**SEMESTER PDC PROJECT REPORT**

**“SPARSE MATRIX-VECTOR MULTIPLICATION FOR LARGE DATASETS”**

Shayan 22k4148

Rania Ghazanfar 22k4159

Marium Arif 22k4320

Advisor: Muhammad Kashif

Submission Date: 26-11-2024

# Overview

The project at hand focuses on the parallelization of matrix-vector multiplication using OpenMP and MPI. The objective of the project was to analyze and compare the performance (speedup and efficiency) of these techniques for different data sizes and sparsity levels. We used three large data sets 1400000, 1600000, and 1800000 for testing parallelization. The main problem was storing large data sets. The solution we came up with was to compute a fixed chunk size over and over again iteratively to reach the original size. We partitioned the large datasets into fixed chunk sizes and performed the multiplication and various metrics.

# Project Detail

The project consists of three codes:

1: SERIAL EXECUTION:

The serial code implements sparse matrix-vector multiplication, where a sparse matrix is generated based on a given sparsity percentage, and the multiplication is performed with an input vector to produce an output vector. The program also calculates various metrics, such as the maximum, minimum, and sum of the result vector, as well as the row and column averages of the matrix. It initializes the matrix and vectors with random values, performs the multiplication in a single loop, and measures execution time. While the code efficiently handles sparse matrices, its performance is constrained by the lack of parallelism, making it less scalable for larger matrix sizes. Hence the execution increases in increasing datasets for any level of sparsity.

2: OPENMP EXECUTION:

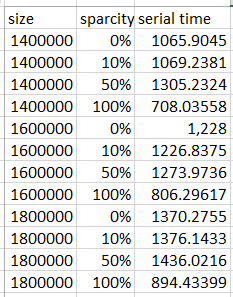
The parallelized code uses OpenMP to implement sparse matrix-vector multiplication and various matrix operations, leveraging parallelism to improve performance. It initializes the sparse matrix according to sparsity percentage, input, and output vectors, and then computes the multiplication using multiple threads. The program also computes several matrix statistics, such as the maximum, minimum, and sum of the result vector, as well as row and column averages. Key operations, including matrix generation, vector initialization, and calculations, are parallelized using OpenMP constructs such as #pragma omp for and #pragma omp sections. The program dynamically schedules the sparse matrix-vector multiplication and uses reductions for summing operations, ensuring that the parallel execution results are combined correctly. Additionally, the user can specify the number of threads for parallel execution, making the program scalable for larger matrix sizes. The code shows how parallelism can significantly speed up matrix operations compared to a serial implementation.

3:MPI EXECUTION:

We used the code to perform sparse matrix-vector multiplication and matrix statistics using MPI (Message Passing Interface) to distribute the computation across multiple processes. The matrix is randomly populated with sparse values, and the input vector is filled with random values. The matrix-vector multiplication is performed on each process, where each process handles a portion of the rows of the matrix. Only non-zero elements are used in the multiplication to optimize performance. After each multiplication, the program computes statistics like the maximum, minimum, sum of the result vector, and the sum of matrix rows and columns. These statistics are computed locally by each process and then reduced to global values using MPI's MPI\_Reduce. The program calculates row and column sums, which are then reduced across all processes. It also calculates row and column averages based on these sums. The matrix is split into smaller submatrices, each assigned to a different MPI process. Communication between processes is achieved through MPI broadcasting and reductions to combine results.

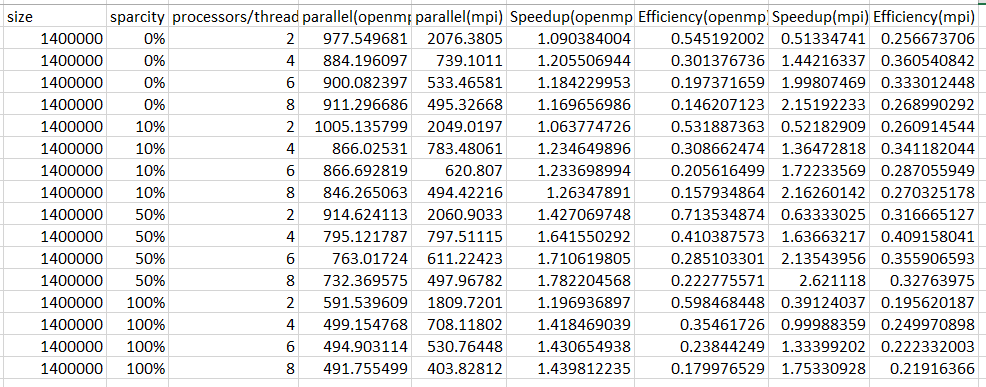
# Project Testing and Numerical Results:

