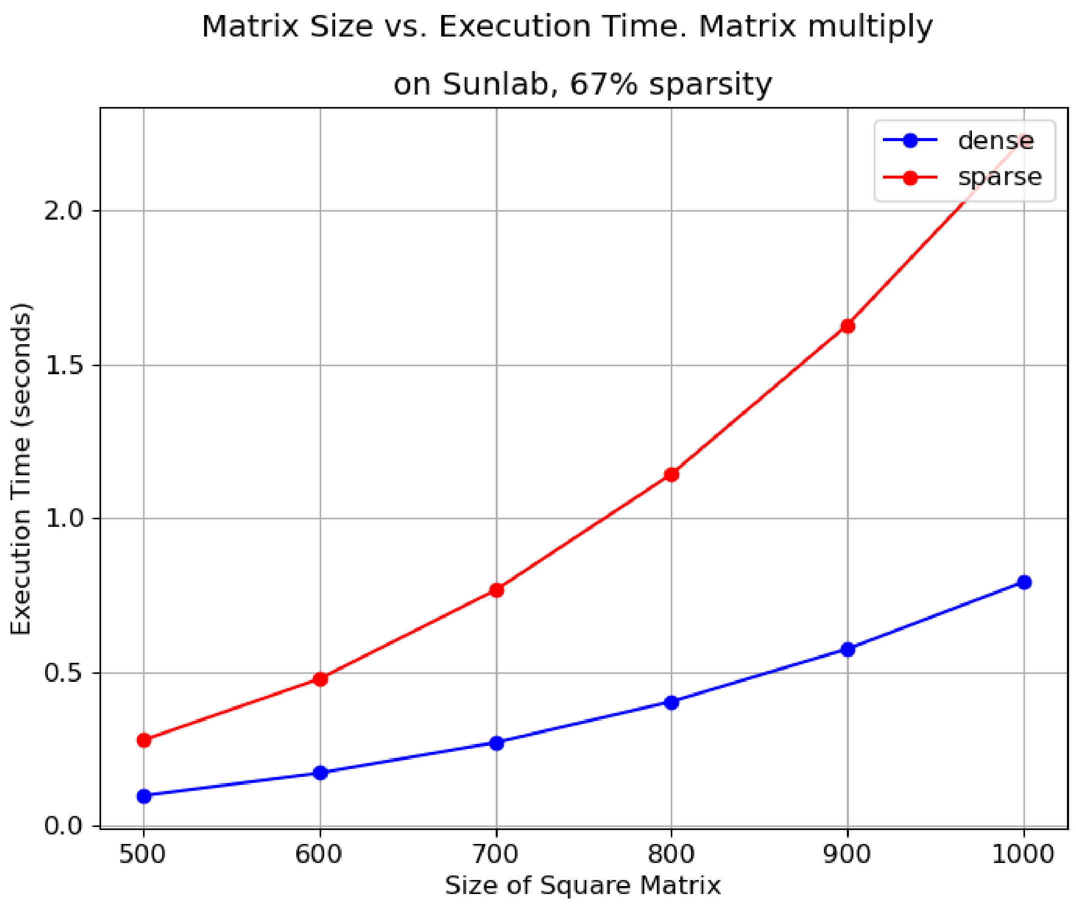
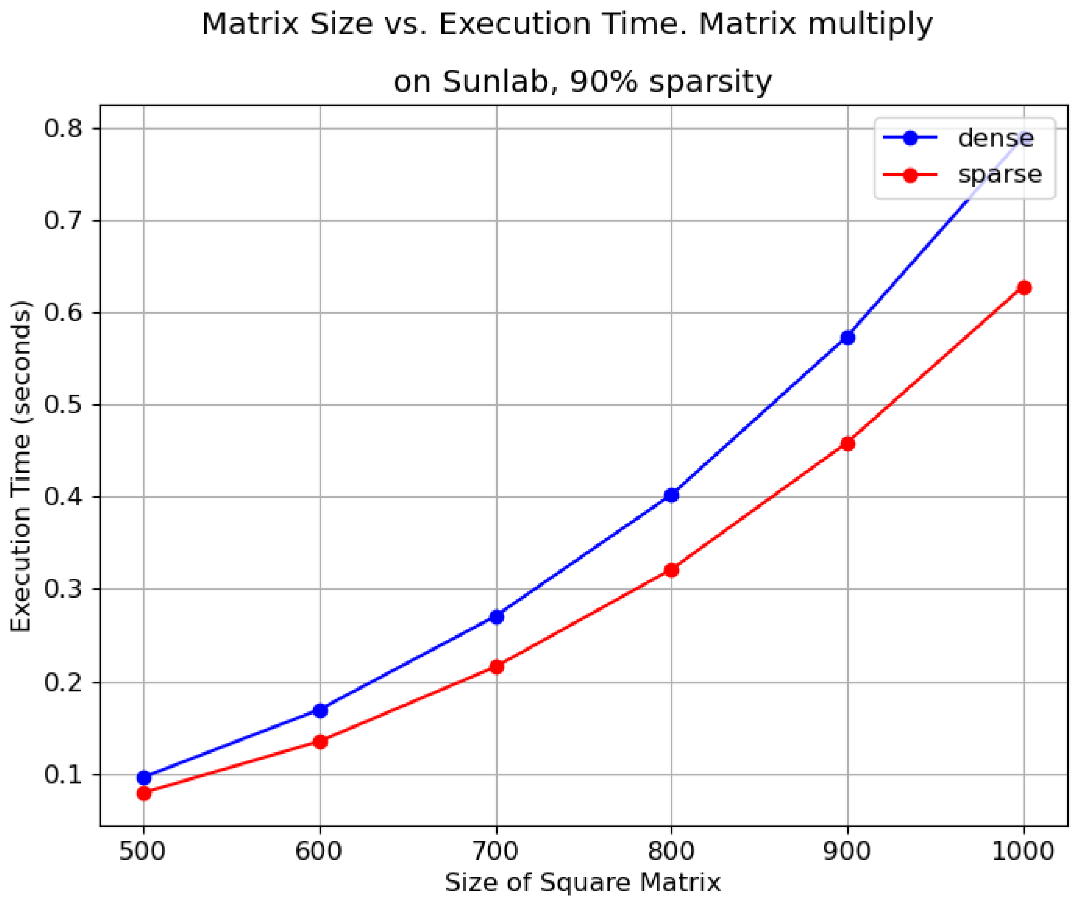
Luke Hale & Michael Speckhart
CSE 375
Final Project Write Up
5/5/2023

