University of New Mexico

Project Report

CS 442 / 542: Introduction to Parallel Processing
Professor Amanda Bienz

Chai Capili
Thomas Fisher
Praneeth Marri
Sara Romero

10 December 2023

1. Introduction:

Sparse matrices are essential elements in a number of science and technology fields generally related to computer science and engineering. To better solve these types problems which depend on computation involving sparse matrices, knowing under what conditions the performance of these structures will be optimized is a key step towards unlocking maximum efficiency, especially when viewed in the context of parallel computing, where optimized algorithms running at scale are a necessity to see expected benefits of using multiple machines.

The specific hypothesis our team decided to investigate was whether sparse matrices of dimension $n \times n$ with $O(n)$ non-zero values demonstrate performance advantages over sparse matrices of different densities for the same dimension. The idea for this area of focus was based on a claim presented in lecture which associated "useful sparsity" with this level of non-zeros elements. It should be noted that, in order to exclude entirely empty matrix rows or columns, we proceeded under the assumption that $O(n)$ non-zero values was the density floor for a given dimension n.

2. Dataset and Codebase (GitHub Repository)

2.1. Matrix Files

To measure the relationship between the three criteria mentioned above, it was important that each section used the same matrices for testing purposes. Our group created a matrix generator, which given a dimension n, partitions the density space evenly from $m^{-1}n^2$ to $(1 - m^{-1})n^2$ and generates m matrices with those densities. The spread of non-zero elements in the matrix began with populating the diagonal, and any additional elements needed to satisfy the required density were added in a uniformly random way. Most of our testing analyzed matrices of dimension 100 and 1,000, with 10 density partitions per dimension.

2.2. Sparse Matrix Structures

In order to test sparse matrix formats consistently across the portions of our team, we developed custom C structs to represent COO, CSR, and CSC formats. These structs are generated from a 2D array and each contain arrays for row data, column data, and value data, as well as the number of non-zero elements in the matrix. These structs were used in all of our testing whenever sparse formats are involved.

3. Communication Cost

The first performance metric our group analyzed was Communication Cost, evaluated through the simple yet effective method of Ping-Pong testing. In this technique, two processes send and receive back and forth messages repeatedly, allowing us to measure the round-trip time for two processes, which gives us a general idea about the communication overhead. Our group ran these tests on our 100x100 and 1000x1000 matrices, which ranged in different densities, as described in section 2.1 . For each matrix run, we had to first convert it to CSC, CSR, COO, and Dense format from our matrix (.mtx) files. By converting to each of the formats above, we are able to use Ping-Pong tests to give us some insight into the different communication performance of each format with varying densities and matrix dimensions. We employed blocking MPI_Sends and MPI_Recvs to ensure reliable data transfer. Furthermore, involving CSC, CSR, and COO, I made use of MPI_Pack and MPI_Unpack. Since these formats involve three subarrays each, it would not be accurate to have three separate sends and receives for each subarray. MPI_Pack and MPI_Unpack allow us to combine these subarrays into one contiguous block of data, which allows us to use single sends and receives with the MPI_PACKED data type.

Each Ping-Pong test included 5000 iterations, where Process 1 and 0 sent data back and forth at each iteration. To optimize our timing measurements, we utilized the MPI_Reduce function combined with the MPI_MAX operation. This strategy was instrumental in aggregating the maximum time taken across all processes for each test. Consequently, it provided a more reliable measure of the worst-case communication cost. All of these steps were taken to eliminate possible biases in our timings so we can get the most accurate picture, and investigate our hypothesis of sparsity O(n) non zero numbers having the best communication performance. We investigated this claim by running the Ping-Pong tests outlined above on Wheeler.

