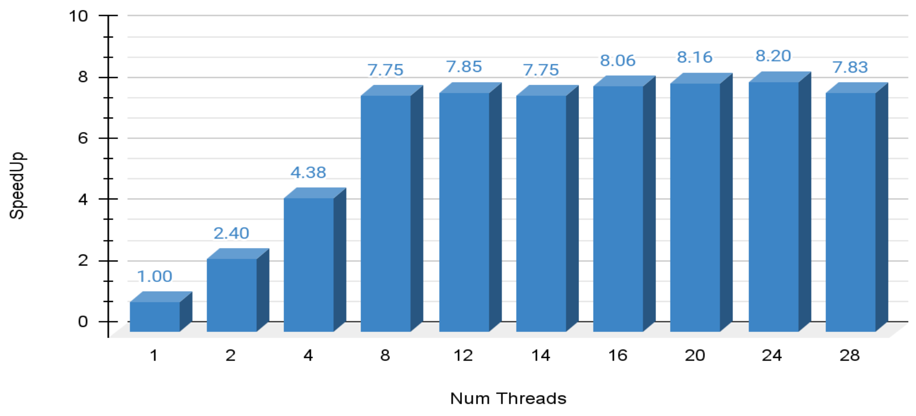
- Time taken with 1 thread : 204.86 sec
-Achieved best speed up with 28 threads: 53.13 sec

A[20000, 5300] X B[5300, 50000], f = 0.1



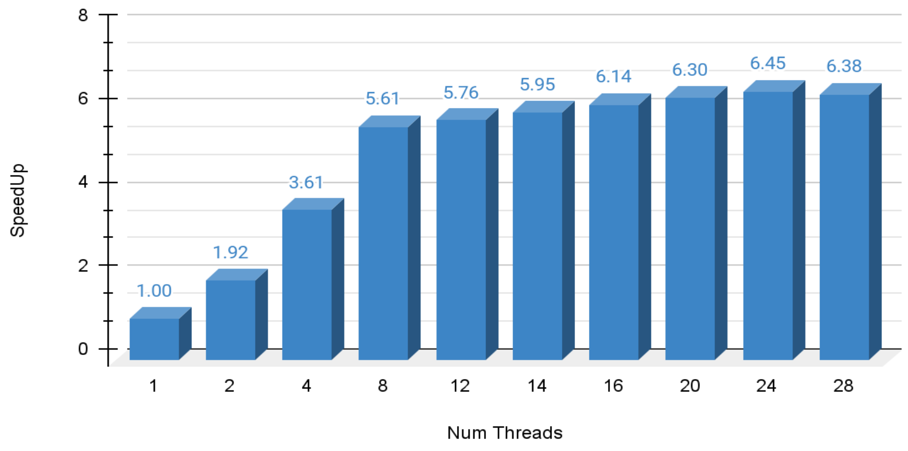
- Time taken with 1 thread : 675.22 sec
-Achieved best speed up with 28 threads: 119.7 sec

A[20000, 5300] X B[5300, 50000], f = 0.15



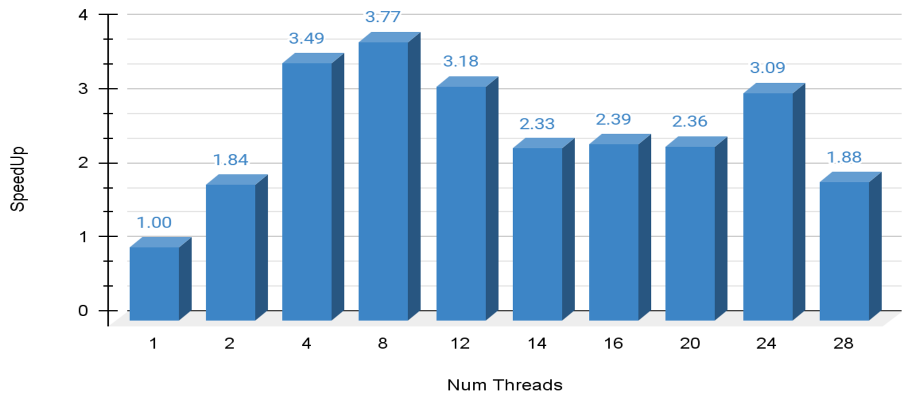
- Time taken with 1 thread : 1841.92 sec
-Achieved best speed up with 24 threads: 224.52 sec

A[20000, 5300] X B[5300, 50000], f = 0.2



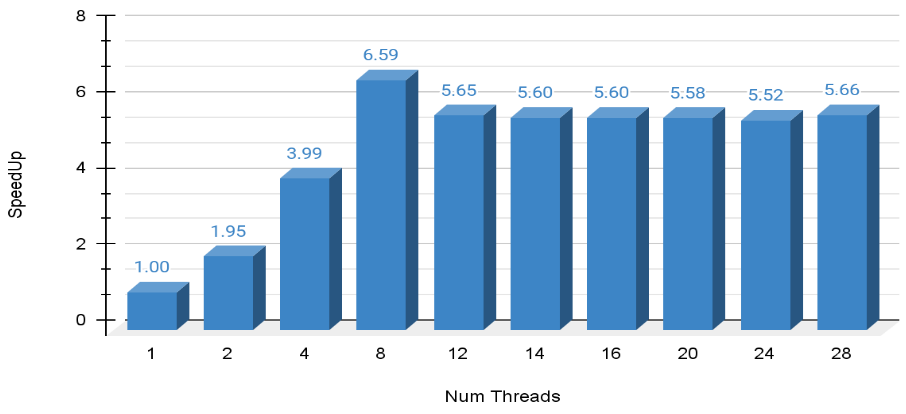
- Time taken with 1 thread : 2471.04 sec
-Achieved best speed up with 24 threads: 382.81 sec