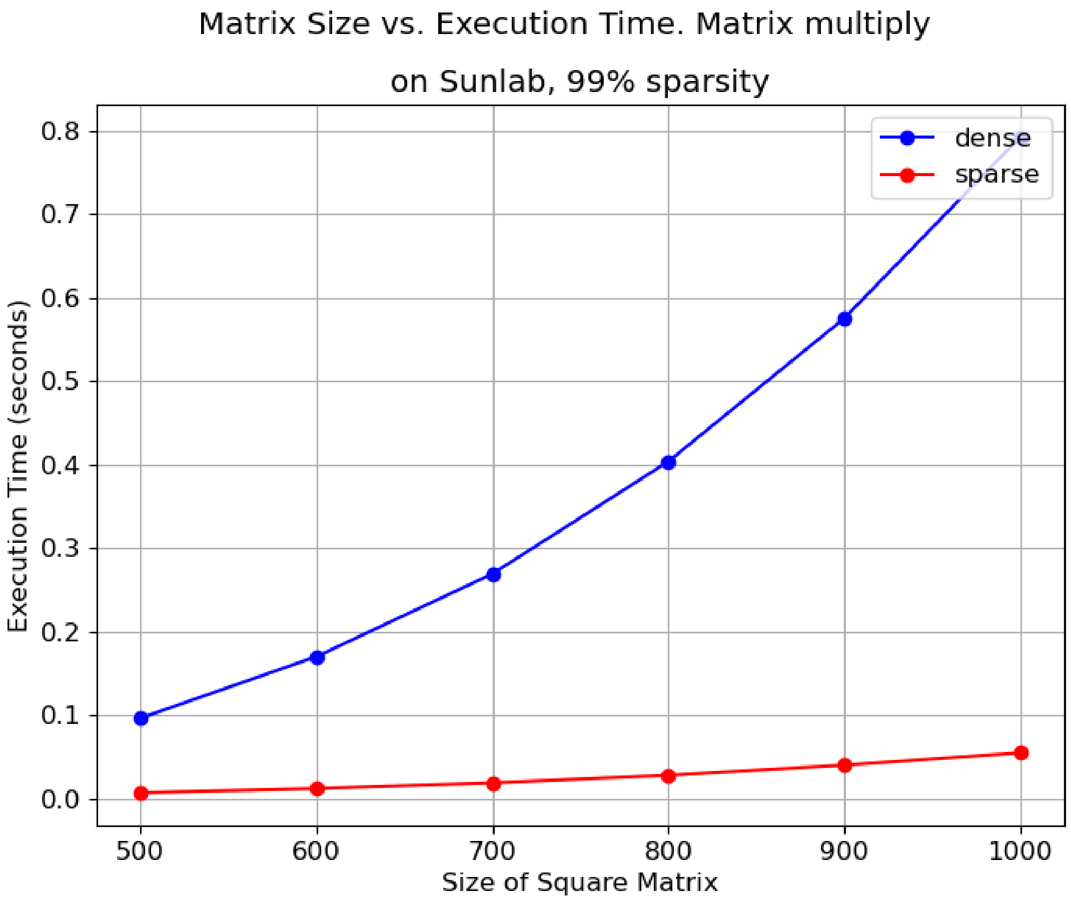
Parallelizing Sparse Matrix Functions

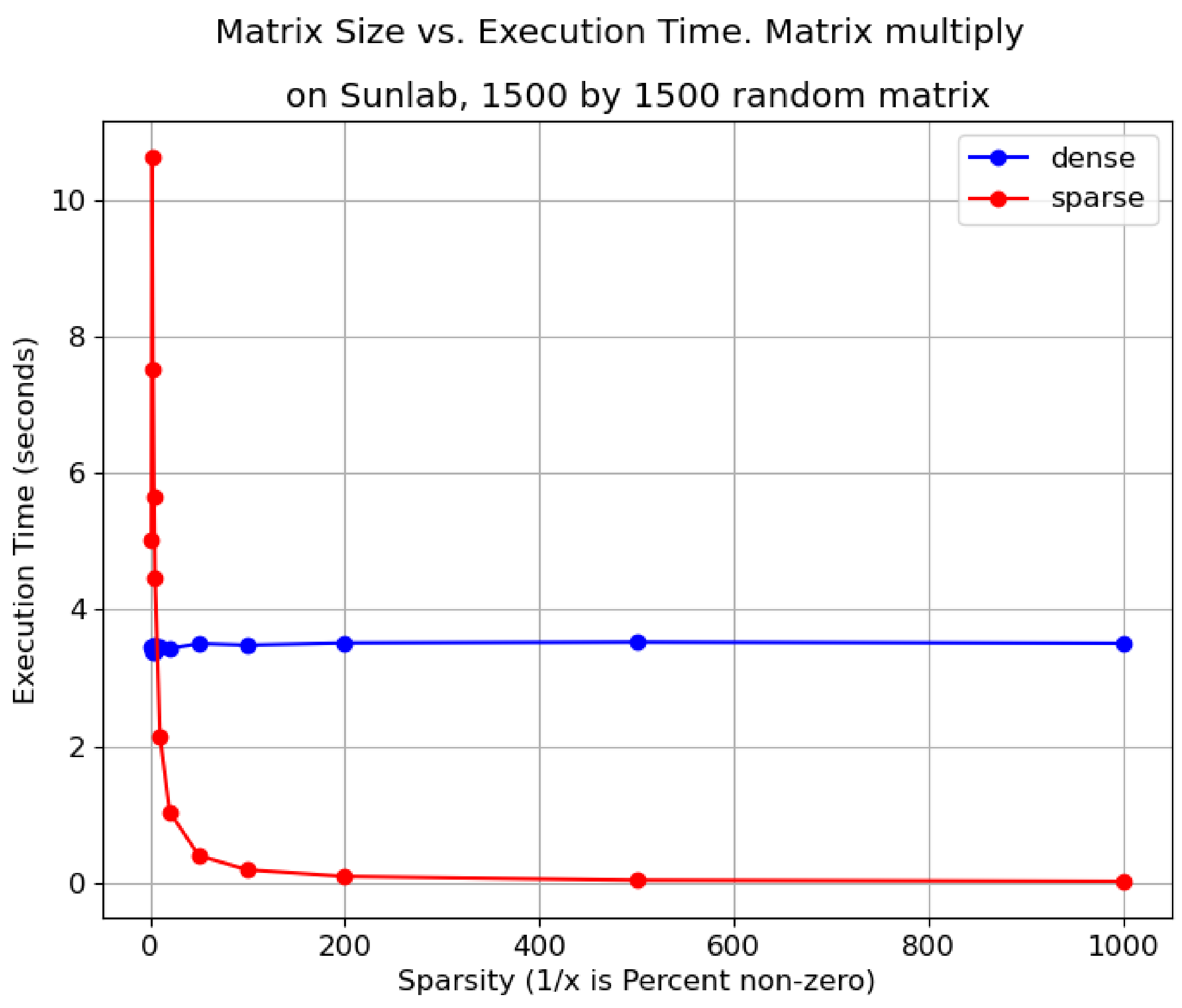
Luke and Michael's capstone project is Numerical Analysis Platform with Professor Carr.

Our goal for the capstone project this year was to implement sparse data formats into the software to make the code more efficient. Sparse matrices are where the majority of the elements are zeros. The basic idea of this is instead of storing an entire 2-D array of all the elements, where there are a lot of repeated zeros, only store the Nonzero elements with their respective row and column. While this has downsides, such as not allowing for constant time lookup of elements, it saves significantly on memory and actually allows for speed up in operations. This is because operations on matrices require iterating over each element, and there are fewer iterations with the sparse formats. The following graphs show the benefits of sparse formats with basic matrix multiplication, as the sparsity increases.