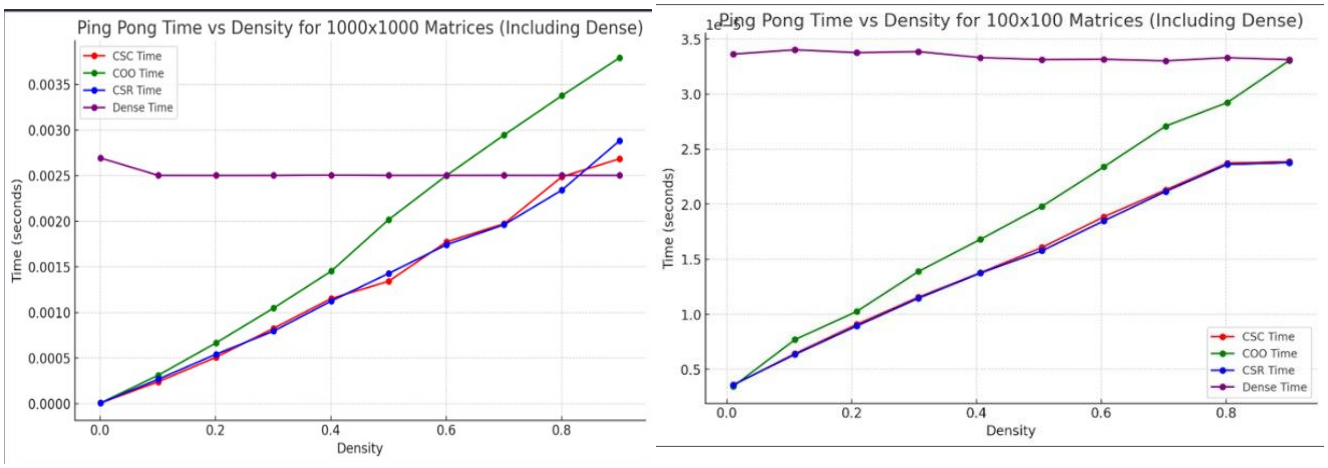

Figure 1: Time costs of sending matrices of varying formats and densities.

The bottom left corner of each graph refers to O(n) sparsity, which is consistently the best performer in both graphs. We also see in the 1000x1000 test, that the dense format begins to outperform the sparse formats as density grows. In both graphs, CSR and CSC outperform COO and Dense formats at higher sparsity and lower density. These findings support our hypothesis.

4. Storage Cost

Our team's analysis of the storage costs associated with holding sparse matrices in memory involved first developing mathematical models to produce expected byte values for a matrix of given format and density. Those predictions were then compared against test data to gain evidence in support of our hypothesis. Examining storage requirements of matrix formats also led to interesting results related to their relative optimality with regard to storage.

4.1. Predictive Modeling

Our team's modeling of the expected storage required to hold matrices of different formats was based on the way that the formats were presented in class and made the assumption of no overhead being needed in actually maintaining the structures holding the values. The table below shows the expected storage requirements expressed as mathematical expressions for different matrix formats and densities, with n denoting matrix dimension and d denoting matrix density. Total storage cost is split between terms involving integers (used for values in row and column arrays for sparse formats) and terms involving matrix elements (dependent on the type of matrix data), from which the number of total bytes can be calculated. See the dedicated Storage Analysis document in our team's repository for a more in-depth look at how these expressions were derived.

Format	Predicted Storage Requirement
Dense	n^2 matrix values
COO	$2dn^2$ integers + dn^2 matrix elements
CSR/CSC	$(n + 1) + dn^2$ integers + dn^2 matrix elements

Table 1: Predicted storage requirements across matrix formats, for dimension n and density d.

Expressed graphically, the models above produce the plots below in Figure [2] which yield the cut-off points at which we expect one matrix format to outperform the others. For example, the plots below show dense format becoming the most performant with respect to storage

requirements for matrix density of approximately $O(0.65n^2)$. This late-occurring advantage to using a dense format for larger densities is not unexpected, as the sparse formats require the overhead of 3 separate arrays while the dense format can use only 1. Finding the actual theoretical value, while not directly associated with our team's larger hypothesis, is an excellent example of the kind of ancillary sparse matrix information found in the course of our project.