1) SERIAL TIMES:

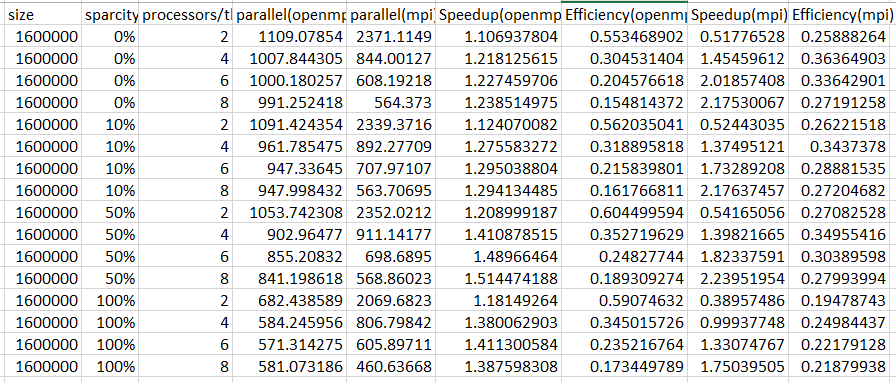


2) SPEEDUP AND EFFICIENCY OF OPENMP AND MPI OF LARGE DATASETS:

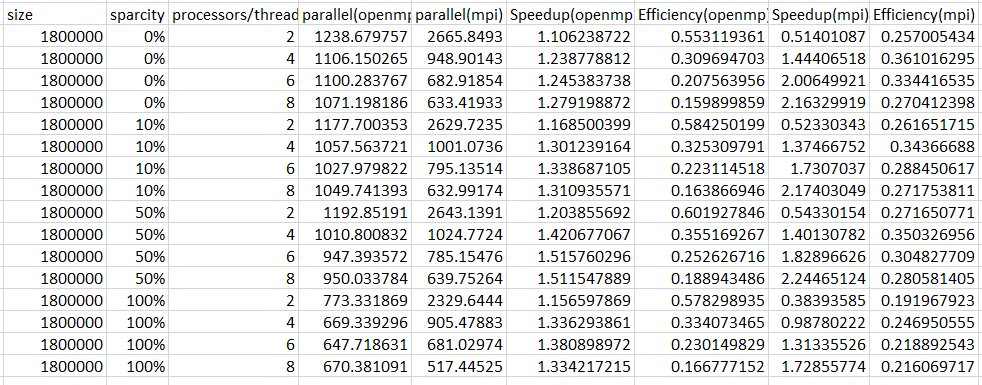
**SIZE = 1400000:**

****

**SIZE = 1600000:**

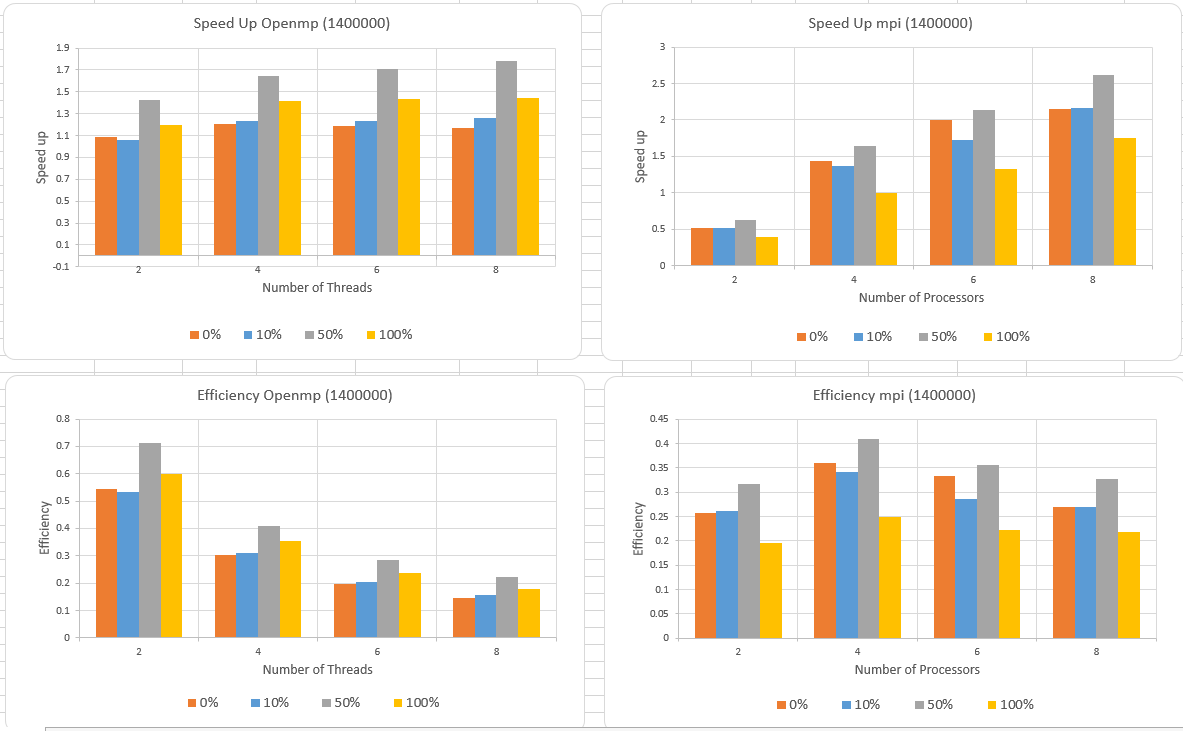
****

**SIZE = 1800000:**

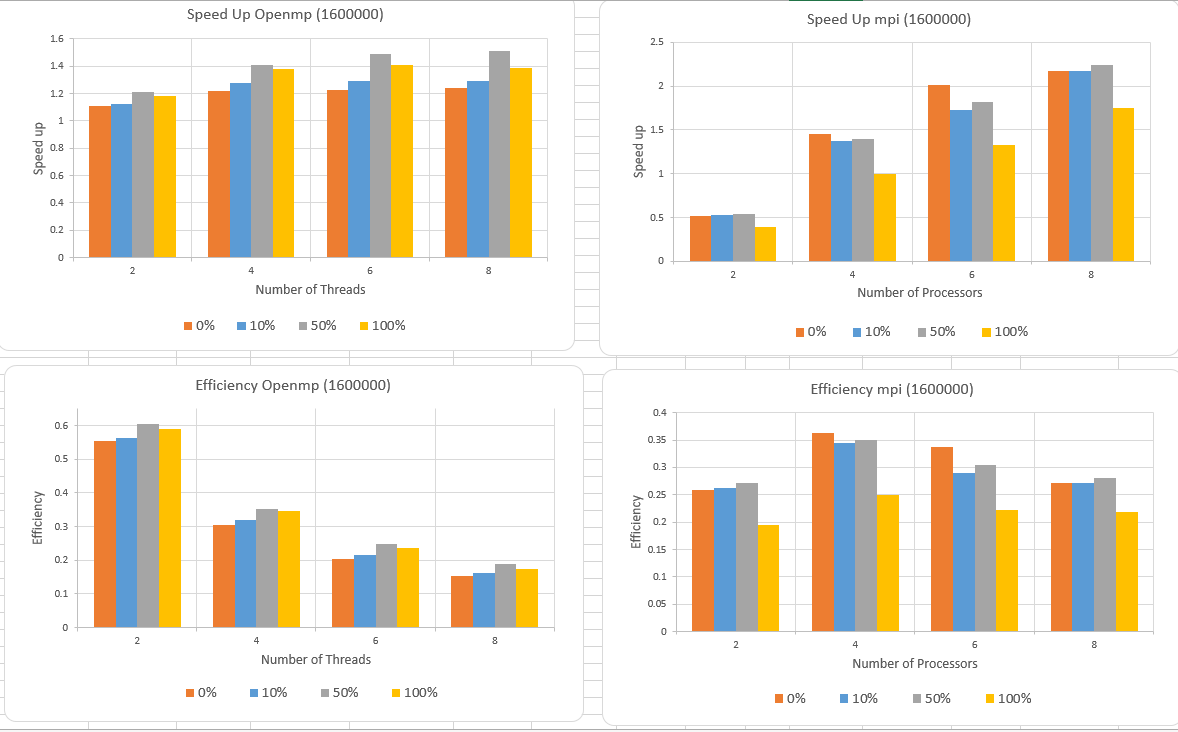


# Visual Representation:

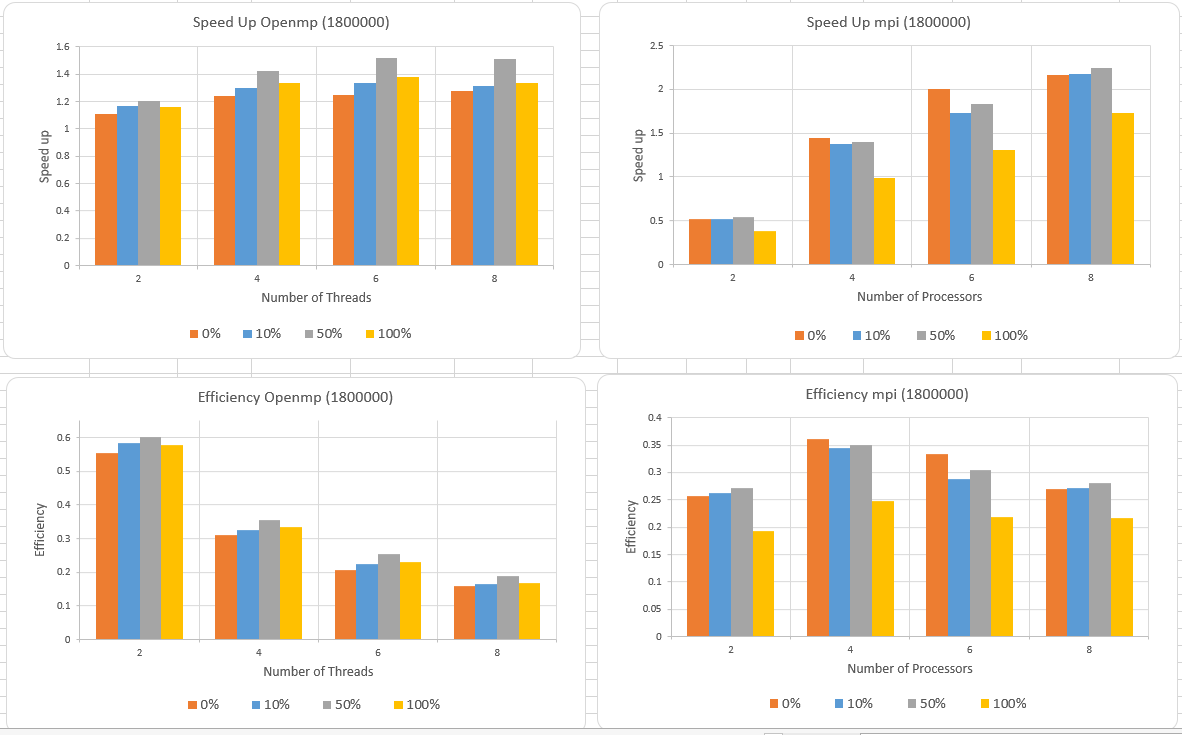
SIZE = 1400000:



SIZE = 1600000:



SIZE = 1800000:



# Analysis of Performance Data:

##### 1) Matrix Size: 1400000:

* **OpenMP Performance**:  
  For the 0% sparsity case, OpenMP showed a steady increase in speedup with the number of processors/threads, but the efficiency decreased as the number of threads increased. This indicates that while parallelism improved the speedup, it did not fully utilize the added resources effectively. At 2 threads, the efficiency was 0.545, which decreased to 0.146 at 8 threads.

At 10% sparsity, the performance was slightly better, with speedup increasing to 1.23 at 4 threads. However, efficiency continued to decline, reaching 0.15 at 8 threads.