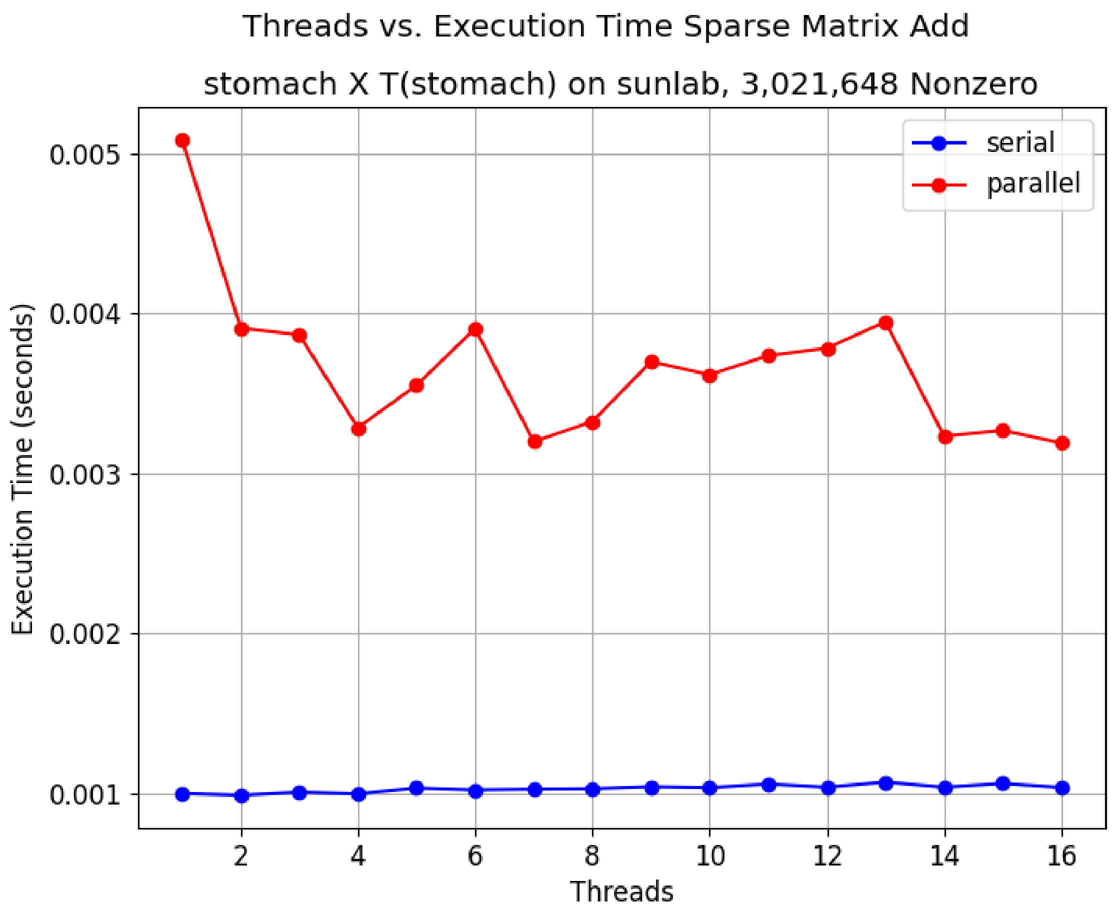

Matrix Size vs. Execution Time. Matrix multiply
on Sunlab, 90% sparsity
Execution Time (seconds)
Size of Square Matrix
dense
sparse
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
500
600
700
800
900
1000


Matrix Size vs. Execution Time. Matrix multiply
on Sunlab, 99% sparsity
dense
sparse
Execution Time (seconds)
Size of Square Matrix
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
500
600
700
800
900
1000

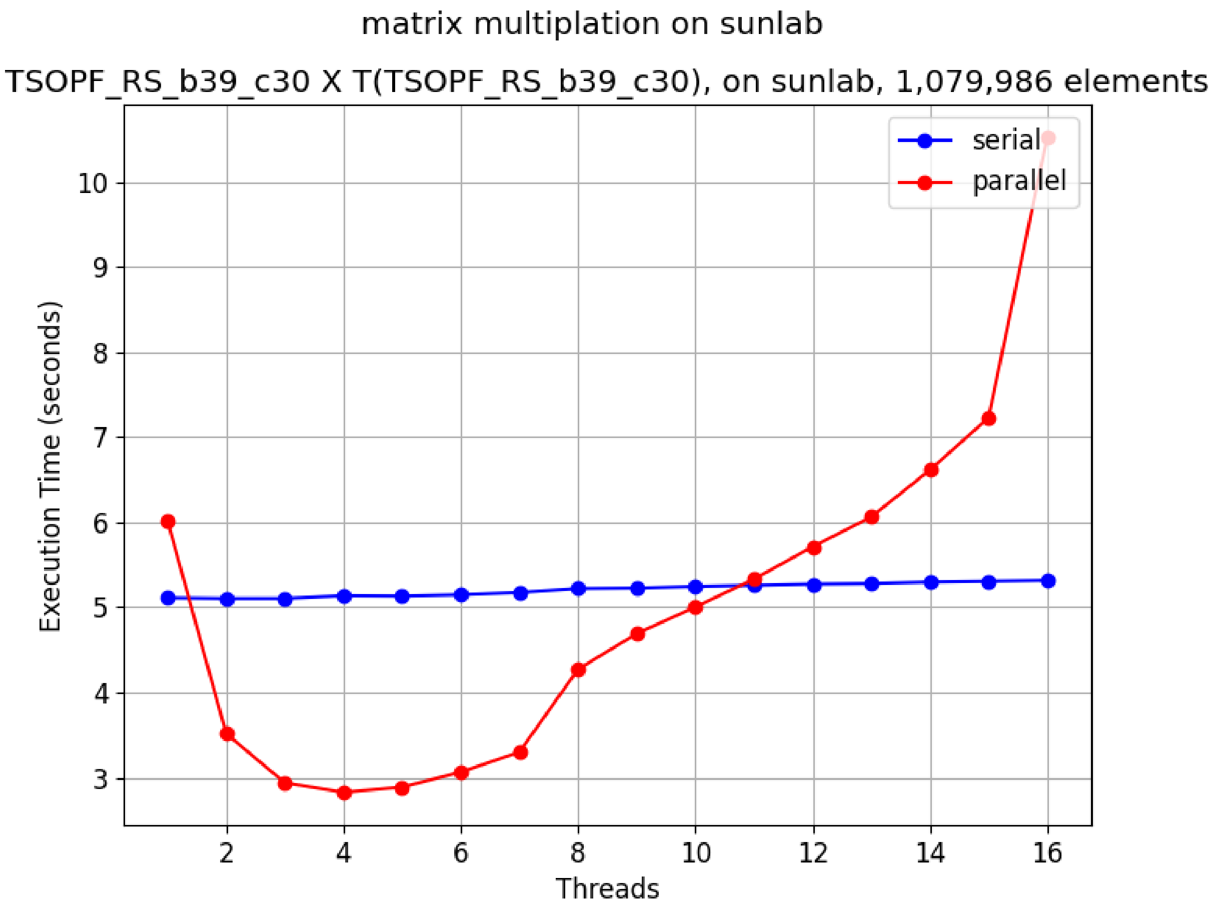


The last graph shows that when doing matrix multiplication on the same matrices, but one in the dense format and one in the sparse format, the sparse format is 195.5 times faster than the dense format for .1% sparsity. While a sparsity of .1% may seem unrealistic, it is actually quite common in the real world, and we even see matrices up to 100 times more sparse than this. (https://sparse.tamu.edu/ is the collection we use to find sparse matrices to test on). This kind of speedup would be much harder to achieve if we were just adding more threads to our computation and shows why it is important to refactor the code first before paralyzing the code.

We started by paralyzing with the simplest functions in our software and worked our way up to the more complex functions. We started with finding the maximum in a matrix, which was easily paralyzed with a TBB reduction, but we found it hard to see a speed increase because of how fast the function already ran. We kept growing the size of the matrix trying to see a speed up, but soon enough the matrices were too big to fit on Sunlab. We found out that even if code might be obviously parallelized, it might not be worth it if the code already runs fast. This was also true with matrix addition, as shown by the graph below on a matrix with over 3 million Nonzereos the add only takes one millisecond.



Matrix multiplication however is more complex because for AxB=C, each element in C is the result of multiplying all of the elements in the same row of A by all of the elements in the same column as B. For a square matrix, the time complexity is n^3 where n is the size of the matrix. This worst-time complexity makes it so there is more of an opportunity for speed up. The difficulty with parallelizing the multiplication however is that the Compressed Sparse Row format (CSR) was that the matrix is represented by three vectors (one as a row pointer, a second for the column, and a third for the value). This makes it so the row pointer must be calculated in serial as each row pointer depends on how long the previous row was. We could not use a TBB reduction to perform this operation because the reduction must be associative, but there is a strict ordering of our vectors. To fix this we tried parallelizing over the column part of the matrix multiplication using TBB. We were then able to append these results to a concurrent vector, and then sort the concurrent vector based on the column index. Once this was done we could append them to our result matrix and then calculate the row pointer. This solution was not scalable as shown by the graph below. This is likely due to the fact that with more threads, each column has more partitions to sort.