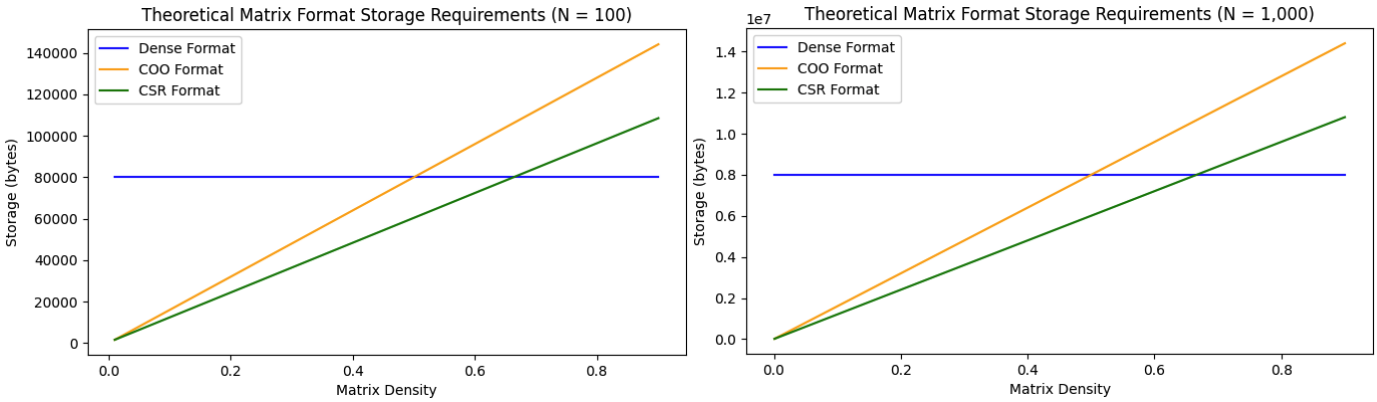

Figure 2: Plots of predicted matrix storage requirements over different densities and formats for dimensions 100 and 1,000. Matrix values are of type double.

4.2. Experimental Analysis

The predicted storage requirements above were tested against the actual storage cost of different sparse matrices using Valgrind's massif heap profiler tool [2] on CARC's Wheeler cluster [3]. To do this, a simple program was used which created one of dense, COO, or CSR matrix formats from a matrix file, and then freed the struct. This process was repeated for all formats for matrix files of varying densities. The peak heap usage over the run of the program was then used to represent the matrix's required storage. Note that using a heap profiler is appropriate in this analysis as all matrix formats are allocated on the heap, not the stack.

The figure below shows the observed heap usage for the various matrix parameters. The trends between the formats align closely with those predicted by the storage models, with the slopes of the sparse format curves perhaps slightly steeper in the experimental case. These results speak to the correctness of the predictive models and support our hypothesis as the best storage performance for a matrix of given dimension is seen at minimal density.

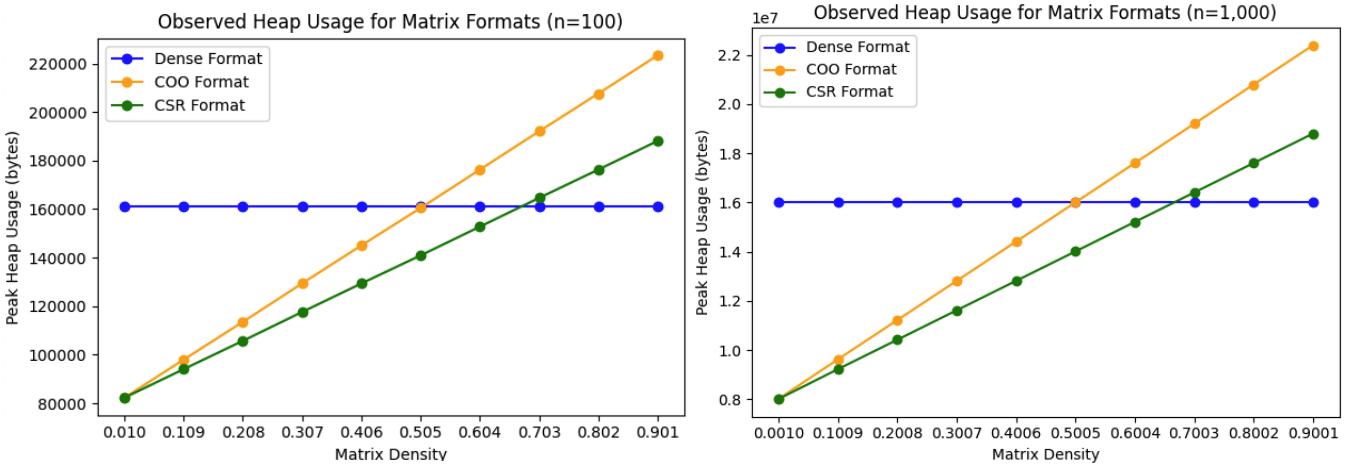

Figure 3: Plot of analyzed matrix storage requirements over different densities and formats for dimensions 100 and 1,000. Matrix is of type double.

