



Alexis Balayre

High Performance Technical Computing Assignment

School of Aerospace, Transport and Manufacturing
Computational Software of Techniques Engineering

MSc

Academic Year: 2023 - 2024

Supervisor: Dr Irene Moulitsas

5th February 2024

Abstract

Replace with your abstract text of not more than 300 words.

Table of Contents

Abstract	ii
Table of Contents	iii
List of Figures	vi
List of Tables	viii
1 Introduction	1
2 Methodology	2
2.1 Data Structures	2
2.1.1 Sparse Matrix	2
2.1.2 Fat Vector	3
2.2 Sequential Algorithm	3
2.2.1 Algorithm Flow	3
2.2.2 Temporal Complexity Analysis	3
2.3 Line-Based Parallelism	4
2.3.1 Algorithm Flow	4
2.3.2 Time Complexity Analysis	5
2.3.2.1 Computational Complexity	5
2.3.2.2 Communication Complexity	5
2.3.2.3 Final Result Reconstruction	6
2.3.2.4 Overall Time Complexity	6
2.3.3 Performance Analysis	6
2.3.3.1 Performance Dependence	6
2.3.3.2 Expected Performance	6
2.4 Column-Wise Parallelism	7
2.4.1 Algorithm Flow	7
2.4.2 Temporal Complexity Analysis	8
2.4.2.1 Computation Time Complexity	8
2.4.2.2 Communication Time Complexity	8
2.4.2.3 Final Result Reconstruction	8
2.4.2.4 Overall Time Complexity	8
2.4.3 Performance Analysis	9
2.4.3.1 Performance Dependence	9
2.4.3.2 Expected Performance	9

2.5	Non-Zero Element Parallelism	10
2.5.1	Algorithm Flow	10
2.5.2	Temporal Complexity Analysis	11
2.5.3	Computation Complexity	11
2.5.4	Communication Complexity	11
2.5.4.1	Final Result Reconstruction	11
2.5.5	Overall Time Complexity	11
2.5.6	Performance Analysis	12
2.5.6.1	Performance Dependence	12
2.5.6.2	Expected Performance	12
2.6	Performance Metrics	13
3	Results and Discussion	14
3.1	Results	14
3.1.1	Sparse Matrix Impact	14
3.1.1.1	Execution Time	14
3.1.1.2	Communication Time	16
3.1.1.3	Computation Time	18
3.1.1.4	Performance	19
3.1.2	Fat Vector Impact	21
3.1.2.1	Execution Time	21
3.1.2.2	Communication Time	23
3.1.2.3	Computation Time	24
3.1.2.4	Performance	26
3.2	HPC Environmental Impact	27
4	Conclusion	29
	References	30
A	Documentation	31
A.A	Project tree	31
A.B	Getting Started	31
A.C	Methods Overview	32
A.C.1	Utils.h	32
A.C.1.1	ConvertPETScMatToFatVector	32
A.C.1.2	areMatricesEqual	32
A.C.1.3	readMatrixMarketFile	32
A.C.1.4	generateLargeFatVector	33
A.C.1.5	serialize and deserialize	33
A.C.2	SparseMatrixFatVectorMultiply.h	33
A.C.2.1	sparseMatrixFatVectorMultiply	33
A.C.3	SparseMatrixFatVectorMultiplyRowWise.h	34
A.C.3.1	sparseMatrixFatVectorMultiplyRowWise	34
A.C.4	SparseMatrixFatVectorMultiplyColumnWise.h	34
A.C.4.1	sparseMatrixFatVectorMultiplyColumnWise	34
A.C.5	SparseMatrixFatVectorMultiplyNonZeroElement.h	34

A.C.5.1	sparseMatrixFatVectorMultiplyNonZeroElement	34
B	Source Codes	35
B.A	Data Structures	35
B.B	Sequential Algorithm	36
B.B.1	Declaration File	36
B.B.2	Implementation File	36
B.C	Line-Based Parallelism	37
B.C.1	Declaration File	37
B.C.2	Implementation File	37
B.D	Column-Wise Parallelism	40
B.D.1	Declaration File	40
B.D.2	Implementation File	40
B.E	Non-Zero Element Parallelism	43
B.E.1	Declaration File	43
B.E.2	Implementation File	43
B.F	Utility Functions	46
B.F.1	Declaration File	46
B.F.2	Implementation File	47
B.G	Main File	51
B.H	Scripts	58
B.H.1	MPI Submission Script	58
B.H.2	Batch Test Script	60
B.H.3	Get CSV Script	62

List of Figures

3.1	Cage4 matrix execution time	14
3.2	DC1 matrix execution time	15
3.3	Amazon0302 matrix execution time	15
3.4	Cop20k_A matrix execution time	15
3.5	FEM_3D_thermal1 matrix execution time	16
3.6	Cage4 matrix communication time	16
3.7	DC1 matrix communication time	16
3.8	Amazon0302 matrix communication time	17
3.9	Cop20k_A matrix communication time	17
3.10	FEM_3D_thermal1 matrix communication time	17
3.11	Cage4 matrix computation time	18
3.12	DC1 matrix computation time	18
3.13	Amazon0302 matrix computation time	18
3.14	Cop20k_A matrix computation time	19
3.15	FEM_3D_thermal1 matrix computation time	19
3.16	Cage4 matrix performance	19
3.17	DC1 matrix performance	20
3.18	Amazon0302 matrix performance	20
3.19	Cop20k_A matrix performance	20
3.20	FEM_3D_thermal1 matrix performance	21
3.21	Cop20k_A matrix execution time	21
3.22	Cop20k_A matrix execution time	21
3.23	Cop20k_A matrix execution time	22
3.24	Cop20k_A matrix execution time	22
3.25	Cop20k_A matrix execution time	22
3.26	Cop20k_A matrix communication time	23
3.27	Cop20k_A matrix communication time	23
3.28	Cop20k_A matrix communication time	23
3.29	Cop20k_A matrix communication time	24
3.30	Cop20k_A matrix communication time	24
3.31	Cop20k_A matrix computation time	24
3.32	Cop20k_A matrix computation time	25
3.33	Cop20k_A matrix computation time	25
3.34	Cop20k_A matrix computation time	25
3.35	Cop20k_A matrix computation time	26
3.36	Cop20k_A matrix performance	26

3.37	Cop20k_A matrix performance	26
3.38	Cop20k_A matrix performance	27
3.39	Cop20k_A matrix performance	27
3.40	Cop20k_A matrix performance	27

List of Tables

3.1	Sparse matrix specifications	14
-----	--	----

Chapter 1

Introduction

High-Performance Computing (HPC) is a branch of computing that uses supercomputers and server clusters to solve complex, computationally intensive problems. Unlike a personal computer with a single processor, an HPC system is made up of many processors working in parallel, considerably increasing processing capacity. This enables scientists and engineers to carry out detailed numerical simulations, such as forecasting the weather or solving structural engineering problems.

Cranfield University has two HPC systems: CRESCENT2 and DELTA. However, this report will focus exclusively on CRESCENT2, which is an HPC cluster designed to provide computing power for teaching and research. CRESCENT 2 nodes are equipped with Intel Xeon E5 2620 processors, and each node contains two 16-core processors and 16 gigabytes of RAM.

The aim of this report is to explore distributed memory parallel programming strategies for optimising the performance of sparse matrix multiplication by a fat vector, a common operation in numerical linear algebra.

Consider a sparse matrix M of dimensions $m \times n$ and a fat vector v of dimensions $n \times k$. The objective is to perform the multiplication $M \times v$, yielding a result that is of dimensions $m \times k$.

The matrix M is defined as:

$$M = \begin{pmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{m1} & m_{m2} & \cdots & m_{mn} \end{pmatrix} \quad (1.1)$$

where most elements of M are zeros.

The vector v is defined as:

$$v = \begin{pmatrix} v_{11} & v_{12} & \cdots & v_{1k} \\ v_{21} & v_{22} & \cdots & v_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n1} & v_{n2} & \cdots & v_{nk} \end{pmatrix} \quad (1.2)$$

Chapter 2

Methodology

2.1 Data Structures

In numerical computation and linear algebra, efficient use of memory and fast computation are crucial. This is particularly true when working with hollow matrices and fat vectors.

2.1.1 Sparse Matrix

The sparse matrix is represented in CSR (Compressed Sparse Row) format, which is particularly effective for storing and manipulating matrices where the majority of elements are zero. The CSR structure consists of three main vectors:

- **values**: A vector storing all the non-zero elements of the matrix.
- **rowPtr**: A vector storing the starting index for each element in the *values* vector.
- **colIndices**: A vector storing the column indices for each element in the vector *values*.

Here is an example of a sparse matrix in CSR format:

- `values = {1, 2, 3, 4}`
- `rowPtr = {0, 2, 3, 3, 4}`
- `colIndices = {0, 2, 1, 3}`

This hollow matrix can be visualised as:

$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix}$$

The **SparseMatrix** structure is defined in Appendix B.A.

2.1.2 Fat Vector

Unlike a hollow matrix, a fat vector (illustrated by equation 1.2) stores all its elements, including zeros. The data structure for a fat vector is a two-dimensional array, where each row represents a separate vector. The **FatVector** structure is defined in Appendix B.A.

2.2 Sequential Algorithm

Let M be a sparse matrix of size $m \times n$ with z non-zero elements, stored in CSR format, and v be a fat vector of size $n \times k$. The sequential algorithm for multiplying M by v is implemented in Appendix B.B.

2.2.1 Algorithm Flow

Algorithm 1 Sequential algorithm

Require: M is an $m \times n$ sparse matrix

Require: v is an $n \times k$ fat vector

Ensure: $Result$ is an $m \times k$ matrix

$Result \leftarrow$ zero matrix of size $m \times k$

for $i \leftarrow 0$ **to** $m - 1$ **do**

for each non-zero element (j, value) in row i of M **do**

for $j \leftarrow 0$ **to** $k - 1$ **do**

$Result[i][l] \leftarrow Result[i][l] + (\text{value} \times v[j][l])$

end for

end for

end for

return $Result$

The algorithmic flow can be more explicitly detailed by:

1. **Initialisation:** Create a zero matrix of size $m \times k$ to store the result.
2. **Row-wise Processing:** Iterate over each row i of matrix M , leveraging the CSR format to efficiently access non-zero elements.
3. **Element-wise Multiplication and Accumulation:** For each non-zero element in row i , identified by its column index j and value, conduct a nested iteration over the columns l of vector v , multiplying the non-zero element by the corresponding vector element and accumulating the result in $Result[i][l]$.

2.2.2 Temporal Complexity Analysis

Given a sparse matrix M of size $m \times n$ with z non-zero elements and a fat vector v of size $n \times k$, the serial algorithm for multiplying $M \times v$ iterates through each non-zero element of the matrix M to compute the product.

The algorithm performs two operations (a multiplication and an addition) for each non-zero element with respect to each column of v , resulting in a total of $2zk$ operations.

Hence, the time complexity of the serial sparse matrix-fat vector multiplication algorithm can be expressed as:

$$T(n) = O(zk) \quad (2.1)$$

2.3 Line-Based Parallelism

This algorithm partitions a sparse matrix into row chunks and distributes these chunks across multiple processes for parallel computation in a line-based manner.

2.3.1 Algorithm Flow

Algorithm 2 Row-wise Parallel Sparse Matrix-Fat Vector Multiplication

Require: M is an $m \times n$ sparse matrix
Require: v is an $n \times k$ fat vector
Require: $worldSize$ is the number of processes
Require: $worldRank$ is the rank of the current process
Ensure: $finalResult$ is an $m \times k$ matrix, result of $M \times v$
 $rowsPerProcess \leftarrow m / worldSize$
 $extraRows \leftarrow m \bmod worldSize$
 $startRow \leftarrow worldRank \times rowsPerProcess + \min(worldRank, extraRows)$
 $endRow \leftarrow startRow + rowsPerProcess$
if $worldRank < extraRows$ **then**
 $endRow \leftarrow endRow + 1$
end if
Initialise $localResult$ with zeros of size $(endRow - startRow) \times k$
for $i \leftarrow startRow$ **to** $endRow - 1$ **do**
 for each non-zero element $(j, value)$ in row i of M **do**
 for $k \leftarrow 0$ **to** $k - 1$ **do**
 $localIndex \leftarrow (i - startRow) \times k + k$
 $localResult[localIndex] \leftarrow localResult[localIndex] + value \times v[j][k]$
 end for
 end for
end for
if $worldRank == 0$ **then**
 Prepare $recvCounts$ and $displacements$ for gathering
end if
MPI.Gatherv($localResult, \dots$)
if $worldRank == 0$ **then**
 Reassemble $finalResult$ from all $localResults$
 return $finalResult$
end if

The algorithmic flow can be explicitly detailed by:

1. **Initialisation:** Obtain MPI world size and rank to determine each process's role.
2. **Row Distribution:** Assign a subset of rows from the sparse matrix to each process based on the rank, ensuring an even distribution with possible adjustments for any remainder.
3. **Local Computation:** Each process calculates the product of its assigned rows with the fat vector, storing results in a local vector.
4. **Gather Results:** Use `MPI_Gatherv` to collect the local result vectors from all processes into a single vector at the root process.
5. **Final Result Reconstruction:** The root process reconstructs the final result matrix from the gathered vector.

2.3.2 Time Complexity Analysis

Given a sparse matrix M of size $m \times n$ with z non-zero elements and a fat vector v of size $n \times k$, the line-based algorithm distributes the multiplication task across p processors.

2.3.2.1 Computational Complexity

Each process computes a portion of the result vector, working on approximately m/p rows of the sparse matrix. The computation time (T_{comp}) is therefore influenced by the distribution of non-zero elements across the rows. Assuming a uniform distribution, the computation time can be estimated as:

$$T_{\text{comp}} = O\left(\frac{z \times k}{p}\right) \quad (2.2)$$

2.3.2.2 Communication Complexity

After computation, each process holds a part of the result vector that needs to be combined to form the final result. The main communication cost arises from gathering these parts at a single process or distributing them among all processes.

- **Startup Overhead:** The initiation of a communication incurs a latency cost, α , which is significant when the number of messages is large.
- **Data Transmission Cost:** Each process sends its portion of the result vector, incurring a cost proportional to the amount of data sent, the number of processes and the network bandwidth, denoted by β .

The communication time complexity (T_{comm}) can be approximated as:

$$T_{\text{comm}} = p \times \alpha + \beta \times \frac{m}{p} \times k \quad (2.3)$$

2.3.2.3 Final Result Reconstruction

After gathering the local results, the root process reconstructs the final result matrix. This step has a time complexity proportional to the size of the final matrix, $O(m \times k)$.

2.3.2.4 Overall Time Complexity

The overall time complexity, including both computation and communication, is given by:

$$T_{\text{overall}} = T_{\text{comp}} + T_{\text{comm}} = O\left(\frac{z \times k}{p}\right) + p \times \alpha + \beta \times \frac{m}{p} \times k + O(m \times k) \quad (2.4)$$

2.3.3 Performance Analysis

2.3.3.1 Performance Dependence

- Number of processes (p): Performance improves with p until communication overhead becomes significant.
- Number of rows (m): Affects workload distribution. Performance optimal when $p \leq m$.
- Number of non-zero elements (z): Higher z increases computation workload but mitigated by parallel execution.
- Number of columns (k): Increases computation linearly. Each row computation involves all columns.

2.3.3.2 Expected Performance

Optimal when $p \leq m$ with even workload distribution. Performance may degrade if $p > m$ due to idle processes or when communication overhead outweighs computation benefits.

2.4 Column-Wise Parallelism

This algorithm distributes the non-zero elements of a sparse matrix among different processes, enabling parallel computation focused on each non-zero element.