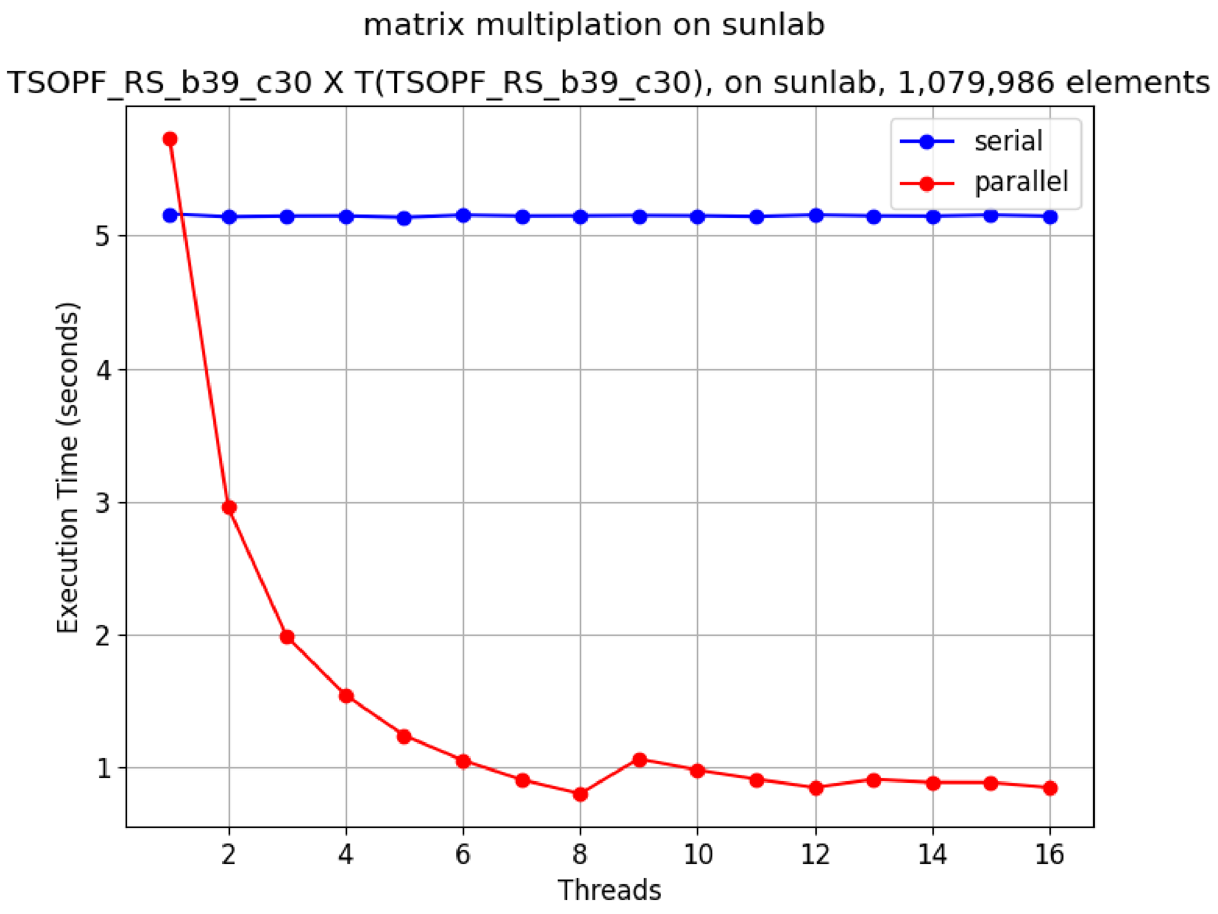


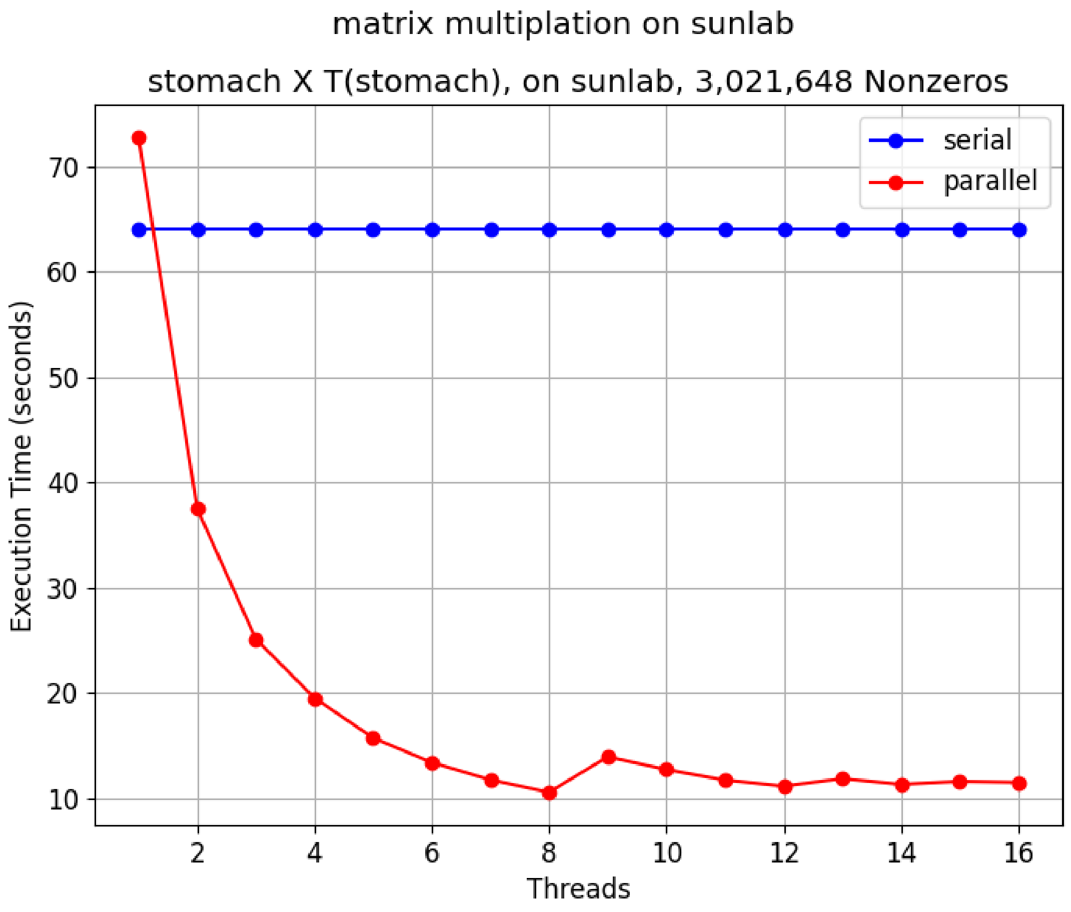
We decided that we were trying too hard to use TBB, even though TBB did not give us the right tools to perform the paralyzation that we wanted to. We then decided to use a vector of threads and lambdas to compute the multiplication.