5. Computation Cost

5.1. Matrix Transpose

To measure optimal density in terms of operational runtime, our group decided to implement a matrix transpose algorithm. For CSR transpose, we discovered the algorithm is equivalent to performing a CSR to CSC format conversion. Our CSR algorithm is based on the csr_tocsc method for Python by SciPy [1], adjusted for C programming, implementation of our database, and use of MPI. The algorithm consists of counting the number of nonzeros in each column, cumulatively summing the column counts to get the new row pointers, and using those counts and the values of the original row pointers to reassign the indices of the columns and values. In contrast, a dense algorithm consists of partitioning the matrix into blocks, transposing each block, and then inserting the transposed block into the final matrix. The results of the CSR algorithm against the dense algorithm can be seen in Algorithm [1].

Algorithm 1 CSR Sparse Matrix Transpose

```
1: procedure CSR TRANSPOE
2:     Get matrix and convert to CSR
3:     for m = 0 to 100 do
4:         Partition column array among procs
5:         for i = start_col to end_col do
6:             Count non-zeros in each column
7:             t_rowPtr_local[col[i]] ++
8:         end for
9:         MPI_Reduce(t_rowPtr ← t_rowPtrLocal)
10:        if rank = 0 then
11:            for i = 0 to n do
12:                Cumulatively sum column counts
13:                temp ← t_rowPtr[i]
14:                t_rowPtr[i] ← sum
15:                sum += temp
16:            end for
17:            for i = 0 to n do
18:                Assign new column and value indices
19:                for j = rowPtr[i] to rowPtr[i+1] do
20:                    index ← t_rowPtr[col[j]]
21:                    t_col[index] ← i
22:                    t_val[index] ← val[j]
23:                    t_rowPtr[col[j]] ++
24:                end for
25:            end for
26:            for i = 0 to n do
27:                Shift row pointer
28:                temp ← t_rowPtr[i]
29:                t_rowPtr[i] ← prev
30:                prev ← temp
31:            end for
32:        end if
33:    end for
34: end procedure
```

Algorithm 1: CSR transpose algorithm with parallelism.

It is evident that the CSR transpose experiences better operation runtime across all densities which does not demonstrate the expected trade-off in runtime at O(n) non-zero values. Due to the vast difference in the two algorithms, it is hard to compare the results effectively. A detailed look into the CSR transpose algorithm with parallelism can be seen in Figure [4].

Figure 4: Plot of CSR transpose algorithm compared with dense, averaging results of 100 iterations on 1 process.

Due to the serial nature of row pointers being cumulative, we experienced major difficulties in parallelizing the algorithm, and only the initial count of the non-zeros in each column was split among the processes.