2.4.1 Algorithm Flow

Algorithm 3 Column-wise Parallel Sparse Matrix-Fat Vector Multiplication

Require: M is an $m \times n$ sparse matrix
Require: v is an $n \times k$ fat vector
Require: $worldSize$ is the number of processes
Require: $worldRank$ is the rank of the current process
Ensure: $finalResult$ is an $m \times k$ matrix, result of $M \times v$
 $colsPerProcess \leftarrow k / worldSize$
 $extraCols \leftarrow k \bmod worldSize$
 $startCol \leftarrow worldRank \times colsPerProcess$
 $endCol \leftarrow startCol + colsPerProcess$
if $worldRank == worldSize - 1$ **then**
 $endCol \leftarrow endCol + extraCols$
end if
 $localSize \leftarrow m \times (endCol - startCol)$
Initialise $localResult$ with zeros of size $localSize$
for $col \leftarrow startCol$ **to** $endCol - 1$ **do**
 for $row \leftarrow 0$ **to** $m - 1$ **do**
 $sum \leftarrow 0$
 for each non-zero element $(i, value)$ in row row of M **do**
 $sum \leftarrow sum + value \times v[i][col]$
 end for
 $localResult[row][col - startCol] \leftarrow sum$
 end for
end for
if $worldRank == 0$ **then**
 Initialise $finalResult$ with zeros of size $m \times k$
end if
Gather $localResult$ from all processes to $finalResult$ at root
if $worldRank == 0$ **then**
 State Reassemble $finalResult$ from gathered $localResults$
 return $finalResult$
end if

The algorithmic flow can be explicitly detailed by:

1. **Initialisation:** Obtain MPI world size and rank to determine each process's role.
2. **Column Distribution:** Calculate the number of columns each process will handle, distributing any extra columns to the last processes, and define the start and end column indices for each process.

3. **Local Computation:** Each process computes a portion of the multiplication result for its assigned columns, iterating through the sparse matrix rows and the relevant columns of the fat vector.
4. **Gather Results:** Use `MPI_Gatherv` to collect the local results from all processes into a single result vector at the root process.
5. **Final Result Reconstruction:** The root process reassembles the gathered results into the final fat vector matrix, ensuring the elements are correctly positioned according to their original indices.

2.4.2 Temporal Complexity Analysis

Consider a sparse matrix M of size $m \times n$ with z non-zero elements and a fat vector v of size $n \times k$, the column-wise algorithm distributes the multiplication task across p processors.

2.4.2.1 Computation Time Complexity

Each process is responsible for approximately k/p columns of the fat vector. The computation time (T_{comp}) is therefore influenced by the distribution of non-zero elements across the columns. Assuming a uniform distribution, the computation time can be estimated as:

$$T_{\text{comp}} = O\left(\frac{k \times z}{p}\right) \quad (2.5)$$

2.4.2.2 Communication Time Complexity

After local computations, partial results must be gathered at the root process. The communication time complexity (T_{comm}) can be approximated as:

$$T_{\text{comm}} = p \times \alpha + \beta \times m \times \frac{k}{p} \quad (2.6)$$

2.4.2.3 Final Result Reconstruction

After gathering the local results, the root process reconstructs the final result matrix. This step has a time complexity proportional to the size of the final matrix, $O(m \times k)$.

2.4.2.4 Overall Time Complexity

The overall time complexity of the column-wise parallel sparse matrix-vector multiplication algorithm is dominated by the sum of computation and communication complexities:

$$T_{\text{overall}} = T_{\text{comp}} + T_{\text{comm}} = O\left(\frac{k \times z}{p}\right) + p \times \alpha + \beta \times m \times \frac{k}{p} + O(m \times k) \quad (2.7)$$

2.4.3 Performance Analysis

2.4.3.1 Performance Dependence

- Number of processes (p): Performance improvement depends on the ratio of k to p .
- Number of rows (m): Less impact on performance scaling compared to p and k .
- Number of non-zero elements (z): Impacts computation time, balanced by parallel processing.
- Number of columns (k): Critical for performance. Optimal when large and divisible by p .

2.4.3.2 Expected Performance

Best when k is significantly larger than p and divisible, allowing efficient parallel processing with minimal communication overhead. Performance may degrade if $p > k$ due to idle processes or when communication outweighs computation benefits.

2.5 Non-Zero Element Parallelism

This algorithm combines line-based and non-zero element-based approaches by distributing chunks of rows to each process and then performing parallel computations on the non-zero elements within those chunks.

2.5.1 Algorithm Flow

Algorithm 4 Non-Zero Element Parallel Sparse Matrix-Fat Vector Multiplication

Require: M is an $m \times n$ sparse matrix

Require: v is an $n \times k$ fat vector

Require: $worldSize$ is the number of processes

Require: $worldRank$ is the rank of the current process

Ensure: $finalResult$ is an $m \times k$ matrix, result of $M \times v$

Calculate the total number of non-zero elements and distribute them among MPI processes

Determine $startIdx$ and $endIdx$ for non-zero elements for the current process

Map non-zero element indices to their corresponding row indices in the sparse matrix

Initialise $localResult$ with zeros of size $m \times k$

for $idx \leftarrow startIdx$ **to** $endIdx - 1$ **do**

 Determine row , col , and $value$ for each non-zero element

for $k \leftarrow 0$ **to** $k - 1$ **do**

$localResult[row \times k + k] \leftarrow localResult[row \times k + k] + value \times v[col][k]$

end for

end for

Use MPI_Reduce to sum up $localResults$ from all processes to $flatFinalResult$ at the root process

if $worldRank == 0$ **then**

 Reconstruct $finalResult$ from $flatFinalResult$

return $finalResult$

end if

The algorithmic flow can be explicitly detailed by:

1. **Initialisation:** Obtain MPI world size and rank to determine each process's role.
2. **Non-Zero Elements Distribution:** Calculate each process's share of non-zero elements in the sparse matrix.
3. **Local Computation:** Each process multiplies its assigned non-zero elements with corresponding columns in the fat vector, accumulating results locally.
4. **Gather Results:** Use Reduce to sum up all local results into a single vector on the root process.
5. **Final Result Reconstruction:** The root process reconstructs the final result matrix from the gathered vector.

2.5.2 Temporal Complexity Analysis

Given a sparse matrix M of size $m \times n$ with z non-zero elements and a fat vector v of size $n \times k$, the non-zero elements algorithm distributes the multiplication task across p processors.

2.5.3 Computation Complexity

Each process is responsible for a subset of the non-zero elements. The total computation workload is proportional to the number of non-zero elements z , distributed evenly among p processes:

$$T_{\text{comp}} = O\left(\frac{z \times k}{p}\right) \quad (2.8)$$

Given that each non-zero element computation involves a multiplication and an addition, the computation complexity remains linear with respect to the number of non-zero elements handled by each process.

2.5.4 Communication Complexity

The communication complexity involves reducing the local results from all the processes to a single result at the root process. The communication time complexity (T_{comm}) can be approximated as:

$$T_{\text{comm}} = p \times \alpha + \beta \times m \times \frac{k}{p} \quad (2.9)$$

where:

- α is the latency cost of initiating a communication.
- β is the cost of data transmission.

2.5.4.1 Final Result Reconstruction

After reducing the local results, the root process reconstructs the final result matrix. This step has a time complexity proportional to the size of the final matrix, $O(m \times k)$.

2.5.5 Overall Time Complexity

The overall time complexity of the SparseMatrixFatVectorMultiplyNonZeroElement algorithm is the sum of computation and communication complexities:

$$T_{\text{overall}} = T_{\text{comp}} + T_{\text{comm}} = O\left(\frac{z \times k}{p}\right) + p \times \alpha + \beta \times m \times \frac{k}{p} + O(m \times k) \quad (2.10)$$

2.5.6 Performance Analysis

2.5.6.1 Performance Dependence

- Number of processes (p): Performance improves with p , but aggregation communication can be a bottleneck.
- Number of rows (m): Impacts performance through non-zero element distribution.
- Number of non-zero elements (z): Directly proportional to computation workload, benefiting from parallel execution.
- Number of columns (k): Increases workload linearly, balanced by distributing non-zero elements across processes.

2.5.6.2 Expected Performance

Highly efficient for matrices with large and evenly distributed z , maximising parallelism. Optimal when z/p is large, minimising idle time and communication overhead.

2.6 Performance Metrics

In order to evaluate the performance of all the algorithms, they were implemented by the main program (see appendix B.G). Here is the flow of this program:

1. **Initialisation:** Start MPI and PETSc environments.
2. **Matrix and Vector Preparation:**
 - Read the sparse matrix from a file. This feature is provided by the `readMatrixMarketFile` method (see appendix B.F), created thanks to the Matrix Market I/O library (1).
 - Generate a random fat vector with dimensions matching the matrix and a specified number of columns.
 - Distribute the matrix and vector across all processes.
3. **Serial Multiplication:** Perform matrix-vector multiplication using a serial algorithm and measure execution time.
4. **Line-Based Parallel Multiplication:** Perform matrix-vector multiplication using a line-based parallel algorithm and measure execution time.
5. **Column-Based Parallel Multiplication:** Perform matrix-vector multiplication using a column-based parallel algorithm and measure execution time.
6. **Non-Zero Element Parallel Multiplication:** Perform matrix-vector multiplication using a non-zero element parallel algorithm and measure execution time.
7. **PETSc Implementation:**
 - Convert the sparse matrix and fat vector into PETSc formats.
 - Execute matrix-vector multiplication using PETSc operations.
 - Convert the result back into a Fat Vector format for comparison.
8. **Result Comparison:** Compare the results of serial, parallel, and PETSc implementations for correctness.
9. **Finalisation:** Clean up and close MPI and PETSc environments.

This program was executed with different sparse matrices, number of columns in the fat vector, and number of processes to evaluate the performance of the algorithms. To do so, the following performance metrics were used:

Chapter 3

Results and Discussion

3.1 Results

The performance of all algorithms was evaluated using the following five sparse matrix:

Matrix Name	Dimensions	Non-Zero Elements
Cage4	9×9	49
FEM_3D_thermal1	$17,880 \times 17,880$	430,740
DC1	$116,835 \times 116,835$	766,396
Cop20k_A	$121,192 \times 121,192$	2,624,331
Amazon0302	$262,111 \times 262,111$	1,234,877

Table 3.1: Sparse matrix specifications

3.1.1 Sparse Matrix Impact

The first set of experiments focused on the impact of the sparse matrix on the performance of the algorithms.

