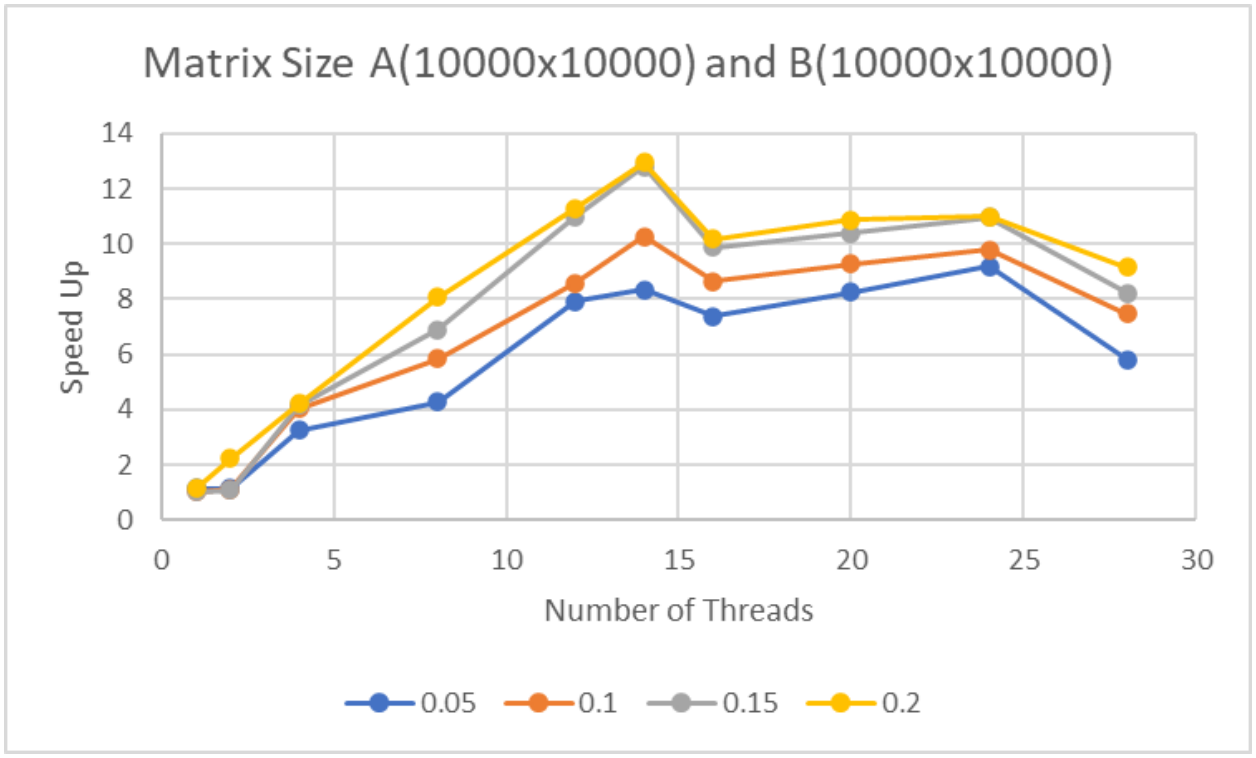
Program 2: Sparse Matrix Mulitiplation

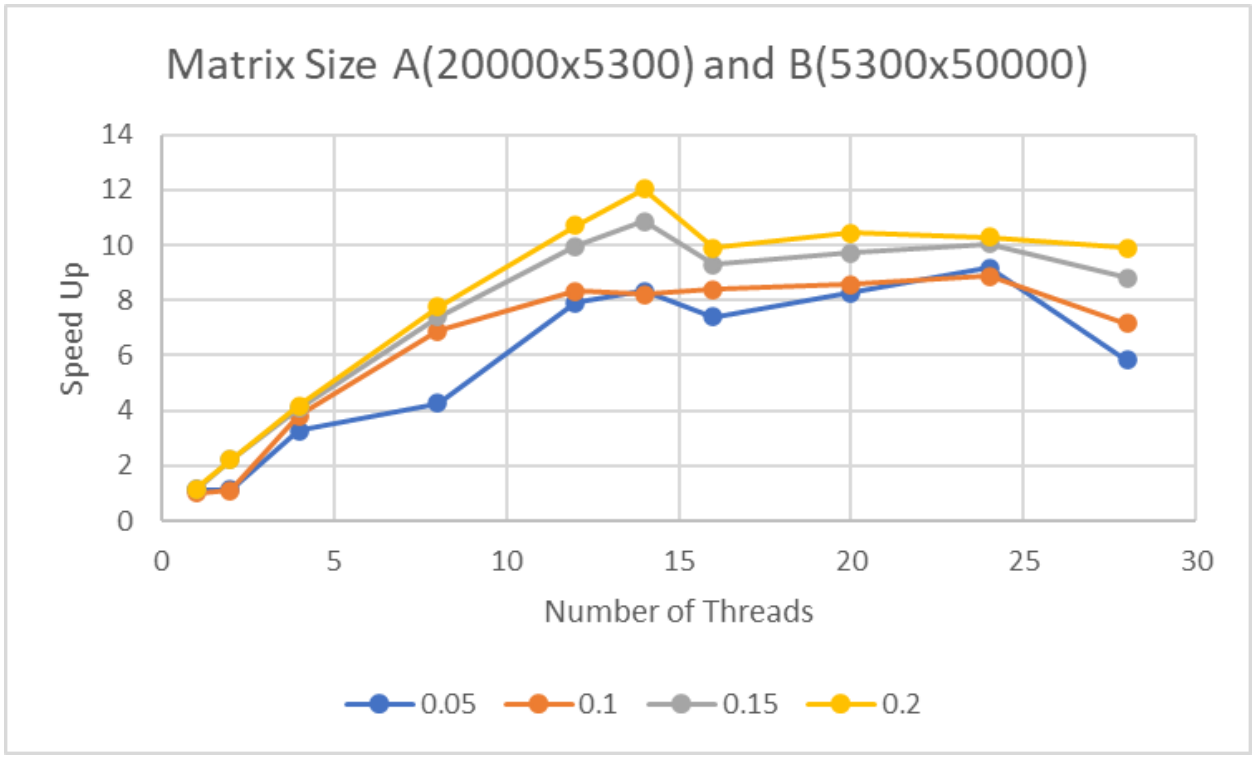
a. **A short description of how you went about parallelizing sparse matrix multiplication. One thing that you should be sure to include is how you decided what work each thread would be responsible for doing.**