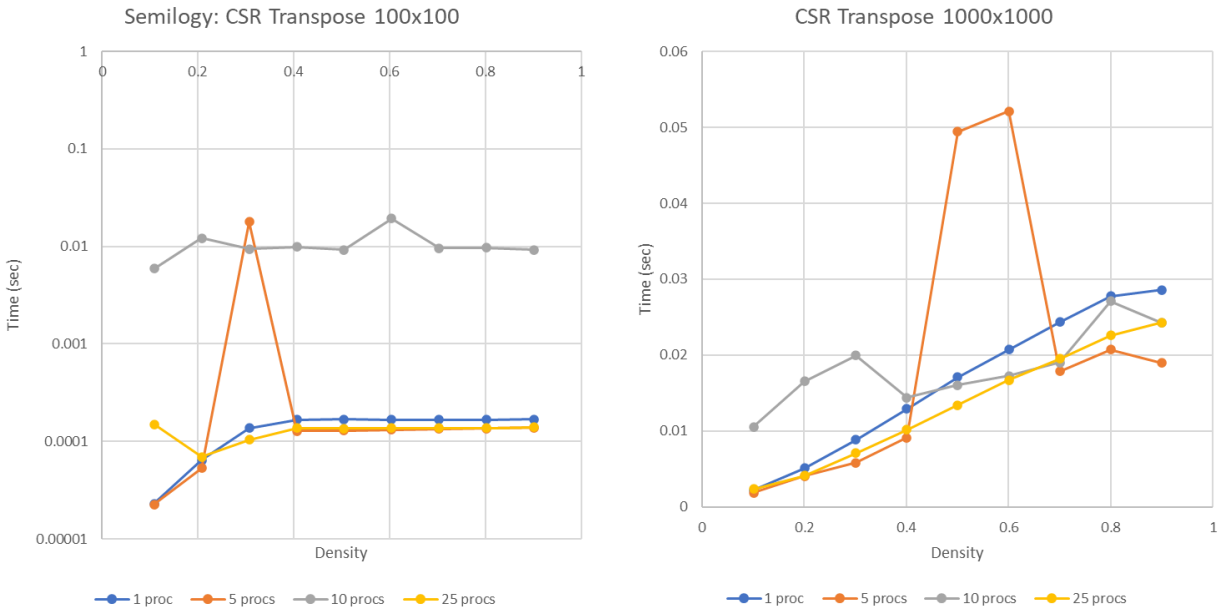

Figure 5: Plot of CSR transpose algorithm, averaging results of 100 iterations on multiple processes.

The results, seen in Figure [5] demonstrate varying communication overhead with 25 processes only minimally outperforming 1 process. We can observe the balance of processors and communication overhead is poor for 5 and 10 processes. Our team also analyzed COO transpose

which consists of swapping the row and column arrays. This could be done with the initial conversion of the matrix to COO, assigning the row values to a column array and the column values to a row array. However, to better analyze the performance, the matrix was read in and converted as normal and then manually swapped the arrays as seen in Algorithm [2].

Algorithm 1 COO Sparse Matrix Transpose

```
procedure COO TRANSPOE
    Get matrix and convert to COO
    for m = 0 to 100 do
        temp ← row
        row ← col
        col ← temp
    end for
end procedure
```

Algorithm 2: COO transpose algorithm without parallelism.

This algorithm was then parallelized with a series of MPI_Scatter and MPI_Gather statements which proved to have major communication overhead with preference to only one process (Figure [6]).

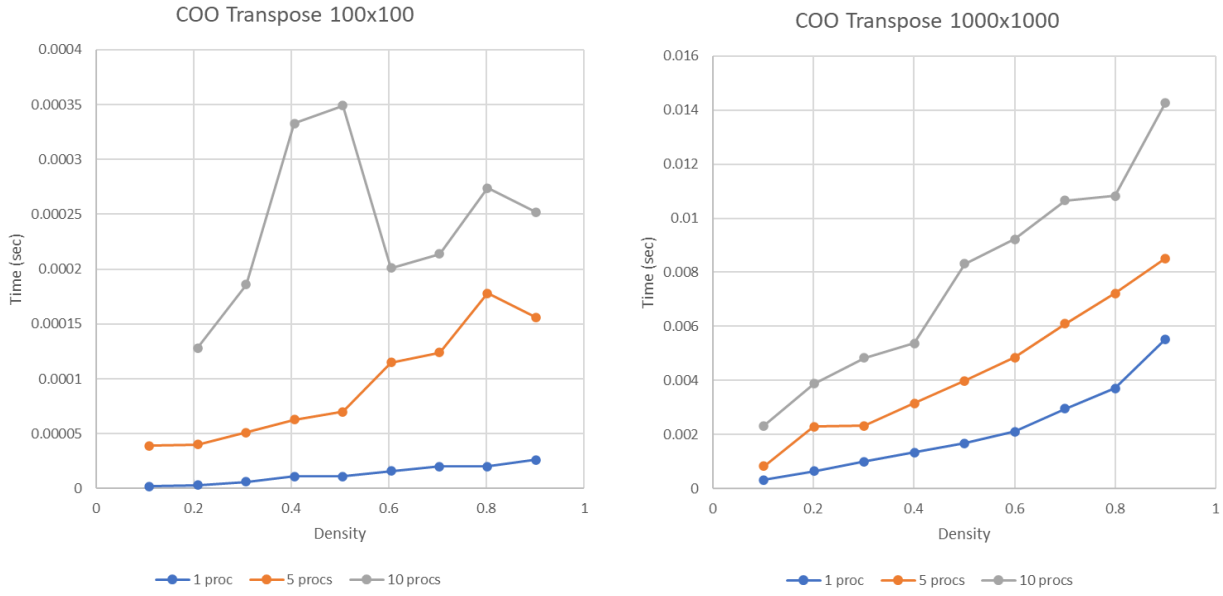

Figure 6: Plot of COO transpose algorithm, averaging results of 100 iterations on multiple processes.