3.1.1.1 Execution Time

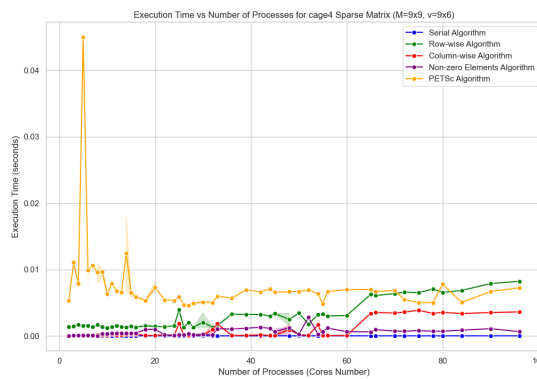


Figure 3.1: Cage4 matrix execution time

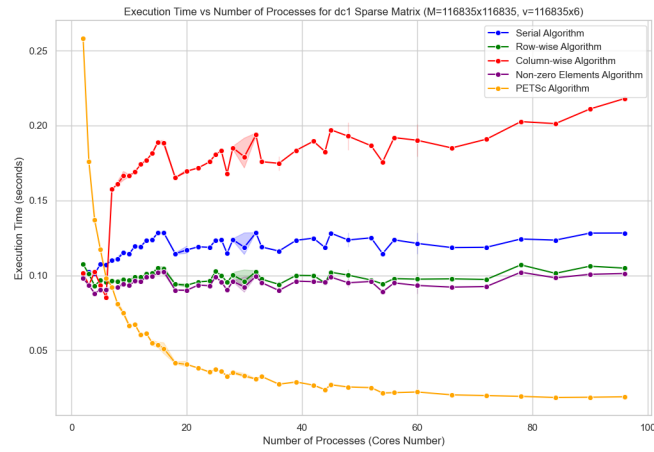


Figure 3.2: DC1 matrix execution time

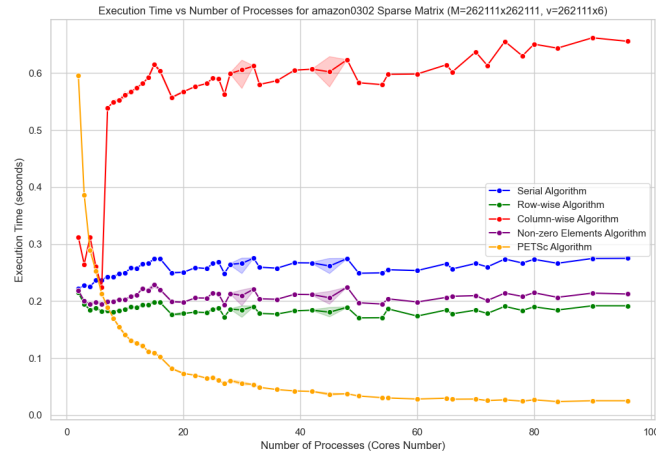


Figure 3.3: Amazon0302 matrix execution time

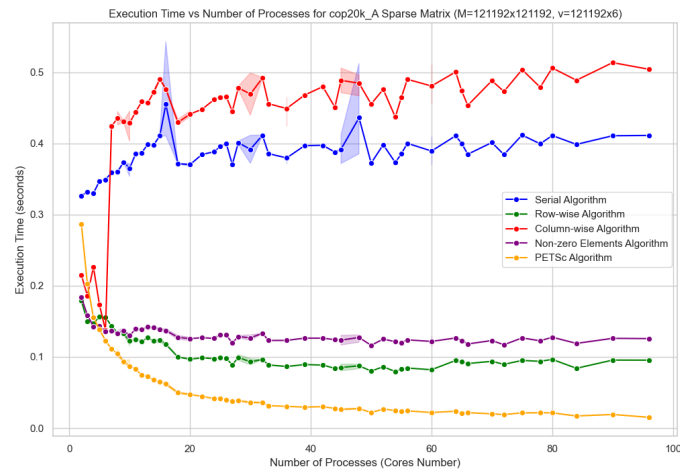


Figure 3.4: Cop20k_A matrix execution time

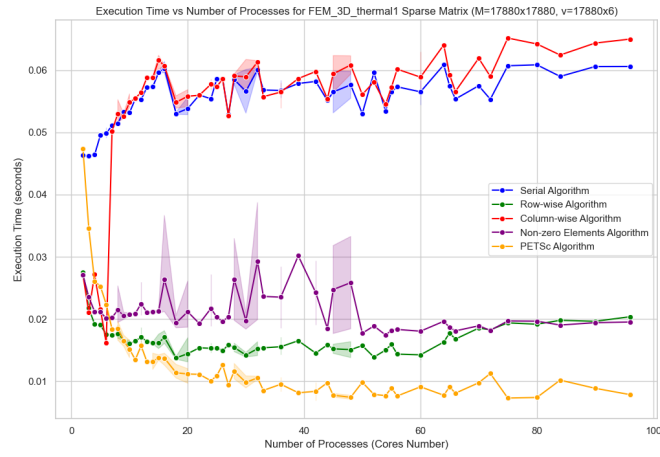


Figure 3.5: FEM_3D_thermal1 matrix execution time

3.1.1.2 Communication Time

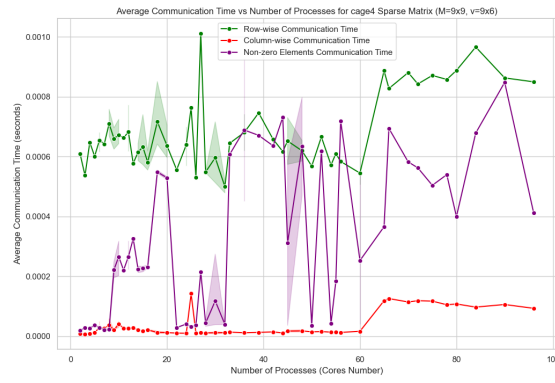


Figure 3.6: Cage4 matrix communication time

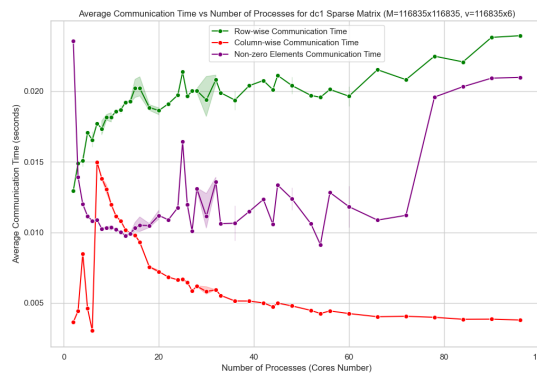


Figure 3.7: DC1 matrix communication time

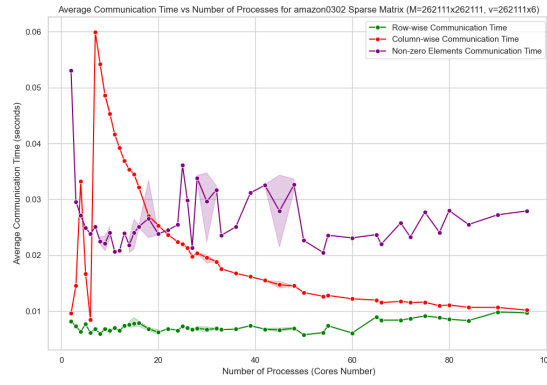


Figure 3.8: Amazon0302 matrix communication time

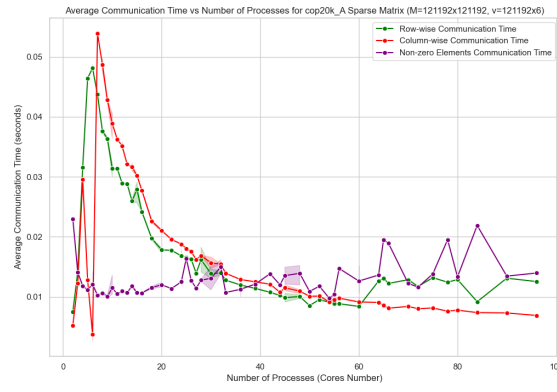


Figure 3.9: Cop20k_A matrix communication time

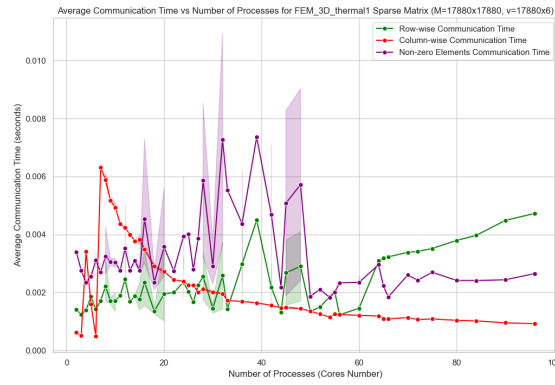


Figure 3.10: FEM_3D_thermal1 matrix communication time

3.1.1.3 Computation Time

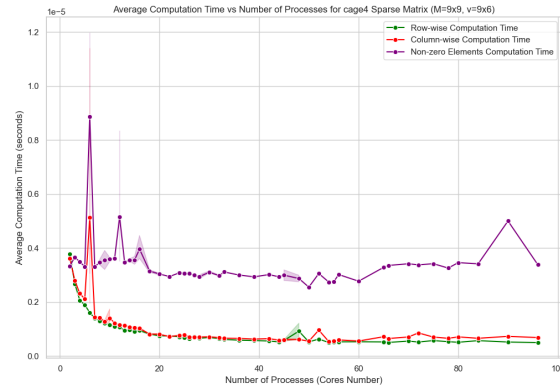


Figure 3.11: Cage4 matrix computation time

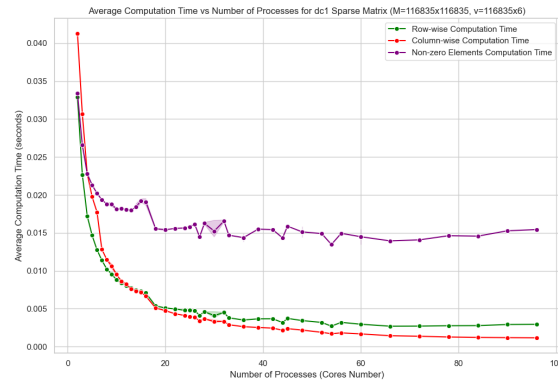


Figure 3.12: DC1 matrix computation time

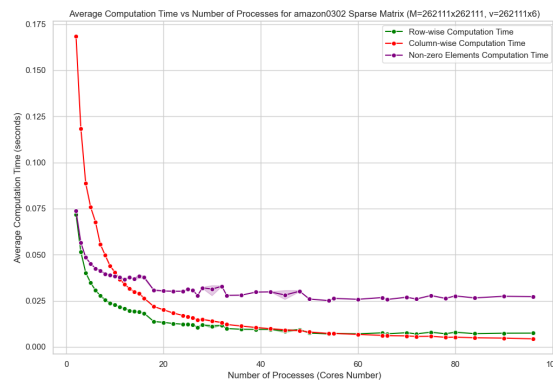


Figure 3.13: Amazon0302 matrix computation time

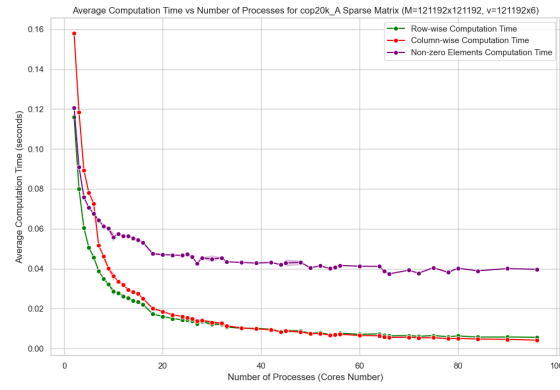


Figure 3.14: Cop20k_A matrix computation time

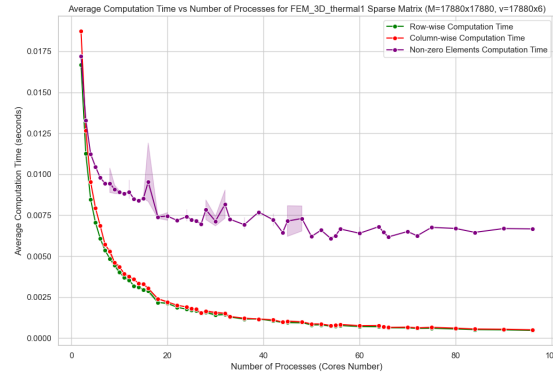


Figure 3.15: FEM_3D_thermal1 matrix computation time

3.1.1.4 Performance

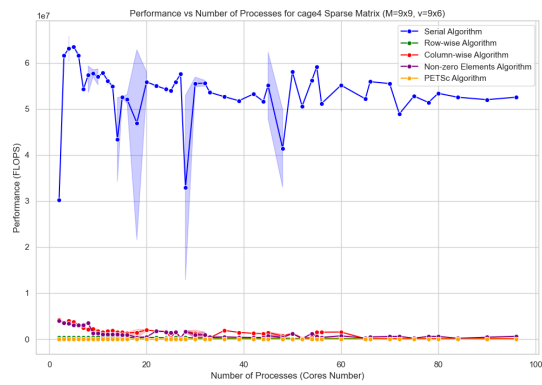


Figure 3.16: Cage4 matrix performance

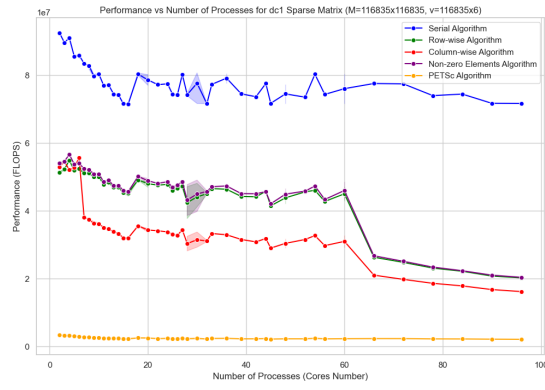


Figure 3.17: DC1 matrix performance

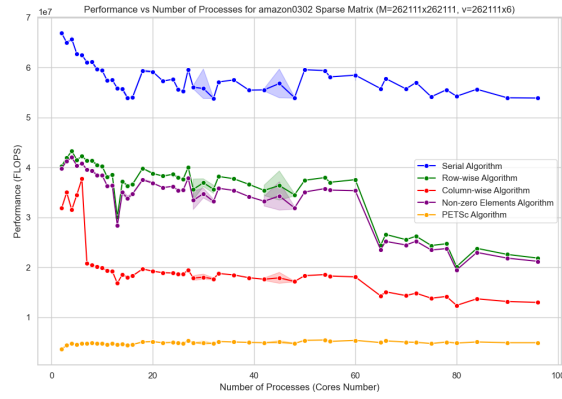


Figure 3.18: Amazon0302 matrix performance

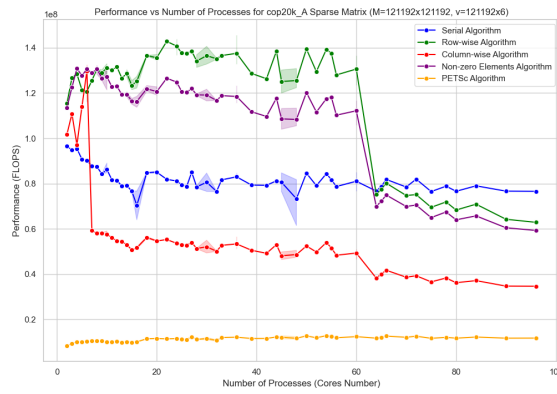


Figure 3.19: Cop20k_A matrix performance

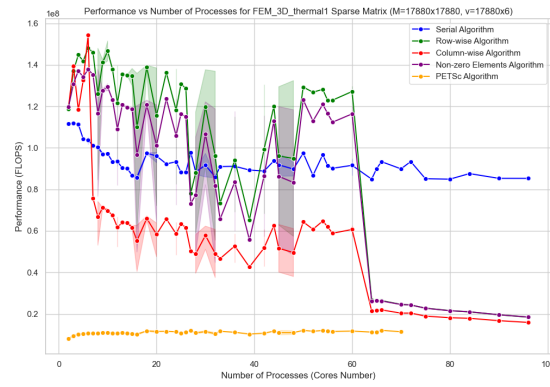


Figure 3.20: FEM_3D_thermal1 matrix performance

3.1.2 Fat Vector Impact

3.1.2.1 Execution Time

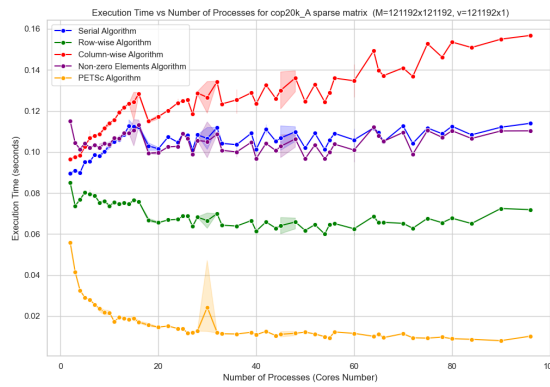


Figure 3.21: Cop20k_A matrix execution time

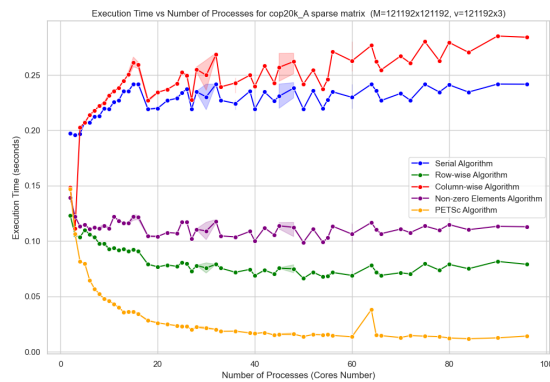


Figure 3.22: Cop20k_A matrix execution time

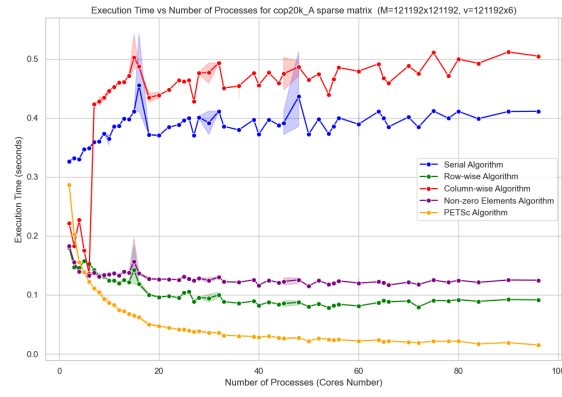


Figure 3.23: Cop20k_A matrix execution time

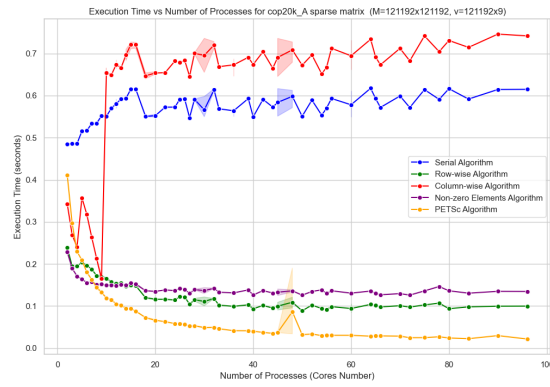


Figure 3.24: Cop20k_A matrix execution time

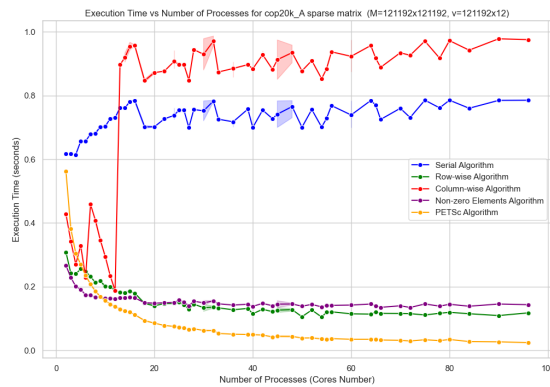


Figure 3.25: Cop20k_A matrix execution time

3.1.2.2 Communication Time

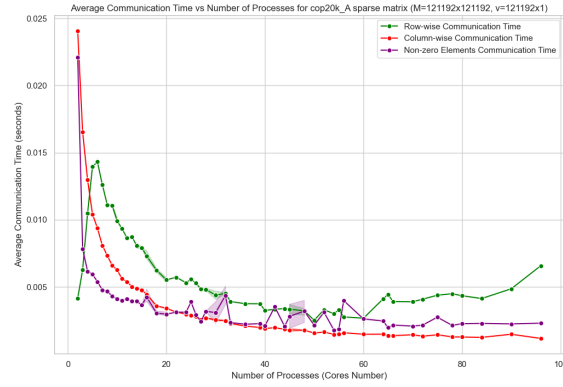


Figure 3.26: Cop20k_A matrix communication time

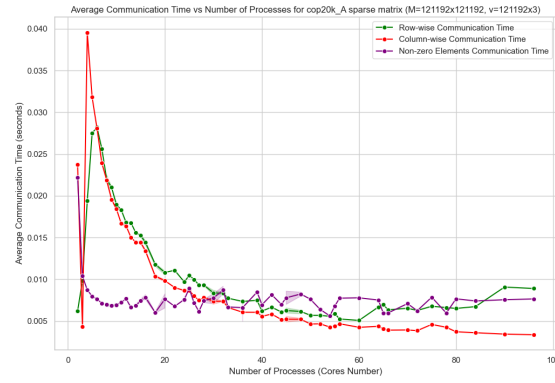


Figure 3.27: Cop20k_A matrix communication time

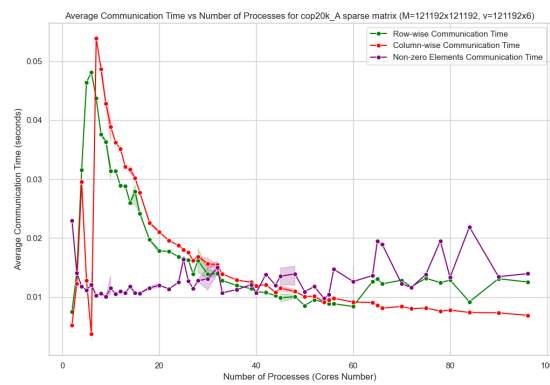


Figure 3.28: Cop20k_A matrix communication time

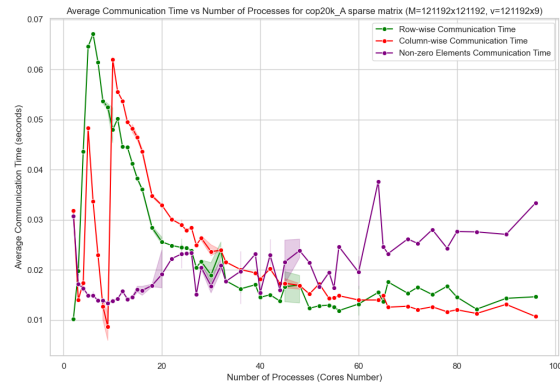


Figure 3.29: Cop20k_A matrix communication time

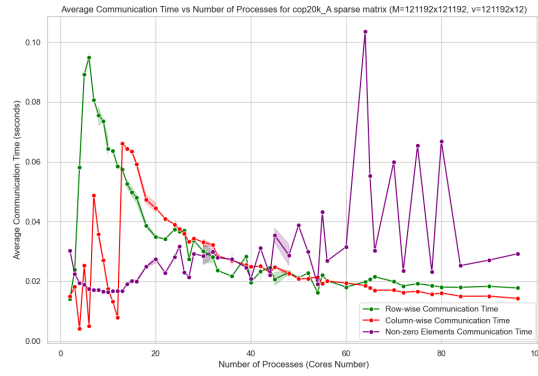


Figure 3.30: Cop20k_A matrix communication time

3.1.2.3 Computation Time

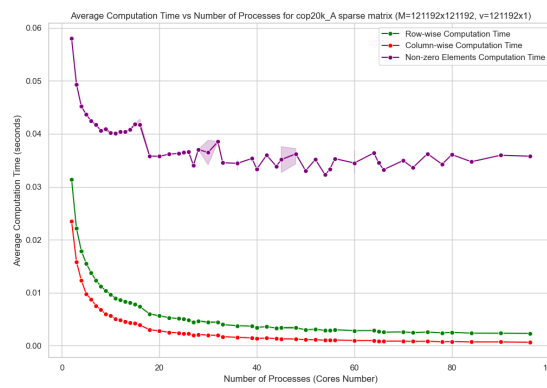


Figure 3.31: Cop20k_A matrix computation time

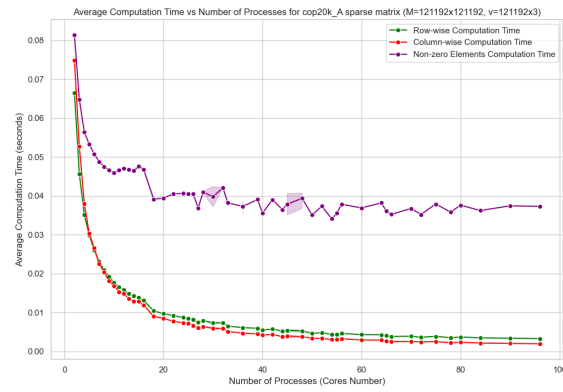


Figure 3.32: Cop20k_A matrix computation time

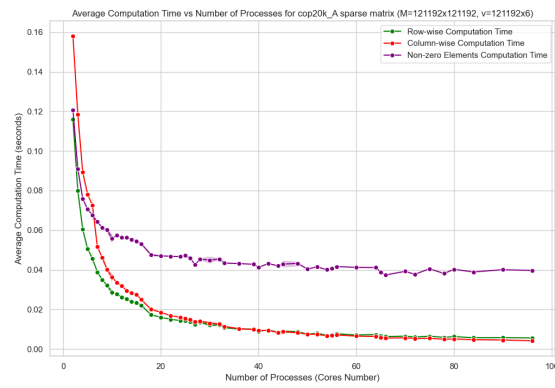


Figure 3.33: Cop20k_A matrix computation time

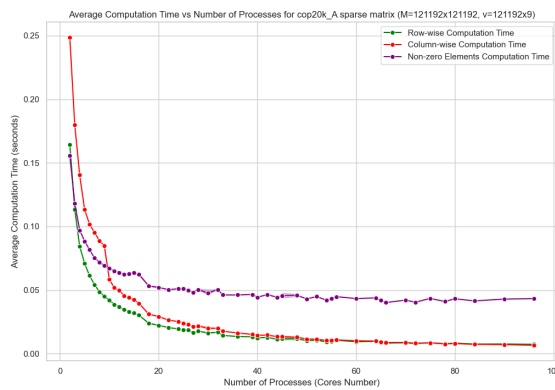


Figure 3.34: Cop20k_A matrix computation time

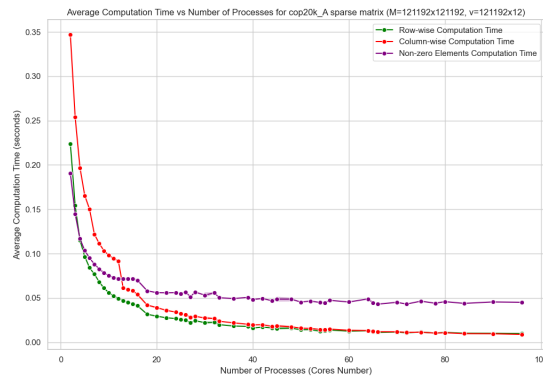


Figure 3.35: Cop20k_A matrix computation time

3.1.2.4 Performance

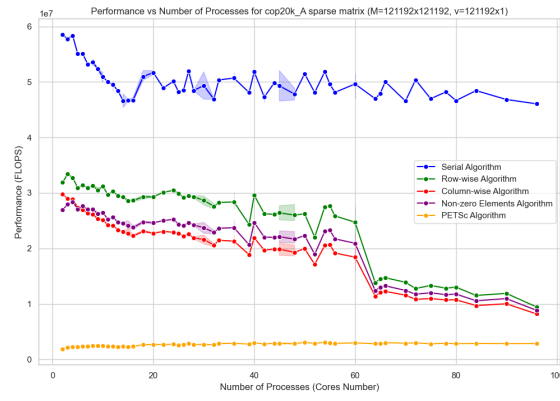


Figure 3.36: Cop20k_A matrix performance

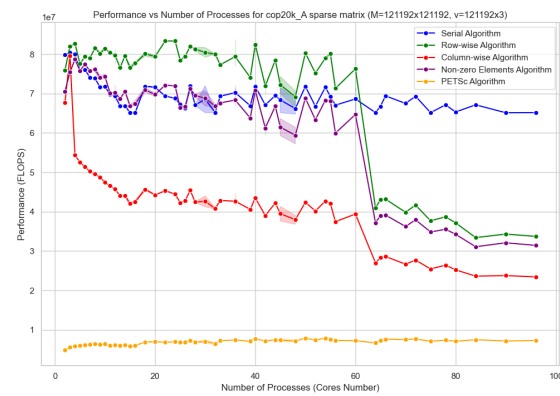


Figure 3.37: Cop20k_A matrix performance

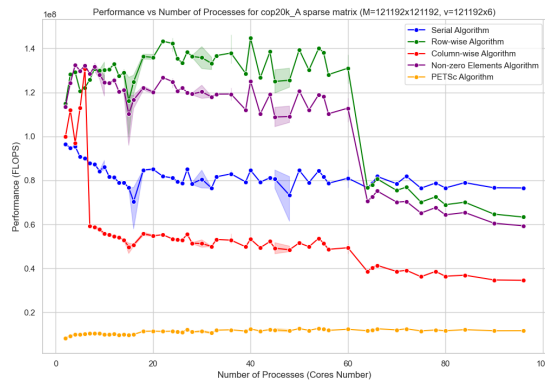


Figure 3.38: Cop20k_A matrix performance

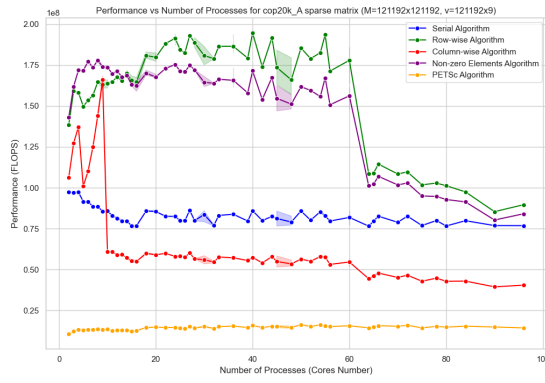


Figure 3.39: Cop20k_A matrix performance

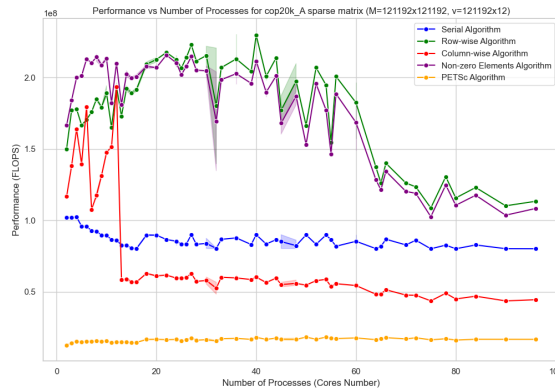


Figure 3.40: Cop20k_A matrix performance

3.2 HPC Environmental Impact

The LUMI supercomputer, based at the CSC-IT Center for Science in Finland, represents a milestone in the field of high-performance computing (HPC), not only because of its computing power, but also because of its approach to environmental sustainability.

LUMI, one of EuroHPC’s world-class supercomputers, began operating in 2021 and is expected to reach full capacity in 2023. It features an environmentally-friendly design and is considered to be one of the most energy-efficient data centres in the world.

Irina Kupiainen, who works as Programme Director for the Open Scholarship Innovation Program at the CSC, plays an important role in the development of policies relating to high-performance computing and open science. With a strong background in international policy and experience in various government and research organisations, Irina Kupiainen leads EU public affairs at the CSC, focusing on policy and international collaboration, particularly in the area of open science.

The impact of supercomputing on the environment is a major concern, not least because of the high energy demands of these systems. However, LUMI is an example of how HPC can make a positive contribution to environmental sustainability. It runs on 100% renewable energy and makes efficient use of waste heat, which can heat up to 20% of the homes in the surrounding city. This approach not only reduces the carbon footprint, but also demonstrates HPC’s potential to meet climate neutrality targets.