We assigned the rows of the multiplication to each thread based on the modulo of the thread number (rows 0,2,4,6 go to thread 1, and rows 1,3,5,7 go to thread 2). We did this instead of breaking up the matrix in blocks(rows 0,1,2,3 go to thread 1, and rows 4,5,6,7 go to thread 2) in case there were multiple dense rows in a row(which does happen). The modulo version of breaking up the matrix could also fall into this issue (Ex. only even rows in the matrix have values), but we have yet to see any real-world example of this. To fix this what we could have done is assigned the rows randomly to the different threads and used probabilistic analysis, but this would require the random function to be called for each row, and this would have to be decided before the multiplication took place. However, all these solutions still have a small probability that all of the most dense rows get assigned to a single thread.

We could have also made a queue of rows, having the threads pop off a new row after each thread completes the current row they are working on. This would guarantee that each thread did equal work. We decided not to do this approach however because it is extremely unlikely that all of the dense rows get assigned to the same thread, and the queue version would require more synchronization and increase the complexity of the code.

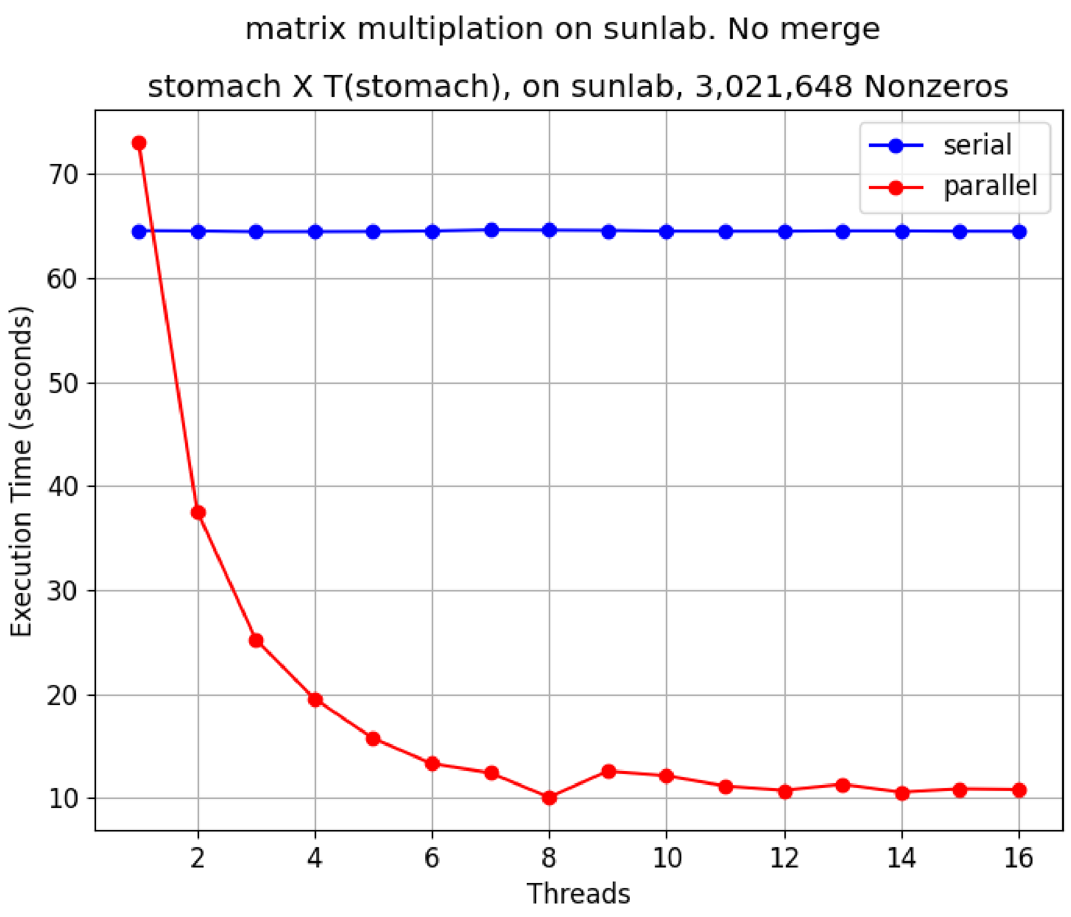
Each thread would then put the results of its computation into a vector. Once all the threads were joined, a loop would combine all the results into one resulting sparse matrix.





As shown we got a good speed-up of over 6x. However, we did not get a speed-up past 8 threads. We suspect this has to do something with hyperthreading not performing as expected on Sunlab.

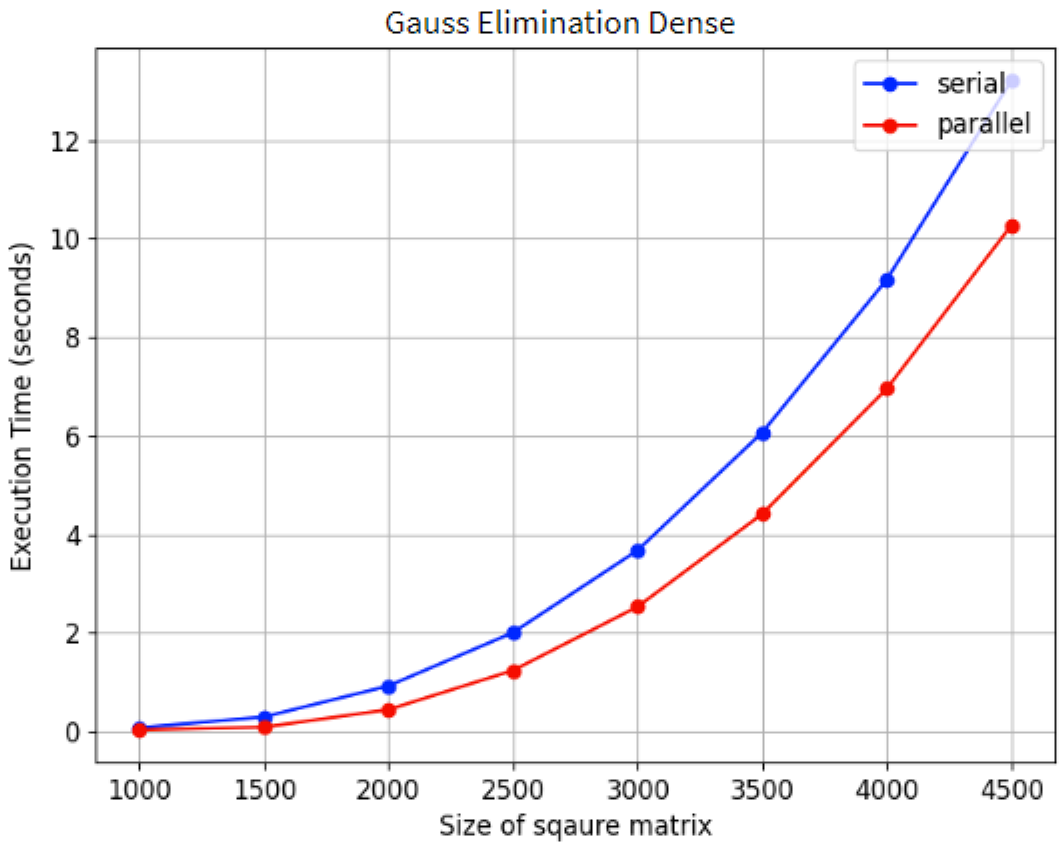
The joining of all the threads results looks expensive because of all the vector appending but in our testing, the cost was not significant. This next graph shows when the merging of each thread's result is taken out.