When comparing all three algorithms, averaging 100 iterations of each method using one process, we can observe in Figure [7] that COO unexpectedly outperforms CSR in operational runtime due to the increased complexity of the CSR transpose algorithm.

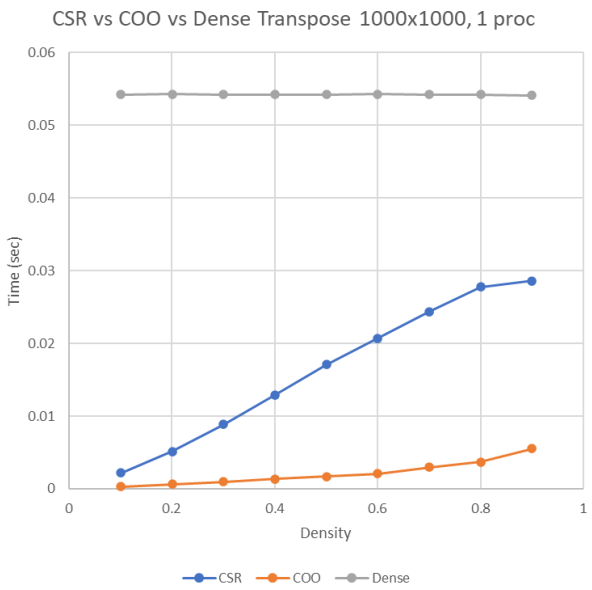

Figure 7: Plot of COO transpose algorithm compared with CSR and dense, averaging results of 100 iterations on 1 process

5.2. Matrix Matrix Multiplication

Our group also examined how different densities and matrix formats affect the runtime of sparse matrix-matrix multiplication (SpGEMM). For our analysis, matrices with dimensions 100 by 100 were chosen with densities varying from 0.01 to 0.9 and the performance metric chosen was operation runtime of SpGEMM. Each matrix is multiplied by itself and performance was evaluated across matrices of various densities.

Our implementation of parallelized matrix multiplication algorithm uses COO format and MPI. We used a transpose function to transpose matrix B for performance improvement. The CooMat function distributes the workload among different processes where each process multiplies a subset of rows from matrix A by all the columns from the transposed matrix B. The main function begins by initializing MPI, broadcasting the matrix dimension and reading input matrices from files, converting them to COO format and measuring the time for multiple iterations of the COO matrix multiplication. The resulting time is then collectively reduced across all processes using MPI reduce and the maximum time is printed providing insights into the performance of the parallelized matrix multiplication algorithm across multiple processes. Runtimes were measured on a local workstation using four processes throughout the experiments. The observed finding was that the minimum runtime occurred when the matrix had O(n) non-zero elements, aligning with our initial hypothesis.