Furthermore, sustainability measures in HPC are not limited to energy consumption and waste heat management. The entire life cycle of the machine needs to be taken into account, including construction, modularity, scalability, recycling and reuse of materials. This holistic view can contribute to the development of a circular economy, supporting sustainability to its full potential.

In conclusion, the LUMI supercomputer, led by experts such as Irina Kupiainen and others at the CSC-IT Center for Science, illustrates how supercomputing can be both a powerful tool for scientific progress and a leader in environmental sustainability. By harnessing renewable energy sources, making efficient use of waste heat and taking into account the full lifecycle of HPC systems, LUMI is setting a precedent for how high-performance computing can contribute to a greener, more sustainable future.

Chapter 4

Conclusion

References

1. Lugowski A. fast matrix market: Fast and Full-Featured Matrix Market I/O Library; 2023. Available at: https://github.com/alugowski/fast_matrix_market. (Accessed: December 28, 2023).

Appendix A

Documentation

Appendix A.A Project tree

```
Source Code/  
  scripts /  
    batch_test.sh  
    get_csv_all.sh  
    get_csv_debug.sh  
    get_csv_specific.sh  
    mpi.sub  
  MatrixDefinitions.h  
  SparseMatrixFatVectorMultiply.h  
  SparseMatrixFatVectorMultiply.cpp  
  SparseMatrixFatVectorMultiplyRowWise.h  
  SparseMatrixFatVectorMultiplyRowWise.cpp  
  SparseMatrixFatVectorMultiplyColumnWise.h  
  SparseMatrixFatVectorMultiplyColumnWise.cpp  
  SparseMatrixFatVectorMultiplyNonZeroElement.h  
  SparseMatrixFatVectorMultiplyNonZeroElement.cpp  
  utils.h  
  utils.cpp  
  main.cpp  
results /  
  fat_vector_dim /  
    <sparse_matrix>_<k>_<metric>.png  
  matrix_dim /  
    <sparse_matrix>_<k>_<metric>.png
```

Appendix A.B Getting Started

To run the program, follow these steps:

1. Install the required libraries: mpi & petsc
2. Compile the main program using the following command:

```

    mmpicxx -o <executable_name> -I${PETSC_DIR}/include -I${
PETSC_DIR}/${PETSC_ARCH}/include -L${PETSC_DIR}/${PETSC_ARCH}/lib -lpetsc
SparseMatrixFatVectorMultiply.cpp main_verify.cpp utils.cpp
SparseMatrixFatVectorMultiplyColumnWise.cpp
SparseMatrixFatVectorMultiplyNonZeroElement.cpp
SparseMatrixFatVectorMultiplyRowWise.cpp

```

3. Run the program using the following command:

```

    mpirun -np <number_of_processes> <executable_name> <k> <
sparse_matrix_file_pathw>

```

Appendix A.C Methods Overview

A.C.1 Utils.h

A.C.1.1 ConvertPETScMatToFatVector

Description: Converts a PETSc matrix to a FatVector stucture.

Parameters:

- Mat C - PETSc matrix to be converted.

Returns: FatVector - Fat vector representation of the PETSc matrix.

A.C.1.2 areMatricesEqual

Description: Compares two matrices for equality within a specified tolerance.

Parameters:

- FatVector &mat1 - First matrix.
- FatVector &mat2 - Second matrix.
- double tolerance - Tolerance for comparison.

Returns: bool - True if matrices are equal within the tolerance, false otherwise.

A.C.1.3 readMatrixMarketFile

Description: Reads a matrix from a Matrix Market file into a sparse matrix format.

Parameters:

- std::string &filename - Name of the Matrix Market file.

Returns: SparseMatrix - Sparse matrix read from the file.

A.C.1.4 generateLargeFatVector

Description: Generates a random Fat Vector with specified dimensions.

Parameters:

- `int n` - Number of rows.
- `int k` - Number of columns.

Returns: `FatVector` - Generated fat vector.

A.C.1.5 serialize and deserialize

Description: Serializes and deserializes a `FatVector` to and from a flat array, respectively.

Parameters for serialize:

- `FatVector &fatVec` - fat vector to serialize.

Returns: `std::vector<double>` - Flat array containing the serialized data.

Parameters for deserialize:

- `std::vector<double> &flat` - Flat array to deserialize.
- `int rows` - Number of rows in the fat vector.
- `int cols` - Number of columns in the fat vector.

Returns: `FatVector` - Deserialized fat vector.

A.C.2 SparseMatrixFatVectorMultiply.h

A.C.2.1 sparseMatrixFatVectorMultiply

Description: Executes the multiplication using a sequential algorithm.

Parameters:

- `SparseMatrix &sparseMatrix` - The sparse matrix.
- `FatVector &fatVector` - The Fat Vector.
- `int vecCols` - Number of columns in the Fat Vector.

Returns: `FatVector` - Result of the multiplication.

A.C.3 SparseMatrixFatVectorMultiplyRowWise.h

A.C.3.1 sparseMatrixFatVectorMultiplyRowWise

Description: Multiplies a sparse matrix with a Fat Vector using row-wise distribution.

Parameters:

- SparseMatrix &sparseMatrix - The sparse matrix.
- FatVector &fatVector - The Fat Vector.
- int vecCols - Number of columns in the Fat Vector.

Returns: FatVector - Result of the multiplication.

A.C.4 SparseMatrixFatVectorMultiplyColumnWise.h

A.C.4.1 sparseMatrixFatVectorMultiplyColumnWise

Description: Executes the multiplication using column-wise parallel algorithm.

Parameters:

- SparseMatrix &sparseMatrix - The sparse matrix.
- FatVector &fatVector - The Fat Vector.
- int vecCols - Number of columns in the Fat Vector.

Returns: FatVector - Result of the multiplication.

A.C.5 SparseMatrixFatVectorMultiplyNonZeroElement.h

A.C.5.1 sparseMatrixFatVectorMultiplyNonZeroElement

Description: Executes the multiplication using non-zero element parallel algorithm.