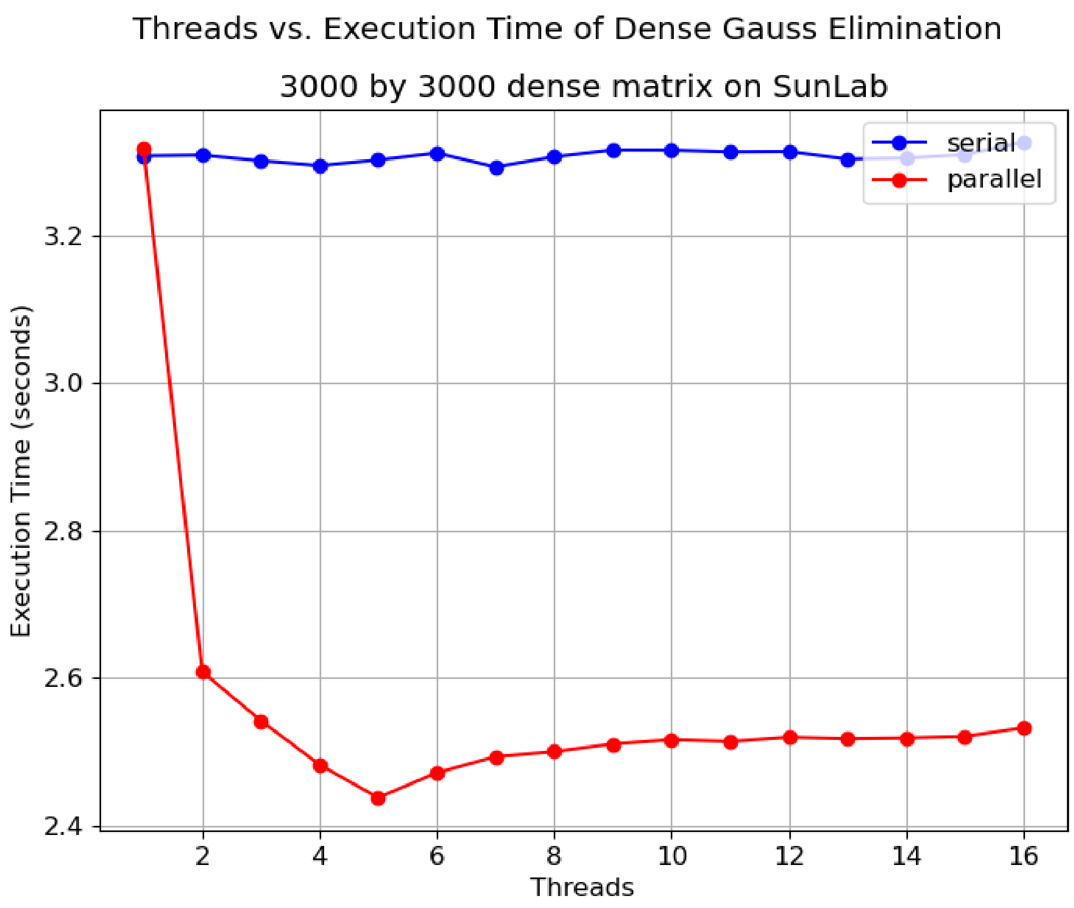


When solving linear systems, there are direct and iterative methods. Direct methods, such as Gaussian elimination or LU Factorization, try to find a solution in a finite number of steps. These steps involve However, this way of solving systems can be computationally expensive, due to many of the algorithms being quadratic in time. Iterative methods take a different approach to solving systems. Iterative algorithms take steps to try and approximate the coefficients and the accuracy is determinant on the number of iterations taken.

Next, we moved on to paralyzing some of the linear systems solvers. The first one we implemented Gauss Elimination using forward elimination, we started with the dense implementation. While Gauss Elimination is an iterative process, we were able to parallelize dividing all of the elements in the pivot row by the pivot and then parallelize subtracting the pivot row from the rest of the matrix. We got a small speed-up but suffered from high contention between threads, and having to allow too much syncretization(as each row needs to be done in order). We also tried running the first part in parallel and then switching to serial once the columns left got under a certain threshold, but we did not have any success in finding a better speed up.

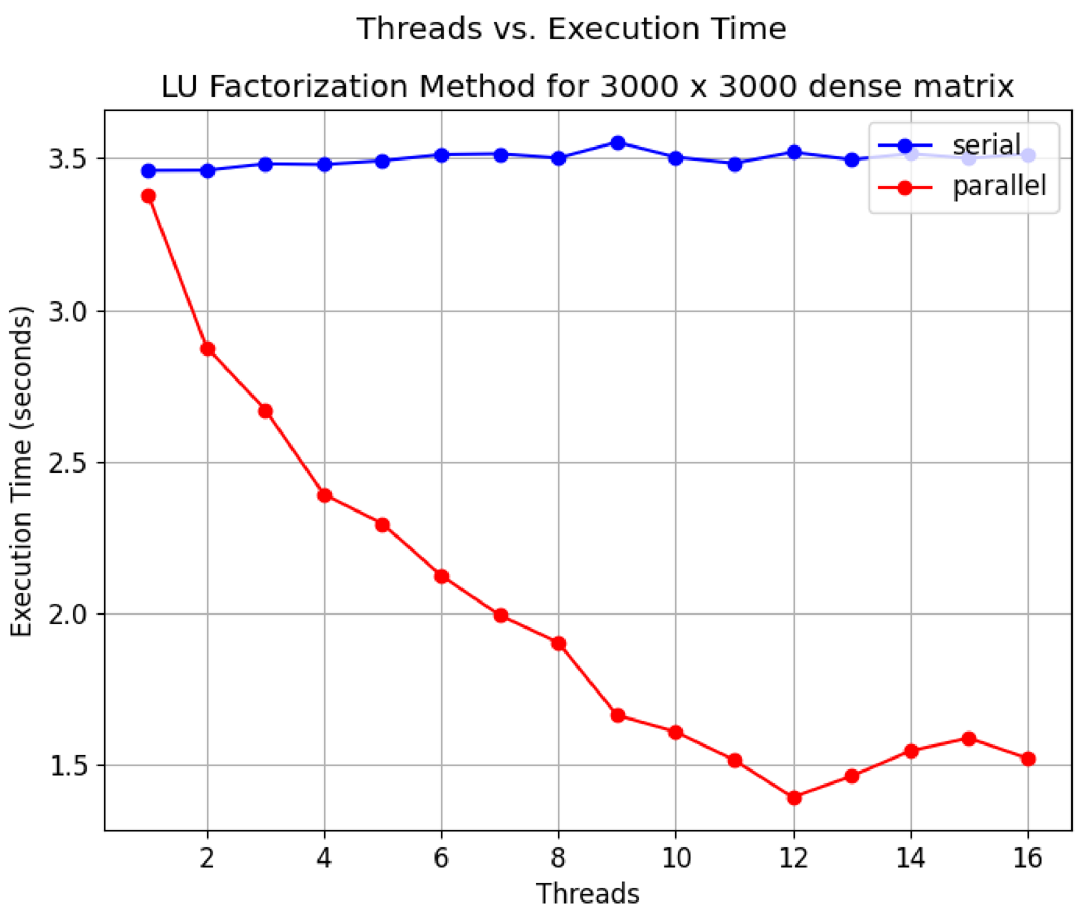
matrix size vs. Execution Time on sunlab





When looking at implementing Gauss Elimination for the sparse format, we found that it was not that useful. The whole point of Gauss Elimination is to make the lower triangle of the matrix zeros. However, that is already the case for a sparse matrix, so much of the computation is wasted. We instead decided to look at other ways to implement solver linear systems.