As the sparsity increased to 50% and 100%, the performance metrics for OpenMP were less favorable. Speedup and efficiency continued to degrade as the data size and sparsity increased. This suggests that OpenMP struggles with scalability, particularly with larger and sparser data sets.

* **MPI Performance**:  
  MPI showed similar trends but with better scalability compared to OpenMP. At 2 processors, the speedup was 0.51, and at 8 processors, it increased to 2.15, showing a greater ability to scale with additional processors. MPI’s efficiency, however, dropped significantly at higher processor counts, reaching as low as 0.219 at 8 processors.

At 50% sparsity, MPI’s speedup and efficiency were higher than OpenMP's, particularly at 8 processors, where the speedup reached 2.62. For 100% sparsity, the performance dropped for both OpenMP and MPI, with MPI showing slightly better efficiency than OpenMP. However, the overhead of message passing became more evident at high processor counts.

##### 2) Matrix Size: 1600000

* **OpenMP Performance**:  
  OpenMP continued to show a similar pattern, with a decreasing efficiency as the number of threads increased. For 0% sparsity, the best performance was at 2 threads, with a speedup of 1.12 and an efficiency of 0.56. At 4 threads, the performance dropped significantly. This trend continued for 10% and 50% sparsity cases, where speedup reached a peak at 6 threads but efficiency continued to fall.
* **MPI Performance**:  
  MPI’s performance was higher than OpenMP’s for most cases, with a maximum speedup of 2.24 at 8 processors for 50% sparsity. Efficiency also improved with MPI compared to OpenMP at higher processor counts, although communication overhead was more pronounced at larger matrix sizes especially at two processors.

##### 3) Matrix Size: 1800000

* **OpenMP Performance**:  
  For the largest data size, OpenMP showed a similar pattern of diminishing returns with increasing threads. For 0% sparsity, the highest speedup was 1.28 at 8 threads, but efficiency dropped drastically to 0.16. The best performance came at 2 threads for 50% sparsity with speedup reaching 1.52 and efficiency around 0.25 at 6 threads.

OpenMP's performance became quite limited as sparsity increased to 100%, with very low speedup and efficiency.

* **MPI Performance**:  
  MPI showed better scalability for larger matrices, with 50% sparsity yielding the highest performance of 2.24 at 8 processors. The performance degradation at 100% sparsity was less severe for MPI compared to OpenMP. The 2 processors case showed a steady speedup of 1.16, indicating better performance retention across the sparsity levels.

# Challenges Encountered:

1) SCALABILITY:  
Both OpenMP and MPI showed difficulty in scaling efficiently with increased processor counts, especially for high sparsity levels. OpenMP's efficiency decreased rapidly with the number of threads, and MPI experienced communication overhead that reduced its efficiency at higher processor counts.

2) MEMORY MANAGEMENT:  
The complexity of managing memory and distributing tasks effectively across processors was a challenge. OpenMP, being a shared-memory model, struggled with thread contention as the number of threads increased, while MPI's distributed-memory approach encountered latency and overhead due to message passing. Large data sets storage was also a problem.

3) LOAD BALANCING:  
Ensuring that work was evenly distributed across processors was another key challenge. In both OpenMP and MPI, the performance suffered when there was uneven workload distribution, especially at larger matrix sizes.

# Conclusion:

OpenMP is best suited for smaller data sizes and lower sparsity levels (such as 0% to 10%). Its ease of implementation and lower overhead make it a good choice for shared-memory architectures with a moderate number of processors. However, it struggles with scalability for large data sizes and high sparsity.

MPI outperforms OpenMP for larger matrix sizes (e.g., 1600000 and 1800000) and higher processor counts. It doesn’t perform well at only 2 processors but gradually increases performance with more processors for large data sets keeping in mind it doesn’t involve a lot of communication overhead between processors. It scales better for large, distributed-memory systems but incurs significant communication overhead, particularly when message passing is required for larger matrices. MPI is recommended for high-performance clusters where scalability is crucial.

For smaller matrices and lower sparsity, OpenMP provides good performance with lower overhead, making it suitable for moderate workloads. For large, distributed data, especially with high sparsity, MPI proves more effective due to its ability to handle communication and distribution across multiple processors.