Parameters:

- SparseMatrix &sparseMatrix - The sparse matrix.
- FatVector &fatVector - The Fat Vector.
- int vecCols - Number of columns in the Fat Vector.

Returns: FatVector - Result of the multiplication.

Appendix B

Source Codes

Appendix B.A Data Structures

Data structures of the sparse matrix and fat vector.

```
1  #ifndef MATRIXDEFINITIONS_H
2  #define MATRIXDEFINITIONS_H
3
4  #include <vector>
5
6  //
7  /**
8   * @brief Struct to represent a sparse matrix
9   *
10  * @param values    Non-zero values
11  * @param colIndices Column indices of non-zero values
12  * @param rowPtr    Row pointers
13  */
14  struct SparseMatrix
15  {
16      std::vector<double> values;
17      std::vector<int> colIndices;
18      std::vector<int> rowPtr;
19  };
20
21  // Type definition for a Fat Vector
22  typedef std::vector<std::vector<double>> FatVector;
23
24  #endif
```

Appendix B.B Sequential Algorithm

Sequential algorithm for multiplying a sparse matrix by a fat vector.

B.B.1 Declaration File

```

1 #ifndef SPARSEMATRIXFATVECTORMULTIPLY_H
2 #define SPARSEMATRIXFATVECTORMULTIPLY_H
3
4 #include "MatrixDefinitions.h"
5
6 /**
7  * @brief Function to execute the sparse matrix-fat vector multiplication using
8  * sequential algorithm
9  *
10  * @param sparseMatrix Sparse matrix
11  * @param fatVector Fat Vector
12  * @param vecCols Number of columns in the Fat Vector
13  * @return FatVector Result of the multiplication
14  */
15 FatVector sparseMatrixFatVectorMultiply(const SparseMatrix &sparseMatrix,
16                                         const FatVector &fatVector, int vecCols
17                                         );
18 #endif

```

B.B.2 Implementation File

```

1 #include "SparseMatrixFatVectorMultiply.h"
2
3 /**
4  * @brief Function to execute the sparse matrix-Fat Vector multiplication using
5  * sequential algorithm
6  *
7  * @param sparseMatrix Sparse matrix
8  * @param fatVector Fat Vector
9  * @param vecCols Number of columns in the Fat Vector
10  * @return FatVector Result of the multiplication
11  */
12 FatVector sparseMatrixFatVectorMultiply(const SparseMatrix &sparseMatrix,
13                                         const FatVector &fatVector, int vecCols
14                                         )
15 {
16     // Initialisation of the result vector
17     FatVector result(sparseMatrix.numRows, std::vector<double>(vecCols, 0.0));
18     // Iterate over the rows of the sparse matrix
19     for (int i = 0; i < sparseMatrix.numRows; ++i)
20     {
21         // Iterate over the non-zero elements in the current row
22         for (int j = sparseMatrix.rowPtr[i]; j < sparseMatrix.rowPtr[i + 1]; ++j)
23         {
24             // Iterate over the columns of the Fat Vector
25             for (int k = 0; k < vecCols; ++k)
26             {
27                 result[i][k] += sparseMatrix.values[j] * fatVector[sparseMatrix.colIndices[j]][k]; // Compute the result
28             }
29         }
30     }
31     // Return the result
32     return result;
33 }

```

Appendix B.C Line-Based Parallelism

Parallel algorithm for multiplying a sparse matrix by a fat vector using line-based parallelism.

B.C.1 Declaration File

```

1 #ifndef SPARSEMATRIXFATVECTORMULTIPLYROWWISE_H
2 #define SPARSEMATRIXFATVECTORMULTIPLYROWWISE_H
3
4 #include "MatrixDefinitions.h"
5 #include <iostream> // std::cout
6
7 /**
8  * @brief Function to multiply a sparse matrix with a Fat Vector using row-wise
9  * distribution
10  *
11  * @param sparseMatrix The sparse matrix to be multiplied
12  * @param fatVector The Fat Vector to be multiplied
13  * @param vecCols Number of columns in the Fat Vector
14  * @return FatVector Result of the multiplication
15  */
16 FatVector sparseMatrixFatVectorMultiplyRowWise(const SparseMatrix &sparseMatrix,
17                                                const FatVector &fatVector,
18                                                int vecCols);
19 #endif

```

B.C.2 Implementation File

```

1 #include <mpi.h>
2 #include "SparseMatrixFatVectorMultiplyRowWise.h"
3
4 /**
5  * @brief Function to multiply a sparse matrix with a Fat Vector using row-wise
6  * distribution
7  *
8  * @param sparseMatrix The sparse matrix to be multiplied
9  * @param fatVector The Fat Vector to be multiplied
10  * @param vecCols Number of columns in the Fat Vector
11  * @return FatVector Result of the multiplication
12  */
13 FatVector sparseMatrixFatVectorMultiplyRowWise(const SparseMatrix &sparseMatrix,
14                                                const FatVector &fatVector,
15                                                int vecCols)
16 {
17     // Retrieve the rank and size of the MPI world
18     int worldSize, worldRank;
19     MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
20     MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);
21
22     // ===== FOR DEBUGGING ONLY - START LOCAL COMPUTATION
23     // double computation_start = MPI_Wtime();
24     // ===== FOR DEBUGGING ONLY - START LOCAL COMPUTATION
25
26     // Distribute rows among processes
27     int rowsCountPerProcess = sparseMatrix.numRows / worldSize;
28     // Number of rows per process
29     int extraRows = sparseMatrix.numRows % worldSize;
30     // Number of extra rows to be distributed
31     among processes

```

```

28     int startRow = worldRank * rowsCountPerProcess + std::min(worldRank, extraRows)
29         ; // Starting row index for the current process
30     int endRow = startRow + rowsCountPerProcess + (worldRank < extraRows ? 1 : 0);
31         // Ending row index for the current process
32
33     // Local computation
34     int localSize = (endRow - startRow) * vecCols; // Number of elements in the
35         local result vector
36     std::vector<double> localResult(localSize); // Local result vector
37
38     // Iterate over the rows assigned to the current process
39     for (int i = startRow; i < endRow; ++i)
40     {
41         // Iterate over the non-zero elements in the current row
42         for (int j = sparseMatrix.rowPtr[i]; j < sparseMatrix.rowPtr[i + 1]; ++j)
43         {
44             int colIndex = sparseMatrix.colIndices[j]; // Column index of the non-
45                 zero element
46
47             // Iterate over the columns of the Fat Vector
48             for (int k = 0; k < vecCols; ++k)
49             {
50                 int localIndex = (i - startRow) * vecCols + k;
51                 // Index of the element in the
52                 local result vector
53                 localResult[localIndex] += sparseMatrix.values[j] * fatVector[
54                     colIndex][k]; // Compute the result
55             }
56         }
57     }
58
59     // ===== FOR DEBUGGING ONLY - STOP LOCAL COMPUTATION
60     // TIMER =====
61     // double computation_end = MPI_Wtime();
62     // double local_computation_time = computation_end - computation_start;
63     // ===== FOR DEBUGGING ONLY - STOP LOCAL COMPUTATION
64     // TIMER =====
65
66     // ===== FOR DEBUGGING ONLY - START COMMUNICATION TIMER
67     // =====
68     // Start timing for communication
69     // double communication_start = MPI_Wtime();
70     // ===== FOR DEBUGGING ONLY - START COMMUNICATION TIMER
71     // =====
72
73     // Preparation for Gather operation
74     std::vector<int> recvCounts(worldSize), displacements(worldSize);
75     if (worldRank == 0)
76     {
77         int totalSize = 0; // Total number of elements to be received
78
79         // Compute the number of elements to be received from each process
80         for (int rank = 0; rank < worldSize; ++rank)
81         {
82             int startRowThisRank = rank * rowsCountPerProcess + std::min(rank,
83                 extraRows); // Starting row index for the current
84                 process
85             int endRowThisRank = startRowThisRank + rowsCountPerProcess + (rank <
86                 extraRows ? 1 : 0); // Ending row index for the current process
87             recvCounts[rank] = (endRowThisRank - startRowThisRank) * vecCols;
88                 // Number of elements to be received from
89                 the current process
90             displacements[rank] = totalSize;
91                 //
92             Displacement for the current process
93             totalSize += recvCounts[rank];
94                 //
95             Update the total number of elements to be received
96         }
97     }

```

```

78
79 // Gather all local results into the root process
80 std::vector<double> gatheredResults;
81 if (worldRank == 0)
82 {
83     gatheredResults.resize(recvCounts[0] * worldSize); // Resize the vector to
84     hold all the results
85 }
86 MPI_Gatherv(localResult.data(), localSize, MPI_DOUBLE,
87             gatheredResults.data(), recvCounts.data(),
88             displacements.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD); // Gather the
89             local results in the root process
90
91 // ===== FOR DEBUGGING ONLY - STOP COMMUNICATION TIMER
92 // =====
93 // double communication_end = MPI_Wtime();
94 // double local_communication_time = communication_end - communication_start;
95 // ===== FOR DEBUGGING ONLY - STOP COMMUNICATION TIMER
96 // =====
97
98 // ===== FOR DEBUGGING ONLY - COLLECTING AND ANALYSING
99 // PERFORMANCE DATA =====
100 // double total_computation_time = 0.0, total_communication_time = 0.0;
101 // MPI_Reduce(&local_computation_time, &total_computation_time, 1, MPI_DOUBLE,
102 // MPI_SUM, 0, MPI_COMM_WORLD);
103 // MPI_Reduce(&local_communication_time, &total_communication_time, 1,
104 // MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
105 // ===== FOR DEBUGGING ONLY - COLLECTING AND ANALYSING
106 // PERFORMANCE DATA =====
107
108 // Reconstruct the final result matrix in the root process
109 FatVector finalResult;
110 if (worldRank == 0)
111 {
112     // ===== FOR DEBUGGING ONLY - PRINTING PERFORMANCE
113     // DATA =====
114     // double avg_computation_time = total_computation_time / worldSize;
115     // double avg_communication_time = total_communication_time / worldSize;
116     // std::cout << "Row-wise Average Computation Time: " <<
117     // avg_computation_time << std::endl;
118     // std::cout << "Row-wise Average Communication Time: " <<
119     // avg_communication_time << std::endl;
120     // ===== FOR DEBUGGING ONLY - PRINTING PERFORMANCE
121     // DATA =====
122
123     finalResult.resize(sparseMatrix.numRows, std::vector<double>(vecCols, 0.0))
124     ; // Resize the final result matrix
125
126     // Iterate over the rows of the final result
127     for (int i = 0, index = 0; i < sparseMatrix.numRows; ++i)
128     {
129         // Iterate over the columns of the final result
130         for (int j = 0; j < vecCols; ++j, ++index)
131         {
132             finalResult[i][j] = gatheredResults[index]; // Copy the element of
133             the final result
134         }
135     }
136 }
137
138 // Return the final result
139 return (worldRank == 0) ? finalResult : FatVector{};
140 }

```

Appendix B.D Column-Wise Parallelism

Parallel algorithm for multiplying a sparse matrix by a fat vector using column-wise parallelism.

B.D.1 Declaration File

```

1 #ifndef SPARSEMATRIXFATVECTORMULTIPLYCOLUMNWISE_H
2 #define SPARSEMATRIXFATVECTORMULTIPLYCOLUMNWISE_H
3
4 #include "MatrixDefinitions.h"
5 #include <iostream> // std::cout
6
7 /**
8  * @brief Function to execute the sparse matrix-Fat Vector multiplication using
9  * column-wise parallel algorithm
10  *
11  * @param sparseMatrix Sparse matrix
12  * @param fatVector Fat Vector
13  * @param vecCols Number of columns in the Fat Vector
14  * @return FatVector Result of the multiplication
15  */
16 FatVector sparseMatrixFatVectorMultiplyColumnWise(const SparseMatrix &sparseMatrix,
17 const FatVector &fatVector, int vecCols);
18 #endif

```

B.D.2 Implementation File

```

1 #include <mpi.h>
2 #include "SparseMatrixFatVectorMultiplyColumnWise.h"
3 #include <numeric> // std::accumulate
4
5 /**
6  * @brief Function to execute the sparse matrix-Fat Vector multiplication using
7  * column-wise parallel algorithm
8  *
9  * @param sparseMatrix Sparse matrix
10  * @param fatVector Fat Vector
11  * @param vecCols Number of columns in the Fat Vector
12  * @return FatVector Result of the multiplication
13  */
14 FatVector sparseMatrixFatVectorMultiplyColumnWise(const SparseMatrix &sparseMatrix,
15 const FatVector &fatVector, int vecCols)
16 {
17     // Retrieve the rank and size of the MPI world
18     int worldSize, worldRank;
19     MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
20     MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);
21
22     // ===== FOR DEBUGGING ONLY - START LOCAL COMPUTATION
23     // double computation_start = MPI_Wtime();
24     // ===== FOR DEBUGGING ONLY - START LOCAL COMPUTATION
25
26     // Distribute columns among processes
27     int colsPerProcess = vecCols / worldSize;
28
29     // Number of columns per process
30     int extraCols = vecCols % worldSize;
31
32     // Number of extra columns to be distributed among processes
33 }

```



```

27     int startCol = worldRank * colsPerProcess;
28                                     //
29     Starting column index for the current process
30
31     int endCol = (worldRank != worldSize - 1) ? startCol + colsPerProcess :
32     startCol + colsPerProcess + extraCols; // Ending column index for the
33     current process
34
35     // Local computation
36     int localSize = sparseMatrix.numRows * (endCol - startCol); // Number of
37     elements in the local result vector
38     std::vector<double> localResult(localSize, 0.0);
39     // Iterate over the columns assigned to the current process
40     for (int col = startCol; col < endCol; ++col)
41     {
42         // Iterate over the rows of the sparse matrix
43         for (int i = 0; i < sparseMatrix.numRows; ++i)
44         {
45             // Iterate over the non-zero elements in the current row
46             double sum = 0.0;
47             for (int j = sparseMatrix.rowPtr[i]; j < sparseMatrix.rowPtr[i + 1]; ++j)
48             {
49                 int sparseCol = sparseMatrix.colIndices[j]; //
50                 Column index of the non-zero element
51                 sum += sparseMatrix.values[j] * fatVector[sparseCol][col]; //
52                 Compute the result
53             }
54             localResult[i * (endCol - startCol) + (col - startCol)] = sum; // Store
55             the result in the local result vector
56         }
57     }
58
59     // ===== FOR DEBUGGING ONLY - STOP LOCAL COMPUTATION
60     TIMER =====
61     // double computation_end = MPI_Wtime();
62     // double local_computation_time = computation_end - computation_start;
63     // ===== FOR DEBUGGING ONLY - STOP LOCAL COMPUTATION
64     TIMER =====
65
66     // ===== FOR DEBUGGING ONLY - START COMMUNICATION TIMER
67     =====
68     // Start timing for communication
69     // double communication_start = MPI_Wtime();
70     // ===== FOR DEBUGGING ONLY - START COMMUNICATION TIMER
71     =====
72
73     // Preparation for Gather operation
74     std::vector<int> recvCounts(worldSize), displacements(worldSize); // Number of
75     elements to be received from each process, Displacement for each process
76     if (worldRank == 0)
77     {
78         // Compute the number of elements to be received from each process
79         int displacement = 0;
80         for (int i = 0; i < worldSize; ++i)
81         {
82             int startColThisRank = i * colsPerProcess;
83
84             // Starting column index for the current process
85             int endColThisRank = (i != worldSize - 1) ? startColThisRank +
86             colsPerProcess : startColThisRank + colsPerProcess + extraCols; //
87             Ending column index for the current process
88             recvCounts[i] = sparseMatrix.numRows * (endColThisRank -
89             startColThisRank);
90                                     // Number of
91             elements to be received from the current process
92             displacements[i] = displacement;
93
94             // Displacement for the current process
95             displacement += recvCounts[i];