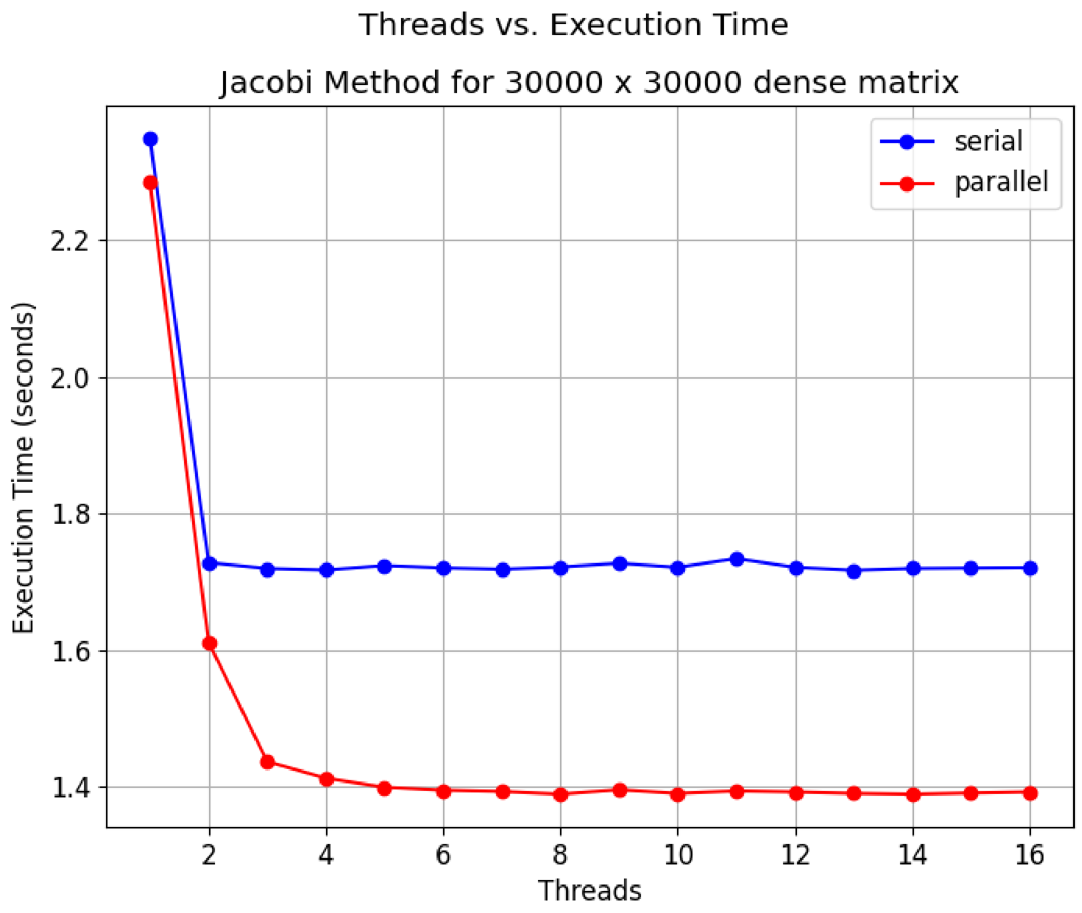
Another direct method to solve systems, and also a preconditioning technique, called LU factorization, was also implemented. In this method, given a dense matrix **A**, the algorithm will split up the matrix into an upper triangular and lower triangular matrix and return their pair. This method is very similar to Gaussian elimination as it uses forward elimination. The **L** and **U** matrices are multiplied to get **A**.



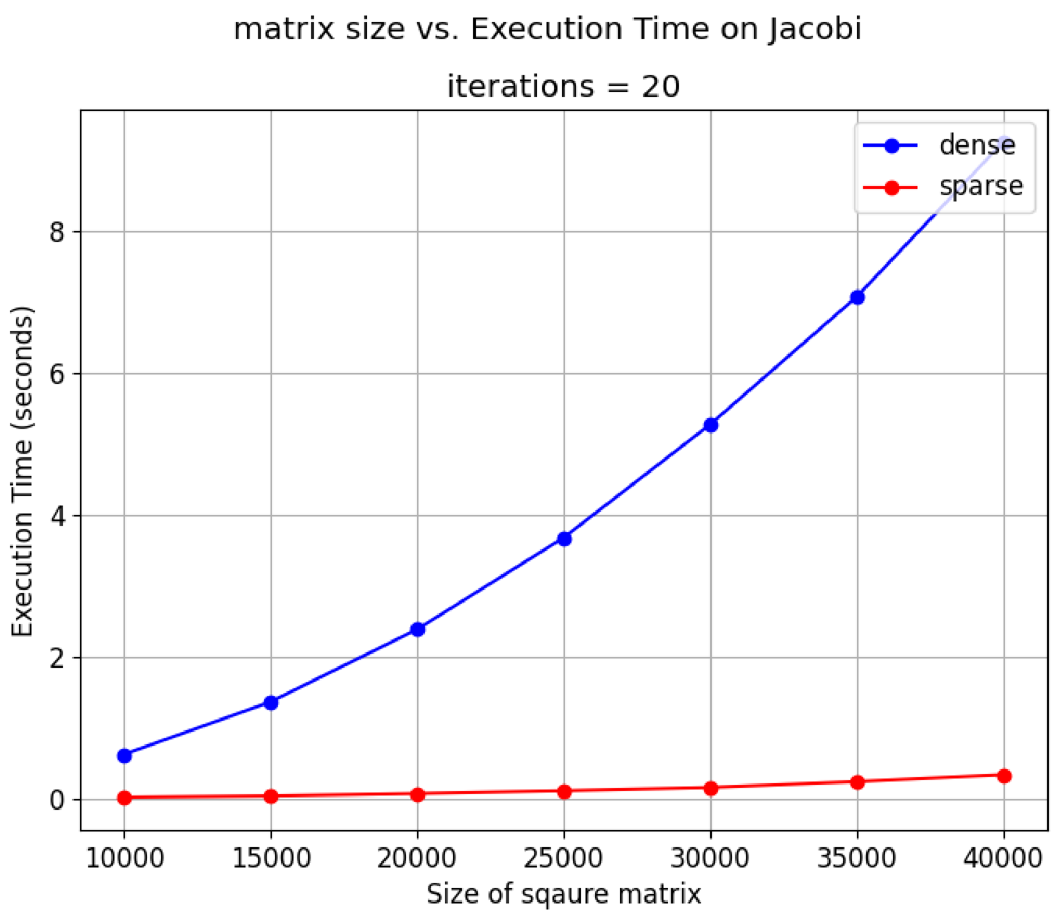
From this graph, it is apparent that parallelizing the LU Factorization provides a significant amount of speedup, especially with more threads working. The serial version, for a 2000x2000 dense matrix size, performs this operation at around 16.5 seconds. The parallel version, although starting off slower, gains a significant advantage once more threads are added. This results in a 5.97x speedup. One reason for this is because LU Factorization involves independently decomposing the matrices into two different matrices, rather than Gaussian elimination, which performs a series of row operations that may be dependent on each other, slowing down the process as a whole. However, this solution still showed high thread contention, but at a higher thread count then Gaussian elimination. Again we did not implement this method for sparse, as much of the computation is wasted.

One iterative method of solving systems is the Jacobi Method. The Jacobi Method is a way to solve a diagonally dominant system of linear equations. Through each iteration, the algorithm rearranges each row so that the *x* value at that certain row is isolated and the other *x* values are set to zero. Then, we approximate the certain value of *x* and use it in the next iteration. Through each iteration, we get closer to the actual value of *x*. Sequentially, the algorithm is trivial due to simply iterating over the dense matrix and storing the updated values of *x* in a vector.