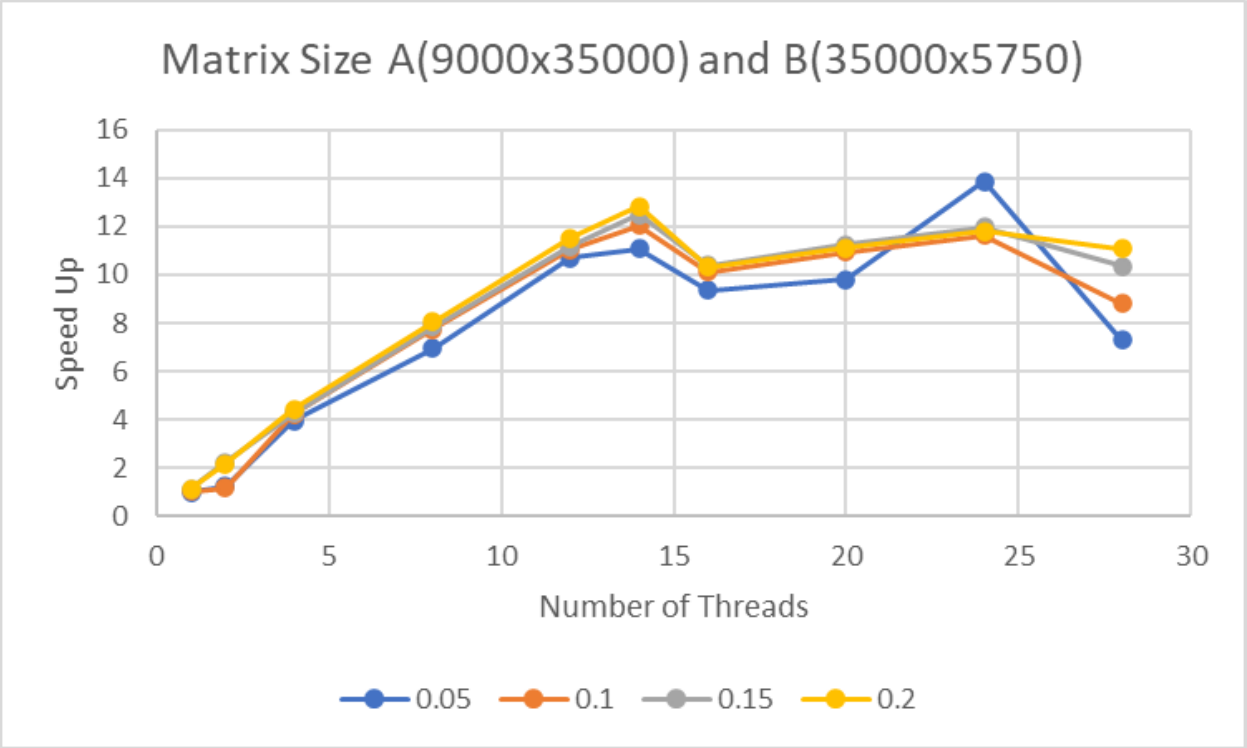
In this program we are given functions that implement the CSR (Compressed Sparse Row) data structure for sparse matrices. Then, to multiply two matrices in CSR data structure, I initialized an accumulator array to store intermediate results during the computation. The size of the accumulator arrays are determined by the number of columns in the second matrix we are multiplying. Then, to do the multiplication there are comments in the "omp_sparsematmult()" function that explain my matrix-matrix multiplication loop.

In this case, since there are nested for loops I utilized the "parallel for" directive of openMP on the outermost loop. There are loop dependencies in the inner loops so I did not attempt to parallelize those because it would require synchronization mechanisms like locks or barriers which is error prone and could cause even greater parallel overhead. Thus, increasing parallel runtime of the programming and decreasing speed up. For my program, each worker thread of the parallel region is calculating one row of the final product matrix C. Specifically, the "#pragma omp for" directive distributes the iteration of the outer loop among the available threads.

b. **Timing results for your parallel executions. These results should be reported in a strong scaling chart reporting the speedup at different numbers of threads vs the serial execution (1 thread).**


Matrix Size A(10000x10000) and B(10000x10000)
Speed Up
Number of Threads
0
2
4
6
8
10
12
14
0
5
10
15
20
25
30
0.05
0.1
0.15
0.2


Matrix Size A(20000x5300) and B(5300x50000)
Speed Up
Number of Threads
0
2
4
6
8
10
12
14
0
5
10
15
20
25
30
0.05
0.1
0.15
0.2



c. A brief analysis of your results. Some things to consider might be:

i. How does the number of threads affect the runtime? Why do you think that is?

According to my results, a greater number of threads does not necessarily guarantee a faster runtime. For all my results, the speed up dipped at 16 threads. Then again at 28 threads. At 16 threads, I hypothesize that the speed up dips because I was not able to increase the CPUs per task any higher than 24 causing the operating system to need to time-share the available cores. The number of cpus per task remained the same, at 24, from 16 threads onwards. So, I think the dip in speed up at 28 threads is caused by granularity of parallelism where the amount of work each thread is assigned is too small and thus overheads may dominate.

ii. How does the size of the problem affect the runtime? Why do you think that is?

The larger the outcome matrix, the longer the runtime because there are more indices and pointers to iterate through.

iii. How well does your program scale, i.e., if you keep increasing the number of threads, do you think the performance will keep increasing at the same rate?

Because of the decrease in speed up at 28 threads across the three matrix sizes tested, I do not think that performance will keep increasing at the same rate unless some changes are made to the code.

iv. Are performance characteristics consistent for the different shapes of the matrices? Why do you think that is?

Performance characteristics are consistent for the different shapes of matrices. For the tested sizes, neither of the matrices are particularly narrower or wider than the other. Thus why the graphs of speed up vs. number of threads are all similar in shape.