```

```

73         // Update the displacement
74     }
75 }
76 // Gather all local results into the root process
77 std::vector<double> gatheredResults;
78 if (worldRank == 0)
79 {
80     gatheredResults.resize(std::accumulate(recvCounts.begin(), recvCounts.end(),
81     , 0)); // Resize the vector to hold the final result
82 }
83 MPI_Gatherv(localResult.data(), localSize, MPI_DOUBLE,
84     gatheredResults.data(), recvCounts.data(),
85     displacements.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD); // Gather the
86     local results in the root process
87
88 // ===== FOR DEBUGGING ONLY - STOP COMMUNICATION TIMER
89 // =====
90 // double communication_end = MPI_Wtime();
91 // double local_communication_time = communication_end - communication_start;
92 // ===== FOR DEBUGGING ONLY - STOP COMMUNICATION TIMER
93 // =====
94
95 // ===== FOR DEBUGGING ONLY - COLLECTING AND ANALYSING
96 // PERFORMANCE DATA =====
97 // double total_computation_time = 0.0, total_communication_time = 0.0;
98 // MPI_Reduce(&local_computation_time, &total_computation_time, 1, MPI_DOUBLE,
99 // MPI_SUM, 0, MPI_COMM_WORLD);
100 // MPI_Reduce(&local_communication_time, &total_communication_time, 1,
101 // MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
102 // ===== FOR DEBUGGING ONLY - COLLECTING AND ANALYSING
103 // PERFORMANCE DATA =====
104
105 // Reconstruct the final result matrix in the root process
106 FatVector finalResult;
107 if (worldRank == 0)
108 {
109     // ===== FOR DEBUGGING ONLY - PRINTING PERFORMANCE
110     // DATA =====
111     // double avg_computation_time = total_computation_time / worldSize;
112     // double avg_communication_time = total_communication_time / worldSize;
113     // std::cout << "Column-wise Average Computation Time: " <<
114     //     avg_computation_time << std::endl;
115     // std::cout << "Column-wise Average Communication Time: " <<
116     //     avg_communication_time << std::endl;
117     // ===== FOR DEBUGGING ONLY - PRINTING PERFORMANCE
118     // DATA =====
119
120     // Reconstruct the final result matrix
121     finalResult.resize(sparseMatrix.numRows, std::vector<double>(vecCols, 0.0))
122     ; // Resize the final result matrix
123     int resultIndex = 0;
124     // Iterate over the processes
125     for (int rank = 0; rank < worldSize; ++rank)
126     {
127         int numColsThisRank = (rank != worldSize - 1) ? colsPerProcess :
128             colsPerProcess + extraCols; // Number of columns assigned to the
129             current process
130         int startColThisRank = rank * colsPerProcess;
131
132         // Starting column
133         // index for the current process
134
135         // Iterate over the rows of the sparse matrix
136         for (int row = 0; row < sparseMatrix.numRows; ++row)
137         {
138             // Iterate over the columns assigned to the current process
139             for (int col = 0; col < numColsThisRank; ++col)
140             {
141                 finalResult[row][startColThisRank + col] = gatheredResults[
142                     resultIndex++]; // Reconstruct the final result matrix

```

```

124         }
125     }
126 }
127
128 // Return the final result
129 return (worldRank == 0) ? finalResult : FatVector{};
130 }
131

```

Appendix B.E Non-Zero Element Parallelism

Parallel algorithm for multiplying a sparse matrix by a fat vector using non-zero element parallelism.

B.E.1 Declaration File

```

1  #ifndef SPARSEMATRIXFATVECTORMULTIPLYNONZEROELEMENT_H
2  #define SPARSEMATRIXFATVECTORMULTIPLYNONZEROELEMENT_H
3
4  #include "MatrixDefinitions.h"
5  #include <iostream> // std::cout
6
7  /**
8   * @brief Function to execute the sparse matrix-Fat Vector multiplication using non
9   *         -zero element parallel algorithm
10  *
11  * @param sparseMatrix Sparse matrix
12  * @param fatVector Fat Vector
13  * @param vecCols Number of columns in the Fat Vector
14  * @return FatVector Result of the multiplication
15  */
16 FatVector sparseMatrixFatVectorMultiplyNonZeroElement(const SparseMatrix &
17     sparseMatrix, const FatVector &fatVector, int vecCols);
18
19 #endif

```

B.E.2 Implementation File

```

1  #include "SparseMatrixFatVectorMultiplyNonZeroElement.h"
2  #include <mpi.h>
3
4  /**
5   * @brief Function to execute the sparse matrix-Fat Vector multiplication using non
6   *         -zero element parallel algorithm
7   *
8   * @param sparseMatrix Sparse matrix
9   * @param fatVector Fat Vector
10  * @param vecCols Number of columns in the Fat Vector
11  * @return FatVector Result of the multiplication
12  */
13 FatVector sparseMatrixFatVectorMultiplyNonZeroElement(const SparseMatrix &
14     sparseMatrix, const FatVector &fatVector, int vecCols)
15 {
16     // Retrieve the rank and size of the MPI world
17     int worldSize, worldRank;
18     MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
19     MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);
20
21     // ===== FOR DEBUGGING ONLY - START LOCAL COMPUTATION
22     TIMER =====

```

```

20 // double computation_start = MPI_Wtime();
21 // ===== FOR DEBUGGING ONLY - START LOCAL COMPUTATION
    TIMER =====
22
23 // Distribute non-zero elements among processes
24 int totalNonZeroElements = sparseMatrix.values.size(); // Total number of
    non-zero elements
25 int elementsPerProcess = totalNonZeroElements / worldSize; // Number of non-
    zero elements per process
26 int extraElements = totalNonZeroElements % worldSize; // Number of extra
    non-zero elements to be distributed among processes
27 int startIdx, endIdx; // Starting and
    ending indices of the non-zero elements for the current process
28
29 // Determine the starting and ending indices of the non-zero elements for the
    current process
30 if (worldRank < extraElements)
31 {
32     startIdx = worldRank * (elementsPerProcess + 1); // Add 1 to account for
        the extra non-zero elements
33     endIdx = startIdx + elementsPerProcess + 1; // Add 1 to account for
        the extra non-zero elements
34 }
35 else
36 {
37     startIdx = worldRank * elementsPerProcess + extraElements; // Add
        extraElements to account for the extra non-zero elements
38     endIdx = startIdx + elementsPerProcess; // Add
        extraElements to account for the extra non-zero elements
39 }
40
41 // Map the indices of the non-zero elements to their corresponding row indices
42 std::vector<int> rowIndexMap(sparseMatrix.values.size());
43 // Iterate over the rows of the sparse matrix
44 for (int row = 0, idx = 0; row < sparseMatrix.rowPtr.size() - 1; ++row)
45 {
46     // Iterate over the non-zero elements in the current row
47     for (; idx < sparseMatrix.rowPtr[row + 1]; ++idx)
48     {
49         rowIndexMap[idx] = row; // Map the index of the non-zero element to its
            corresponding row index
50     }
51 }
52
53 // Local computation
54 std::vector<double> localResult(sparseMatrix.numRows * vecCols, 0.0);
55 // Iterate over the non-zero elements assigned to the current process
56 for (int idx = startIdx; idx < endIdx; ++idx)
57 {
58     int row = rowIndexMap[idx]; // Row index of the non-zero
        element
59     int col = sparseMatrix.colIndices[idx]; // Column index of the non-zero
        element
60     double value = sparseMatrix.values[idx]; // Value of the non-zero element
61
62     // Iterate over the columns of the Fat Vector
63     for (int k = 0; k < vecCols; ++k)
64     {
65         localResult[row * vecCols + k] += value * fatVector[col][k]; // Compute
            the result
66     }
67 }
68
69 // ===== FOR DEBUGGING ONLY - STOP LOCAL COMPUTATION
    TIMER =====
70 // double computation_end = MPI_Wtime();
71 // double local_computation_time = computation_end - computation_start;
72 // ===== FOR DEBUGGING ONLY - STOP LOCAL COMPUTATION
    TIMER =====
73

```

```

74 // ===== FOR DEBUGGING ONLY - START COMMUNICATION TIMER
75 // =====
76 // FOR DEBUGGING ONLY - START COMMUNICATION TIMER
77 // double communication_start = MPI_Wtime();
78 // ===== FOR DEBUGGING ONLY - START COMMUNICATION TIMER
79 // =====
80 // Initialise the final result only in the root process
81 FatVector finalResult;
82 if (worldRank == 0)
83 {
84     finalResult.resize(sparseMatrix.numRows, std::vector<double>(vecCols, 0.0))
85     ;
86 }
87 // Gather the local results in the root process
88 std::vector<double> flatFinalResult(sparseMatrix.numRows * vecCols, 0.0);
89 // Flat vector to
90 hold the final result
91 MPI_Reduce(localResult.data(), flatFinalResult.data(), sparseMatrix.numRows *
92 vecCols, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD); // Gather the local
93 results in the root process
94
95 // ===== FOR DEBUGGING ONLY - STOP COMMUNICATION TIMER
96 // =====
97 // double communication_end = MPI_Wtime();
98 // double local_communication_time = communication_end - communication_start;
99 // ===== FOR DEBUGGING ONLY - STOP COMMUNICATION TIMER
100 // =====
101 // ===== FOR DEBUGGING ONLY - COLLECTING AND ANALYSING
102 // PERFORMANCE DATA =====
103 // double total_computation_time = 0.0, total_communication_time = 0.0;
104 // MPI_Reduce(&local_computation_time, &total_computation_time, 1, MPI_DOUBLE,
105 // MPI_SUM, 0, MPI_COMM_WORLD);
106 // MPI_Reduce(&local_communication_time, &total_communication_time, 1,
107 // MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
108 // ===== FOR DEBUGGING ONLY - COLLECTING AND ANALYSING
109 // PERFORMANCE DATA =====
110
111 // Reconstruct the finalResult from flatFinalResult in the root process
112 if (worldRank == 0)
113 {
114     // ===== FOR DEBUGGING ONLY - PRINTING PERFORMANCE
115     // DATA =====
116     // double avg_computation_time = total_computation_time / worldSize;
117     // double avg_communication_time = total_communication_time / worldSize;
118     // std::cout << "Non-zero elements Average Computation Time: " <<
119     // avg_computation_time << std::endl;
120     // std::cout << "Non-zero elements Average Communication Time: " <<
121     // avg_communication_time << std::endl;
122     // ===== FOR DEBUGGING ONLY - PRINTING PERFORMANCE
123     // DATA =====
124
125     // Iterate over the rows of the final result
126     for (int i = 0; i < sparseMatrix.numRows; ++i)
127     {
128         std::copy(flatFinalResult.begin() + i * vecCols, flatFinalResult.begin
129             () + (i + 1) * vecCols, finalResult[i].begin()); // Copy the row of
130         the final result
131     }
132 }
133
134 // Return the final result
135 return (worldRank == 0) ? finalResult : FatVector{};
136 }

```

Appendix B.F Utility Functions

Utility functions used by the main file.

B.F.1 Declaration File

```

1  #ifndef UTILS_H
2  #define UTILS_H
3
4  #include <iostream> // std::cout
5  #include <vector>   // std::vector
6  #include <cstdlib>  // rand() and srand()
7  #include <ctime>    // time()
8  #include <mpi.h>
9  #include <petsc.h>
10 #include <fstream>   // std::ifstream
11 #include <string>    // std::string
12 #include <sstream>   // std::stringstream
13 #include <utility>   // std::pair
14 #include <algorithm> // std::sort
15 #include <stdexcept> // std::runtime_error
16 #include <cmath>     // std::fabs
17 #include "MatrixDefinitions.h"
18
19 /**
20  * Method to convert a PETSc matrix to a fat vector
21  * @param C PETSc matrix
22  * @return FatVector fat vector
23  */
24 FatVector ConvertPETScMatToFatVector(Mat C);
25
26 /**
27  * Method to compare two matrices
28  * @param mat1 First matrix
29  * @param mat2 Second matrix
30  * @param tolerance Tolerance for comparison
31  * @return bool True if the matrices are equal, false otherwise
32  */
33 bool areMatricesEqual(const FatVector &mat1, const FatVector &mat2, double
    tolerance);
34
35 /**
36  * Method to read a matrix from a Matrix Market file
37  * @param filename Name of the file
38  * @return SparseMatrix Sparse matrix
39  */
40 SparseMatrix readMatrixMarketFile(const std::string &filename);
41
42 /**
43  * Method to generate a random fat vector
44  * @param n Number of rows
45  * @param m Number of columns
46  * @return FatVector fat vector
47  */
48 FatVector generateLargeFatVector(int n, int k);
49
50 /**
51  * @brief Method to serialize a FatVector to a flat array
52  * @param denseVec fat vector to serialize
53  * @return std::vector<double> Flat array containing the serialized data
54  */
55 std::vector<double> serialize(const FatVector &denseVec);
56
57 /**
58  * @brief Method to deserialize a flat array to a FatVector
59  * @param flat Flat array to deserialize
60  * @param rows Number of rows in the fat vector

```

```

61  * @param cols Number of columns in the fat vector
62  * @return FatVector fat vector
63  */
64  FatVector deserialize(const std::vector<double> &flat, int rows, int cols);
65
66  #endif

```

B.F.2 Implementation File

```

1  #include "utils.h"
2
3  /**
4   * Method to convert a PETSc matrix to a fat vector
5   * @param C PETSc matrix
6   * @return FatVector fat vector
7   */
8  FatVector ConvertPETScMatToFatVector(Mat C)
9  {
10     PetscInt m, n;           // Number of rows and columns in the matrix
11     MatGetSize(C, &m, &n); // Get the number of rows and columns in the matrix
12
13     FatVector denseVec(m, std::vector<double>(n, 0.0)); // fat vector to hold the
14     matrix
15
16     // Iterate over the rows of the matrix
17     for (int i = 0; i < m; ++i)
18     {
19         // Iterate over the columns of the matrix
20         for (int j = 0; j < n; ++j)
21         {
22             PetscScalar value;           // Value of the element
23             MatGetValues(C, i, j, &value); // Get the value of the element
24             denseVec[i][j] = PetscRealPart(value); // Copy the value of the element
25         }
26     }
27
28     // Return the fat vector
29     return denseVec;
30 }
31
32 /**
33  * Method to compare two matrices
34  * @param mat1 First matrix
35  * @param mat2 Second matrix
36  * @param tolerance Tolerance for comparison
37  * @return bool True if the matrices are equal, false otherwise
38  */
39 bool areMatricesEqual(const FatVector &mat1, const FatVector &mat2, double
40 tolerance)
41 {
42     // Check if the matrices have the same dimensions
43     if (mat1.size() != mat2.size())
44         return false;
45
46     // Iterate over the rows of the matrices
47     for (size_t i = 0; i < mat1.size(); ++i)
48     {
49         // Check if the rows have the same dimensions
50         if (mat1[i].size() != mat2[i].size())
51             return false;
52
53         // Iterate over the columns of the matrices
54         for (size_t j = 0; j < mat1[i].size(); ++j)
55         {
56             // Check if the elements are equal
57             if (std::fabs(mat1[i][j] - mat2[i][j]) > tolerance)
58                 return false;
59         }
60     }
61     return true;
62 }

```

```

57         return false; // Matrices are not equal
58     }
59 }
60 }
61
62     return true; // Matrices are equal
63 }
64
65 /**
66  * Method to read a matrix from a Matrix Market file
67  * @param filename Name of the file
68  * @return SparseMatrix Sparse matrix
69  */
70 SparseMatrix readMatrixMarketFile(const std::string &filename)
71 {
72     std::ifstream file(filename); // Input file stream
73
74     // Check if the file was opened successfully
75     if (!file.is_open())
76     {
77         throw std::runtime_error("Unable to open file: " + filename);
78     }
79
80     std::string line; // String to hold the current line
81     bool isSymmetric = false, isPattern = false; // Flags to indicate if the matrix
82     // is symmetric or pattern only
83
84     // Skip the comments
85     while (std::getline(file, line))
86     {
87         // Check if the line is a comment
88         if (line[0] == '%')
89         {
90             // Check if the line contains the word "symmetric"
91             if (line.find("symmetric") != std::string::npos)
92             {
93                 isSymmetric = true; // Set the symmetric flag
94             }
95
96             // Check if the line contains the word "pattern"
97             if (line.find("pattern") != std::string::npos)
98             {
99                 isPattern = true; // Set the pattern flag
100             }
101         }
102         else
103         {
104             break; // First non-comment line reached, break out of the loop
105         }
106     }
107
108     // Read the matrix dimensions
109     int numRows, numCols, nonZeros; // Number of rows,
110     // columns and non-zero elements in the matrix
111     std::stringstream(line) >> numRows >> numCols >> nonZeros; // Read the
112     // dimensions from the line
113
114     // Check if the file was read successfully
115     if (!file)
116     {
117         throw std::runtime_error("Failed to read matrix dimensions from file: " +
118             filename);
119     }
120
121     SparseMatrix matrix; // Sparse
122     // matrix to hold the data
123     matrix.rowPtr.resize(numRows + 1, 0); // Resize
124     // the row pointer vector
125     std::vector<std::vector<std::pair<int, double>>> tempRows(numRows); //
126     // Temporary vector to hold the data