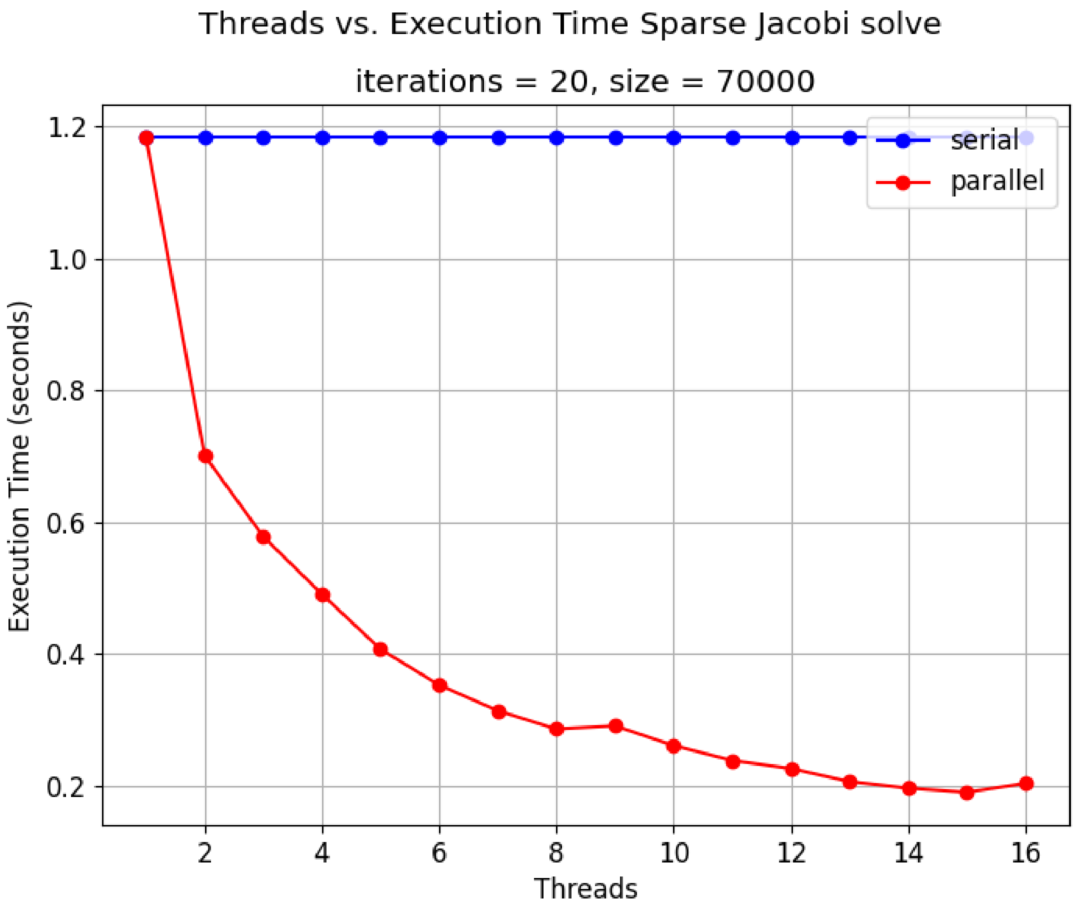
In parallel, since each row is independent of each other, TBB simply needs to map each thread to a row, perform the operations on the row, and update its x values. First, this was done on dense matrices, with the speed up being minimal. There are a few reasons as to why no substantial gains were made. Firstly, the method itself is simple and fast. It is made up of three loops, the first for counting the number of iterations, the second for iterating over the number of rows, and the last for iterating over the columns. In a 30000x30000 size dense matrix, this averages out to around 1.76 seconds. Meanwhile, the parallel version, the execution time averages around 1.46 seconds. This means in that size matrix, with 900 million elements, the serial version is fast, making the benchmark for parallelizing even harder. As for the size of the matrix, there seems to be an upper limit as if the size becomes too large, the program will run out of memory and crash.

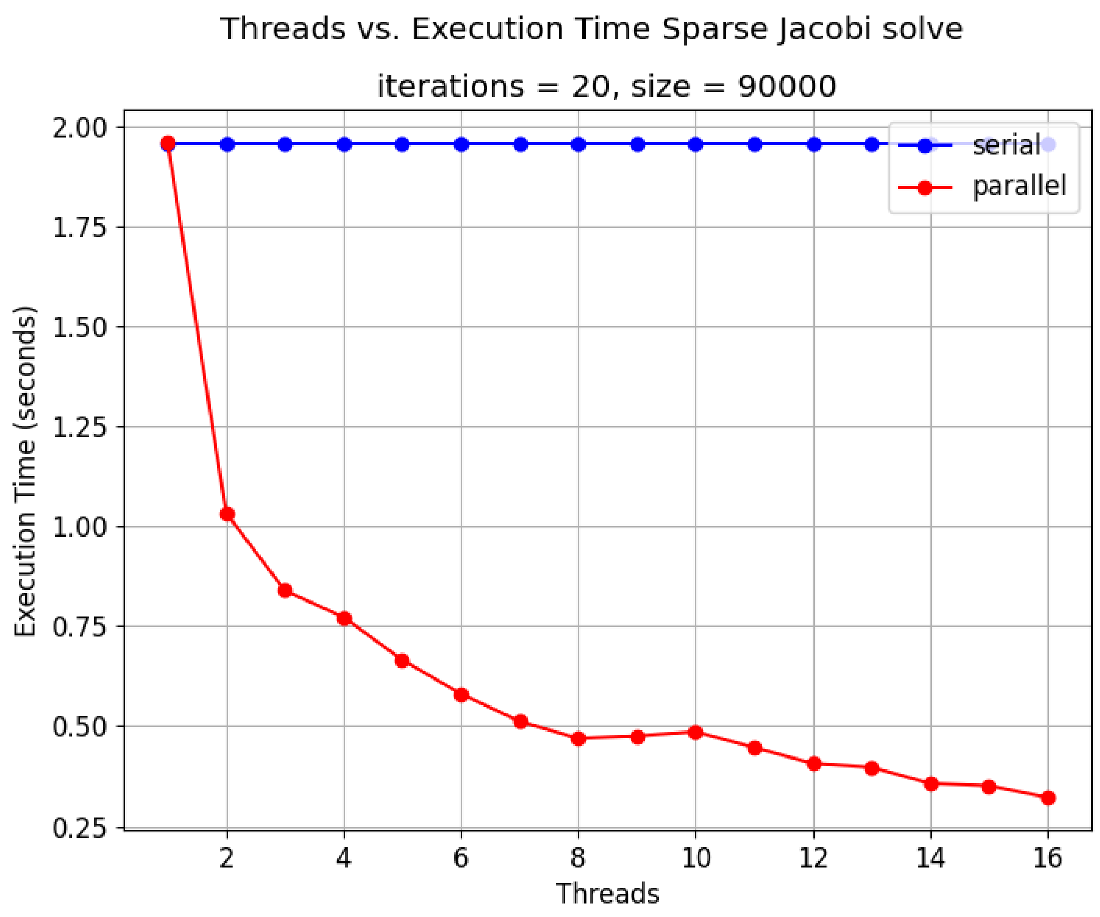


Next we implemented Jacobi in the sparse format, and saw a big speed up compared to the dense version. This is caused by the fact that diagonally dominant matrices have a high number of zeros, and thus see similar preformance gains to sparse vs. dense matrix multiplication.



We also parallezed the sparse matrix Jacobi Matrix, which was trivial as each rows calculation is independent of each other, and saw a good speed up. This method is by far the most efficient for solving a linear system that has A as a sparse matrix.


Threads vs. Execution Time Sparse Jacobi solve
iterations = 20, size = 70000
serial
parallel
Execution Time (seconds)
Threads
0.2
0.4
0.6
0.8
1.0
1.2
2
4
6
8
10
12
14
16



Overall, we found success paralyzing our matrix functions and saw good speed-ups. We are looking forward to continuing this work next semester with our capstone project. We think that the more complex functions, such as using preconditioning techniques to solve linear systems, will also have good opportunities for speed up. We hope you stop by our project at the capstone fair next semester to see more opitmzation and performance gains that we will continue to work on next semester.