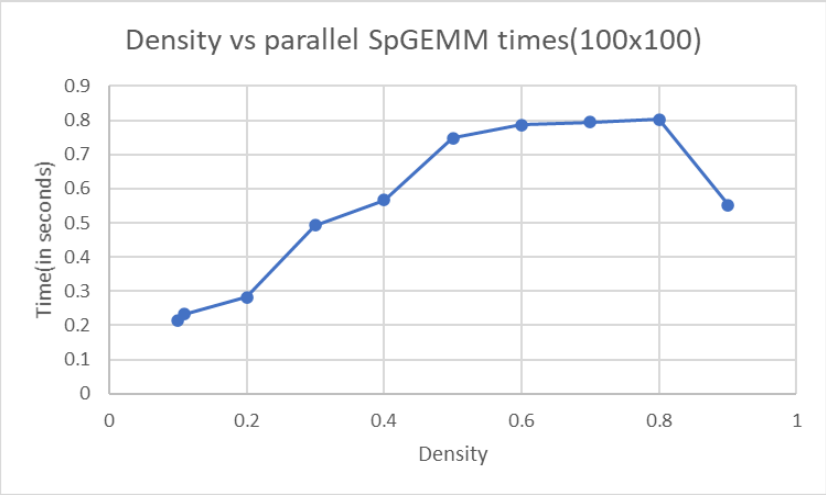

*Figure 8: Parallel SpGEMM Runtimes of COO for 100*100 matrices (4 processes)*

In addition to this a serial algorithm was also implemented using CSR and COO formats for the same matrices with densities varying from 0.01 to 0.9. For each format matrix multiplication was done 100 times and the time for each iteration was measured using CPU clock. Comparing the average time of CSR and COO over 100 iterations gave us two important findings: 1) CSR outperformed COO format for both matrix dimensions and 2) Both COO and CSR formats, irrespective of the dimensions of the matrix, had least runtime when the number of non-zeros in matrix is O(n) which supports the hypothesis.

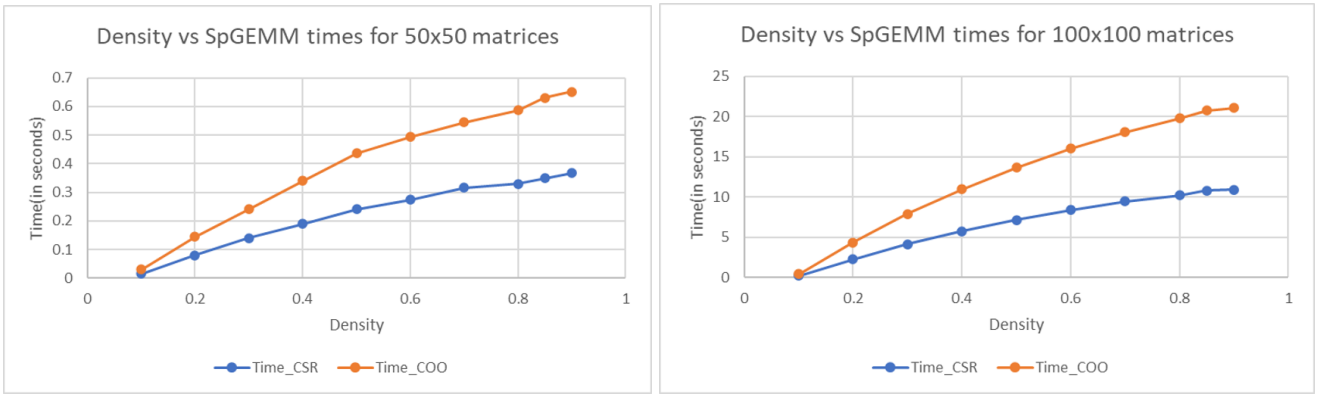

Figure 9: Average serial SpGEMM Runtimes of COO vs CSR (100 iterations)

6. Conclusion and Future Work

Our team employed a fourfold approach to exploring the question of whether $O(n)$ density in a sparse matrix led to measurable performance benefits as compared to higher densities. Using a variety of different sparse matrix formats in algorithms of our own design, we examined communication cost, computation cost, and storage cost to gain a comprehensive picture of sparse matrix performance. Taken as a whole, our data does support $O(n)$ density as being the

ideal amount of non-zeros to have in a sparse matrix for best performance in the three metrics mentioned above.

As our project drew to a close, our team identified areas where, given more time, we would be able to learn even more about the factors associated with sparse matrix performance. Firstly, we were interested in investigating sparsity patterns beyond a uniform random distribution of non-zeros. Analyses similar to the ones presented above could be repeated comparing different sparsity patterns to identify whether some or all evaluation metrics we examined would perform better or worse on one. Secondly, the idea of introducing GPU analysis . And lastly, since optimization and maximum parallelism was less of a primary objective in our algorithms than comprehensibility and reproducibility, we would be interested to discover whether any of our results would differ significantly if rerun in a more efficient manner.

References

[1] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E.A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. (2020) **SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python**. *Nature Methods*, 17(3), 261-272.

[2] Nicholas Nethercote, Julian Seward. **How to Shadow Every Byte of Memory Used by a Program**. *Proceedings of the Third International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE 2007)*, San Diego, California, USA, June 2007.

[3] We would like to thank the UNM Center for Advanced Research Computing, supported in part by the National Science Foundation, for providing the high performance computing resources used in this work.