```



```

120     int rowIndex, colIndex;                                // Row and
        column indices
121     double value;                                          // Value of
        the non-zero element
122
123     // Read the non-zero elements
124     for (int i = 0; i < nonZeros; ++i)
125     {
126         // If the matrix is pattern only, the value of the non-zero element is 1.0
127         if (isPattern)
128         {
129             file >> rowIndex >> colIndex; // Read the row and column indices
130             value = 1.0;                  // Default value for pattern entries
131         }
132         else
133         {
134             file >> rowIndex >> colIndex >> value; // Read the row and column
                indices and the value
135         }
136
137         // Check if the file was read successfully
138         if (!file)
139         {
140             throw std::runtime_error("Failed to read data from file: " + filename);
141         }
142
143         rowIndex--; // Adjusting from 1-based to 0-based indexing
144         colIndex--; // Adjusting from 1-based to 0-based indexing
145
146         tempRows[rowIndex].emplace_back(colIndex, value); // Store the data in the
                temporary vector
147
148         // If the matrix is symmetric, store the data in the transpose as well
149         if (isSymmetric && rowIndex != colIndex)
150         {
151             tempRows[colIndex].emplace_back(rowIndex, value); // Store the data in
                the temporary vector
152         }
153     }
154
155     // Sort each row by column index
156     for (auto &row : tempRows)
157     {
158         std::sort(row.begin(), row.end());
159     }
160
161     // Reconstruct SparseMatrix structure
162     int cumSum = 0; // Cumulative sum of the number of non-zero elements
163
164     // Iterate over the rows of the matrix
165     for (int i = 0; i < numRows; ++i)
166     {
167         matrix.rowPtr[i] = cumSum; // Store the cumulative sum in the row pointer
                vector
168
169         // Iterate over the non-zero elements in the current row
170         for (const auto &elem : tempRows[i])
171         {
172             matrix.values.push_back(elem.second); // Store the value of the non-
                zero element
173             matrix.colIndices.push_back(elem.first); // Store the column index of
                the non-zero element
174         }
175
176         cumSum += tempRows[i].size(); // Update the cumulative sum
177     }
178
179     matrix.rowPtr[numRows] = cumSum; // Store the cumulative sum in the row pointer
        vector
180     matrix.numRows = numRows;        // Store the number of rows

```

```

181     matrix.numCols = numCols;           // Store the number of columns
182
183     // Return the sparse matrix
184     return matrix;
185 }
186
187 /**
188  * Method to generate a random fat vector
189  * @param n Number of rows
190  * @param m Number of columns
191  * @return FatVector fat vector
192  */
193 FatVector generateLargeFatVector(int n, int k)
194 {
195     FatVector denseVector(n, std::vector<double>(k)); // fat vector to hold the
196     random values
197
198     // Iterate over the rows of the fat vector
199     for (int i = 0; i < n; ++i)
200     {
201         // Iterate over the columns of the fat vector
202         for (int j = 0; j < k; ++j)
203         {
204             denseVector[i][j] = rand() % 100 + 1; // Generate a random value
205             between 1 and 100
206         }
207     }
208
209     // Return the fat vector
210     return denseVector;
211 }
212
213 /**
214  * @brief Method to serialize a FatVector to a flat array
215  * @param denseVec fat vector to serialize
216  * @return std::vector<double> Flat array containing the serialized data
217  */
218 std::vector<double> serialize(const FatVector &denseVec)
219 {
220     std::vector<double> flat; // Flat array to hold the serialized data
221
222     // Iterate over the rows of the fat vector
223     for (const auto &vec : denseVec)
224     {
225         flat.insert(flat.end(), vec.begin(), vec.end()); // Copy the elements
226     }
227
228     // Return the flat array
229     return flat;
230 }
231
232 /**
233  * @brief Method to deserialize a flat array to a FatVector
234  * @param flat Flat array to deserialize
235  * @param rows Number of rows in the fat vector
236  * @param cols Number of columns in the fat vector
237  * @return FatVector fat vector
238  */
239 FatVector deserialize(const std::vector<double> &flat, int rows, int cols)
240 {
241     FatVector denseVec(rows, std::vector<double>(cols)); // fat vector to hold the
242     deserialized data
243
244     // Iterate over the rows of the fat vector
245     for (int i = 0; i < rows; ++i)
246     {
247         // Iterate over the columns of the fat vector
248         for (int j = 0; j < cols; ++j)
249         {
250             denseVec[i][j] = flat[i * cols + j]; // Copy the element

```

```

248     }
249 }
250
251 // Return the fat vector
252 return denseVec;
253 }

```

Appendix B.G Main File

Main file for running the different algorithms and comparing their performance.

```

1  #include "utils.h" // Utility functions
2  #include "SparseMatrixFatVectorMultiply.h" // Sequential algorithm
3  #include "SparseMatrixFatVectorMultiplyRowWise.h" // Parallel algorithm (row
   -wise)
4  #include "SparseMatrixFatVectorMultiplyColumnWise.h" // Parallel algorithm (
   column-wise)
5  #include "SparseMatrixFatVectorMultiplyNonZeroElement.h" // Parallel algorithm (non
   -zero element)
6
7  int main(int argc, char *argv[])
8  {
9      //
10
11     // ===== INITIALISATION
12     // =====
13
14     // Initialise MPI and PETSc
15     MPI_Init(&argc, &argv);
16     PetscInitialize(&argc, &argv, NULL, NULL);
17
18     // Retrieve the rank and size of the world communicator
19     int worldRank, worldSize;
20     MPI_Comm_rank(PETSC_COMM_WORLD, &worldRank);
21     MPI_Comm_size(PETSC_COMM_WORLD, &worldSize);
22
23     // Check if the correct number of arguments is provided
24     if (argc != 3)
25     {
26         if (worldRank == 0)
27         {
28             std::cerr << "Usage: " << argv[0] << " <number of columns> <matrix file
               path>" << std::endl;
29         }
30         MPI_Abort(PETSC_COMM_WORLD, 1);
31     }
32
33     // Parse the command-line arguments
34     int k = std::atoi(argv[1]); // Convert the first argument to an integer
35     std::string filename = argv[2]; // The second argument is the filename
36
37     // Declare the sparse matrix and Fat Vector
38     SparseMatrix M;
39     FatVector v;
40
41     // Declare the result of the serial multiplication
42     FatVector resultSerial;
43
44     // Declare the data for broadcasting the sparse matrix and Fat Vector
45     std::vector<double> flatData;
46     int dataSize = 0;
47
48     // Declare the variables for timing the execution of the algorithms

```

```

48     double startTime, endTime;
49
50     //
51     // ===== READ THE SPARSE MATRIX AND GENERATE THE Fat
52     // Vector =====
53
54     if (worldRank == 0)
55     {
56         std::cout << "World size: " << worldSize << std::endl;    // Print the
57         // number of processes
58         std::cout << "Sparse matrix: " << filename << std::endl; // Print the name
59         // of the Matrix Market file
60
61         // Read the sparse matrix from the Matrix Market file
62         M = readMatrixMarketFile(filename);
63         std::cout << "Matrix size: " << M.numRows << "x" << M.numCols << std::endl;
64
65         // Generate a random Fat Vector
66         v = generateLargeFatVector(M.numCols, k);
67         std::cout << "Vector size: " << M.numCols << "x" << k << std::endl;
68
69         // Prepare the data for broadcasting
70         flatData = serialize(v);    // Serialize the Fat Vector
71         dataSize = flatData.size(); // Size of the serialized data
72     }
73
74     //
75     // ===== EXECUTE THE SERIAL MULTIPLICATION
76     // =====
77
78     if (worldRank == 0)
79     {
80         // Execute the serial multiplication
81         startTime = MPI_Wtime();
82         resultSerial = sparseMatrixFatVectorMultiply(M, v, k);
83         endTime = MPI_Wtime();
84         std::cout << "Serial Algo Execution time: " << (endTime - startTime)
85         << std::endl;
86
87         // FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS OF THE RESULT
88         // std::cout << "Result: " << std::endl;
89         // for (int i = 0; i < 10; ++i)
90         // {
91         //     for (int j = 0; j < k; ++j)
92         //     {
93         //         std::cout << resultSerial[i][j] << " ";
94         //     }
95         //     std::cout << std::endl;
96         // }
97     }
98
99     //
100    // ===== BROADCAST THE SPARSE MATRIX AND Fat Vector
101    // =====
102
103    // Wait for the main process to finish the serial multiplication
104    MPI_Barrier(MPI_COMM_WORLD);

```

```

101 // ===== FOR DEBUGGING ONLY - START BROADCAST TIMER
102 // =====
103 // startTime = MPI_Wtime();
104 // ===== FOR DEBUGGING ONLY - START BROADCAST TIMER
105 // =====
106 // Broadcast the Sparse Matrix to all processes
107 // Prepare the data for broadcasting
108 int valuesSize = M.values.size(); // Number of non-
109     zero elements
110 int colIndicesSize = M.colIndices.size(); // Number of column
111     indices
112 int rowPtrSize = M.rowPtr.size(); // Number of row
113     pointers
114 MPI_Bcast(&M.numRows, 1, MPI_INT, 0, MPI_COMM_WORLD); // Broadcast the
115     number of rows
116 MPI_Bcast(&M.numCols, 1, MPI_INT, 0, MPI_COMM_WORLD); // Broadcast the
117     number of columns
118 MPI_Bcast(&valuesSize, 1, MPI_INT, 0, MPI_COMM_WORLD); // Broadcast the
119     number of non-zero elements
120 MPI_Bcast(&colIndicesSize, 1, MPI_INT, 0, MPI_COMM_WORLD); // Broadcast the
121     number of column indices
122 MPI_Bcast(&rowPtrSize, 1, MPI_INT, 0, MPI_COMM_WORLD); // Broadcast the
123     number of row pointers
124 // Resize the vectors for all processes
125 if (worldRank != 0)
126 {
127     M.values.resize(valuesSize);
128     M.colIndices.resize(colIndicesSize);
129     M.rowPtr.resize(rowPtrSize);
130 }
131 // Broadcast the data
132 MPI_Bcast(M.values.data(), valuesSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);
133 MPI_Bcast(M.colIndices.data(), colIndicesSize, MPI_INT, 0, MPI_COMM_WORLD);
134 MPI_Bcast(M.rowPtr.data(), rowPtrSize, MPI_INT, 0, MPI_COMM_WORLD);
135
136 // Broadcast the Fat Vector to all processes
137 // Broadcast the size of the serialized data
138 MPI_Bcast(&dataSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
139 // Resize flatData for all processes
140 if (worldRank != 0)
141 {
142     flatData.resize(dataSize);
143 }
144 // Broadcast the data
145 MPI_Bcast(flatData.data(), dataSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);
146 // Deserialize the data
147 if (worldRank != 0)
148 {
149     v.resize(M.numCols, std::vector<double>(k));
150     v = deserialize(flatData, M.numCols, k);
151 }
152
153 // Wait for all processes to finish the broadcast
154 MPI_Barrier(MPI_COMM_WORLD);
155
156 // ===== FOR DEBUGGING ONLY - STOP BROADCAST TIMER
157 // =====
158 // endTime = MPI_Wtime();
159 // if (worldRank == 0)
160 // {
161 //     std::cout << "Broadcast time: " << (endTime - startTime) << std::endl;
162 // }
163 // ===== FOR DEBUGGING ONLY - STOP BROADCAST TIMER
164 // =====
165 //
166 -----

```

```

157 // ===== EXECUTE THE PARALLEL MULTIPLICATION (ROW-WISE)
158 // -----
159
160 // Execute the parallel multiplication (row-wise)
161 startTime = MPI_Wtime();
162 FatVector resultRowWise = sparseMatrixFatVectorMultiplyRowWise(M, v, k);
163 endTime = MPI_Wtime();
164
165 // Only the main process prints the parallel execution time
166 if (worldRank == 0)
167 {
168     std::cout << "Row-wise Execution time: " << (endTime - startTime)
169     << std::endl;
170
171     // ===== FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS
172     // =====
173     // std::cout << "Result: " << std::endl;
174     // for (int i = 0; i < 10; ++i)
175     // {
176     //     for (int j = 0; j < k; ++j)
177     //     {
178     //         std::cout << resultRowWise[i][j] << " ";
179     //     }
180     //     std::cout << std::endl;
181     // }
182     // ===== FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS
183     // =====
184
185     // Compare the results of the serial and parallel multiplications
186     if (areMatricesEqual(resultSerial, resultRowWise, 1e-6)) // Tolerance = 1e
187     -6
188     {
189         std::cout << "Row-wise: Results are the same!"
190         << std::endl;
191     }
192     else
193     {
194         std::cout << "Row-wise: Results are different!"
195         << std::endl;
196     }
197 }
198
199 // -----
200
201 // ===== EXECUTE THE PARALLEL MULTIPLICATION (COLUMN-WISE)
202 // =====
203
204 // Wait for all processes to finish the parallel multiplication (row-wise)
205 MPI_Barrier(MPI_COMM_WORLD);
206
207 // Execute the parallel multiplication (column-wise)
208 startTime = MPI_Wtime();
209 FatVector resultColumnWise = sparseMatrixFatVectorMultiplyColumnWise(M, v, k);
210 endTime = MPI_Wtime();
211
212 // Only the main process prints the parallel execution time
213 if (worldRank == 0)
214 {
215     std::cout << "Column-wise Execution time: " << (endTime - startTime)
216     << std::endl;
217
218     // ===== FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS
219     // =====

```

```

215 // std::cout << "Result: " << std::endl;
216 // for (int i = 0; i < 10; ++i)
217 // {
218 //     for (int j = 0; j < k; ++j)
219 //     {
220 //         std::cout << resultColumnWise[i][j] << " ";
221 //     }
222 //     std::cout << std::endl;
223 // }
224 // ===== FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS
225 // =====
226 // Compare the results of the serial and parallel multiplications
227 if (areMatricesEqual(resultSerial, resultColumnWise, 1e-6)) // Tolerance =
228     1e-6
229 {
230     std::cout << "Column-wise: Results are the same!"
231     << std::endl;
232 }
233 else
234 {
235     std::cout << "Column-wise: Results are different!"
236     << std::endl;
237 }
238 //
239 // -----
240 // ===== EXECUTE THE PARALLEL MULTIPLICATION (NON-ZERO
241 // ELEMENT) =====
242 // -----
243 // Wait for all processes to finish the parallel multiplication (column-wise)
244 MPI_Barrier(MPI_COMM_WORLD);
245 // Execute the parallel multiplication (non-zero element)
246 startTime = MPI_Wtime();
247 FatVector resultNonZeroElement = sparseMatrixFatVectorMultiplyNonZeroElement(M,
248     v, k);
249 endTime = MPI_Wtime();
250 // Only the main process prints the parallel execution time
251 if (worldRank == 0)
252 {
253     std::cout << "Non-zero Elements Execution time: " << (endTime - startTime)
254     << std::endl;
255 // ===== FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS
256 // =====
257 // std::cout << "Result: " << std::endl;
258 // for (int i = 0; i < 10; ++i)
259 // {
260 //     for (int j = 0; j < k; ++j)
261 //     {
262 //         std::cout << resultNonZeroElement[i][j] << " ";
263 //     }
264 //     std::cout << std::endl;
265 // }
266 // ===== FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS
267 // =====
268 // Compare the results of the serial and parallel multiplications
269 if (areMatricesEqual(resultSerial, resultNonZeroElement, 1e-6)) //
270     Tolerance = 1e-6
271 {
272     std::cout << "Non-zero Elements: Results are the same!"
273     << std::endl;