A[9000, 35000] X B[35000, 5750], f = 0.05



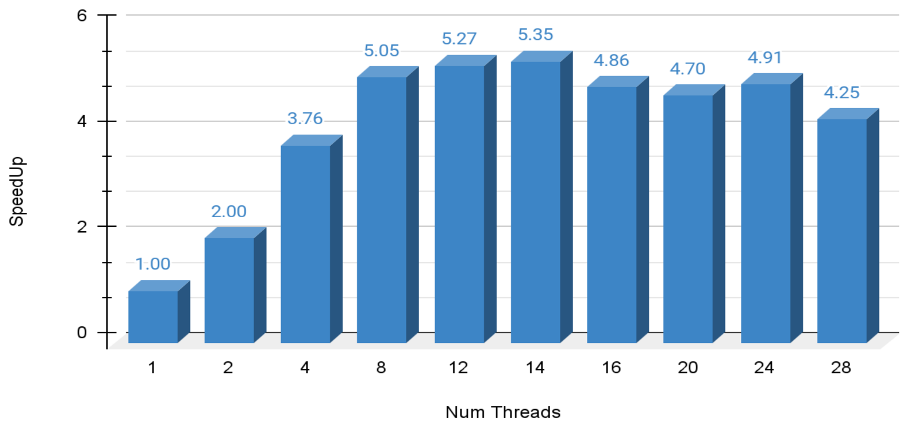
- Time taken with 1 thread : 60.72 sec
-Achieved best speed up with 8 threads: 16.09 sec

A[9000, 35000] X B[35000, 5750], f = 0.1



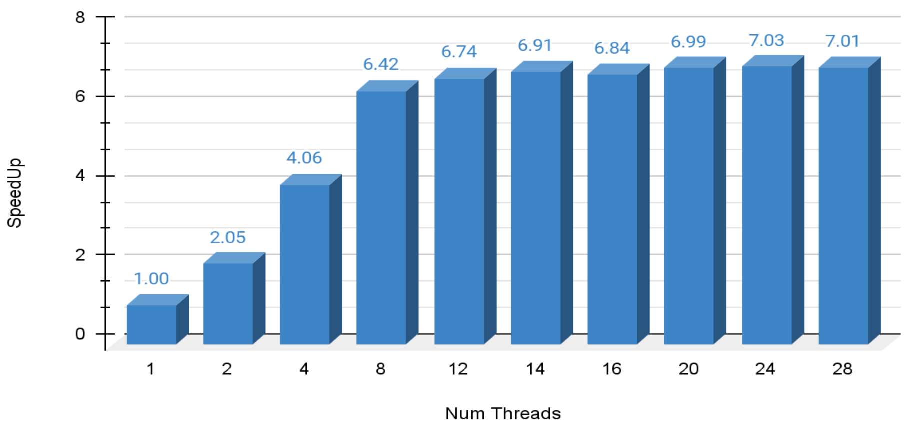
- Time taken with 1 thread : 233.38 sec
-Achieved best speed up with 8 threads: 35.42 sec

A[9000, 35000] X B[35000, 5750], f = 0.15



- Time taken with 1 thread : 487.66 sec
-Achieved best speed up with 14 threads: 91.18 sec

A[9000, 35000] X B[35000, 5750], f = 0.2



- Time taken with 1 thread : 874.81 sec
-Achieved best speed up with 24 threads: 124.41 sec

III. Analysis of results

● As the fill factor increases within a particular size of matrix A and matrix B, we took more time to compute the resulting matrix as the number of computations increased. The serial algorithm iterates over each row of A and multiples with matrix B to compute a row in matrix C. But here, as the matrix is sparse and the number of non-zero elements in a row is unpredictable, we cannot equally divide the work among all the threads.

● With an increasing number of threads we cannot guarantee that our program will run faster because it all depends on how many cores does the machine have and those many threads can run in parallel to improve performance.

● If we keep increasing the number of threads our program runs on, we may end up slowing it down, as threads are not free i.e., OS takes up some memory and time for creating and maintaining threads. If the amount of work each thread does isn't significant, time taken to create these threads and context switch between threads will be an overhead for the program execution time.

● Also, if we observe speedup results are consistent up to a certain factor within the same matrix sizes and then the performance degrades which is also because of the thread creation and context switch between threads taking more time than the actual time taken for work, which is overhead and the performance degraded.

● Each of our test results gave better performance at different numbers of threads and not always at 28 threads.