```

```

274     }
275     else
276     {
277         std::cout << "Non-zero Elements: Results are different!"
278                 << std::endl;
279     }
280 }
281
282 //
283 // ===== EXECUTE THE PARALLEL MULTIPLICATION (
284 // PETSc) =====
285 // =====
286 // Wait for all processes to finish the parallel multiplication (non-zero
287 // element)
288 MPI_Barrier(MPI_COMM_WORLD);
289 // Declare the PETSc matrix
290 Mat A, B, C;
291
292 // ===== FOR DEBUGGING ONLY - START PETSCS SETUP TIMER
293 // =====
294 // startTime = MPI_Wtime();
295 // ===== FOR DEBUGGING ONLY - START PETSCS SETUP TIMER
296 // =====
297 // Create a parallel matrix to store the sparse matrix
298 MatCreate(PETSC_COMM_WORLD, &A);
299 MatSetSizes(A, PETSC_DECIDE, PETSC_DECIDE, M.numRows, M.numCols);
300 MatSetType(A, MATMPIAIJ);
301 MatSetUp(A);
302 // Fill the PETSc matrix with the values from the sparse matrix
303 if (worldRank == 0)
304 {
305     for (int i = 0; i < M.numRows; ++i)
306     {
307         for (int j = M.rowPtr[i]; j < M.rowPtr[i + 1]; ++j)
308         {
309             MatSetValue(A, i, M.colIndices[j], M.values[j], INSERT_VALUES);
310         }
311     }
312 // Assemble the PETSc matrix
313 MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
314 MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
315
316 // Create a parallel matrix to store the Fat Vector
317 MatCreate(PETSC_COMM_WORLD, &B);
318 MatSetSizes(B, PETSC_DECIDE, PETSC_DECIDE, M.numCols, k);
319 MatSetType(B, MATDENSE);
320 MatSetUp(B);
321 // Fill the PETSc matrix B with values from the dense matrix v
322 if (worldRank == 0)
323 {
324     for (int i = 0; i < M.numCols; ++i)
325     {
326         for (int j = 0; j < k; ++j)
327         {
328             MatSetValue(B, i, j, v[i][j], INSERT_VALUES);
329         }
330     }
331 }
332 // Assemble the PETSc matrix
333 MatAssemblyBegin(B, MAT_FINAL_ASSEMBLY);
334 MatAssemblyEnd(B, MAT_FINAL_ASSEMBLY);
335

```



```

336 // ===== FOR DEBUGGING ONLY - STOP PETSCS SETUP TIMER
337 // =====
338 // endTime = MPI_Wtime();
339 // if (worldRank == 0)
340 // {
341 //     std::cout << "PETSc Setup time: " << (endTime - startTime) << std::endl;
342 // }
343 // ===== FOR DEBUGGING ONLY - STOP PETSCS SETUP TIMER
344 // =====
345 // Create a parallel matrix to store the result of the multiplication
346 startTime = MPI_Wtime();
347 MatProductCreate(A, B, NULL, &C);
348 MatMatMult(A, B, MAT_INITIAL_MATRIX, PETSC_DEFAULT, &C);
349 endTime = MPI_Wtime();
350 if (worldRank == 0)
351 {
352     // Print the execution time
353     std::cout << "PETSc Execution time: " << (endTime - startTime) << std::endl;
354 }
355 // ===== FOR DEBUGGING ONLY - START PETSCS CONVERSION
356 // TIMER =====
357 // startTime = MPI_Wtime();
358 // ===== FOR DEBUGGING ONLY - START PETSCS CONVERSION
359 // TIMER =====
360 // Create a sequential matrix to retrieve the result
361 Mat CSeq;
362 MatCreateRedundantMatrix(C, worldSize, MPI_COMM_NULL, MAT_INITIAL_MATRIX, &CSeq);
363
364 if (worldRank == 0)
365 {
366     // Convert the result matrix C to a FatVector
367     FatVector globalMatrix = ConvertPETScMatToFatVector(CSeq);
368
369     // ===== FOR DEBUGGING ONLY - STOP PETSCS CONVERSION
370     // TIMER =====
371     // endTime = MPI_Wtime();
372     // std::cout << "PETSc Conversion time: " << (endTime - startTime) << std::endl;
373     // ===== FOR DEBUGGING ONLY - STOP PETSCS CONVERSION
374     // TIMER =====
375
376     // ===== FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS
377     // =====
378     // std::cout << "Result: " << std::endl;
379     // for (int i = 0; i < 10; ++i)
380     // {
381     //     for (int j = 0; j < k; ++j)
382     //     {
383     //         std::cout << globalMatrix[i][j] << " ";
384     //     }
385     //     std::cout << std::endl;
386     // }
387     // ===== FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS
388     // =====
389
390     // Compare the results of the serial and PETSc multiplications
391     if (areMatricesEqual(resultSerial, globalMatrix, 1e-6)) // Tolerance = 1e-6
392     {
393         std::cout << "PETSc: Results are the same!"
394         << std::endl;
395     }
396     else
397     {
398         std::cout << "PETSc: Results are different!"
399         << std::endl;
400     }
401 }

```

```

395     }
396 }
397
398 // Free the memory
399 MatDestroy(&A);
400 MatDestroy(&B);
401 MatDestroy(&C);
402 MatDestroy(&CSeq);
403
404 // Finalise MPI and PETSc
405 PetscFinalize();
406 MPI_Finalize();
407
408 return 0;
409 }

```

Appendix B.H Scripts

B.H.1 MPI Submission Script

Bash script to submit an MPI job to the cluster.

```

#!/bin/bash
##
## MPI submission script for PBS on CR2
## -----
##
## "MPI-sub2022v1"
## Follow the 6 steps below to configure your job
##
## STEP 1:
##
## Enter a job name after the -N on the line below:
##
##PBS -N mpi_assessment_test_8_cores_16_425500
##
## STEP 2:
##
## Select the number of cpus/cores required by modifying the #PBS -l select line
## below
##
## Normally you select cpus in chunks of 16 cpus
## The Maximum value for ncpus is 16 and mpirprocs MUST be the same value as ncpus.
##
## If more than 16 cpus are required then select multiple chunks of 16
## e.g. 16 CPUs: select=1:ncpus=16:mpiprocs=16
##      32 CPUs: select=2:ncpus=16:mpiprocs=16
##      ..etc..
##
##PBS -l select=2:ncpus=16:mpiprocs=16
##
## STEP 3:
##
## Select the correct queue by modifying the #PBS -q line below
##
## half_hour      - 30 minutes
## one_hour       - 1 hour
## three_hour     - 3 hours
## six_hour       - 6 hours
## half_day       - 12 hours
## one_day        - 24 hours
## two_day        - 48 hours
## five_day       - 120 hours
## ten_day        - 240 hours (by special arrangement)
##
##PBS -q half_hour

```

```

##
## STEP 4:
##
## Replace the hpc@cranfield.ac.uk email address
## with your Cranfield email address on the #PBS -M line below:
## Your email address is NOT your username
##
#PBS -m abe
#PBS -M alexis.balayre.500@cranfield.ac.uk
##
## =====
## DO NOT CHANGE THE LINES BETWEEN HERE
## =====
#PBS -j oe
#PBS -W sandbox=PRIVATE
#PBS -k n
ln -s $PWD $PBS_O_WORKDIR/$PBS_JOBID
## Change to working directory
cd $PBS_O_WORKDIR
## Calculate number of CPUs
export cpus='cat $PBS_NODEFILE | wc -l'
sort -u $PBS_NODEFILE -o mpi_nodes.$$
export I_MPI_HYDRA_IFACE=ib0
export I_MPI_HYDRA_BOOTSTRAP=ssh
export I_MPI_HYDRA_RMK=pbs
export K_VALUE=1
export MATRIX_PATH=/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/cop20k_A.
    mtx
## Debug options - only enable when instructed by HPC support
##export I_MPI_HYDRA_DEBUG=1
##export I_MPI_DEBUG=6
##export I_MPI_DEBUG_OUTPUT=%h-%r-%p-debug.out
## set some MPI tuning parameters to use the correct transport
## =====
## AND HERE
## =====
##
## STEP 5:
##
## Load the default application environment
## For a specific version add the version number, e.g.
## module load intel/2016b
##
module use /apps/modules/all
module load intel/2021b
##
## STEP 6:
##
## Run MPI code
##
## The main parameter to modify is your mpi program name
## - change YOUR_EXECUTABLE to your own filename
##
mpirun -genval1 -hostfile mpi_nodes.$$ -np ${cpus} ../my_program_final_debug ${
    K_VALUE} ${MATRIX_PATH}

## Tidy up the log directory
## DO NOT CHANGE THE LINE BELOW
## =====
rm $PBS_O_WORKDIR/$PBS_JOBID
#

```

B.H.2 Batch Test Script

Bash script to submit multiple MPI jobs to the cluster.

```
#!/bin/bash

# Script to submit a batch of jobs to the cluster

# Path to the original script
original_script="mpi.sub"

# Maximum number of cores used for the job
max_cores=96

# Define a set of k values to test (Number of columns in the Fat Vector)
k_values=(1 3 6 9 12)

# Define a set of paths to test
paths=(
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/cop20k_A.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/adder_dcop_32.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/bcsstk17.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/af23560.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/amazon0302.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/cavity10.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/cage4.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/dc1.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/FEM_3D_thermal1.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/mac_econ_fwd500.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/mcfe.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/mhd4800a.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/olafu.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/raefsky2.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/rdist2.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/thermal1.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/thermomech_TK.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/west2021.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/lung2.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/olm1000.mtx"
    "/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/roadNet-PA.mtx"
)

# Loop over the k values
for k_value in "${k_values[@]"; do
    # Loop over the paths of MTX files
    for path in "${paths[@]"; do
        # Loop over the number of chunks
        for chunks in $(seq 1 $((max_cores / 16))); do
            # Loop over the number of cpus per chunk
            for cpus in $(seq 2 16); do
                # Calculate the total number of cores
                total_cores=$((chunks * cpus))
                # Check if the total number of cores is less than the maximum
                number of cores
                if [ $total_cores -le $max_cores ]; then
                    echo "Submitting job with $total_cores cores, $chunks chunks
and $cpus cpus per chunk"
                    echo "Path: $path"

                    # Create a unique job name
                    matrix_name=$(basename "$path") # Remove the path
                    sanitized_matrix_name=${matrix_name//[a-zA-Z0-9_]/_} # Replace
all non-alphanumeric characters with underscores
                    job_name="${sanitized_matrix_name}_k${k_value}_cores${
total_cores}_chunks${chunks}_cpus${cpus}" # Add the k value to the job name

                    # Create a temporary submission script
                    temp_script="temp_${job_name}.sub"
                    cp "$original_script" "$temp_script"
```

```

# Replace the variables in the temporary script
sed -i "s|export k_value=.*|export k_value=${k_value}|" "
$temp_script" # Export the k value
sed -i "s|export MATRIX_PATH=.*|export MATRIX_PATH=${path}|" "
$temp_script" # Export the path to the MTX file
sed -i "s|#PBS -N .*|#PBS -N $job_name|" "$temp_script" # Set
the job name
sed -i "s|#PBS -l select=.*|#PBS -l select=${chunks}:ncpus=
$cpus:mpiprocs=$cpus|" "$temp_script" # Set the number of chunks and cpus per
chunk

# Submit the job and get the job id
job_id=$(qsub "$temp_script")
echo "Job id: $job_id"

# Wait until the job is finished
while true; do
    # Get the job status and duration
    job_status=$(qstat -f "$job_id" | grep job_state | awk '{
print $3}') # Get the job status
    job_duration=$(qstat -f "$job_id" | grep resources_used.
walltime | awk '{print $3}') # Get the job duration
    job_duration_seconds=$(echo $job_duration | awk -F: '{
print ($1 * 3600) + ($2 * 60) + $3 }') # Convert the job duration to seconds
    echo "Job status: $job_status"
    echo "Job duration: $job_duration"

    # If the job is finished, break the loop
    if [ -z "$job_status" ]; then
        break
    fi

    # if the job is running for more than 4 minutes, cancel it
    if [ "$job_duration_seconds" -gt 240 ]; then
        echo "Job is running for more than 4 minutes.
Cancelling it."
        qdel "$job_id"
        break
    fi

    # Wait for 1 second
    sleep 1
done

# Remove the temporary script
rm "$temp_script"
fi
done
done
done
done

```

B.H.3 Get CSV Script

Bash script to analyse all job results files and extract the relevant information to create a CSV file.

```
#!/bin/bash

# Name of the CSV file to write the data to
output_csv="results.csv"

# Headers for the CSV file
echo "file Name,Cores Number,Sparse Matrix,Fat Vector,Serial Algo Execution time,
Row-wise Average Communication Time,Row-wise Average Computation Time,Row-wise
Execution time,Row-wise Result,Column-wise Average Communication Time,Column-
wise Average Computation Time,Column-wise Execution time,Column-wise Result,Non
-zero elements Average Communication Time,Non-zero elements Average Computation
Time,Non-zero Elements Execution time,Non-zero Elements Result,PETSc Execution
time,PETSc Result" >$output_csv

# Loop over the output files
for file in *.o*; do
    # Check that the file is valid and that it is a result file
    if [[ -s $file && $file == *mtx* ]]; then
        # Extract the job name and the number of cores from the file name
        job_name=$(basename "$file" | sed -e 's/\.([^.]*$)//') # Remove file
        extension
        num_cores=$(echo $file | grep -oP '(?<=_cores)\d+') # Extract the number
        of cores from the file name

        # Extract the matrix size and the vector size from the file
        matrix_size=$(grep "Matrix size" $file | awk '{print $3}' | sed 's/size://')
        ) # Extract the matrix size from the file
        vector_size=$(grep "Vector size" $file | awk '{print $3}' | sed 's/size://')
        ) # Extract the vector size from the file

        # Extract the serial execution time from the file
        serial_time=$(grep "Serial Algo Execution time" $file | awk '{print $5}')

        # Row-wise Data
        row_wise_communication_time=$(grep "Row-wise Average Communication Time"
$file | awk '{print $5}') # Extract the row-wise average communication time
from the file
        row_wise_computation_time=$(grep "Row-wise Average Computation Time" $file
| awk '{print $5}') # Extract the row-wise average computation time from
the file
        row_wise_execution_time=$(grep "Row-wise Execution time" $file | awk '{
print $4}') # Extract the row-wise execution time from the file
        row_wise_result=$(grep "Row-wise: Results are" $file | awk '{print $5}')
        # Extract the row-wise result from the file
        row_wise_result=$(if [ $row_wise_result == "same!" ]; then echo "same";
else echo "different"; fi) # Convert the row-wise result to a boolean

        # Column-wise Data
        col_wise_communication_time=$(grep "Column-wise Average Communication Time"
$file | awk '{print $6}') # Extract the column-wise average communication time
from the file
        col_wise_computation_time=$(grep "Column-wise Average Computation Time"
$file | awk '{print $6}') # Extract the column-wise average computation
time from the file
        col_wise_execution_time=$(grep "Column-wise Execution time" $file | awk '{
print $4}') # Extract the column-wise execution time from the
file
        col_wise_result=$(grep "Column-wise: Results are" $file | awk '{print $5}')
        # Extract the column-wise result from the file
        col_wise_result=$(if [ $col_wise_result == "same!" ]; then echo "same";
else echo "different"; fi) # Convert the column-wise result to a boolean

        # Non-zero element Data
        nonzero_communication_time=$(grep "Non-zero elements Average Communication
Time" $file | awk '{print $6}') # Extract the non-zero elements average
```

```

communication time from the file
    nonzero_computation_time=$(grep "Non-zero elements Average Computation Time
" $file | awk '{print $6}') # Extract the non-zero elements average
computation time from the file
    nonzero_execution_time=$(grep "Non-zero Elements Execution time" $file |
awk '{print $5}') # Extract the non-zero elements execution
time from the file
    nonzero_result=$(grep "Non-zero Elements: Results are" $file | awk '{print
$6}') # Extract the non-zero elements result from the
file
    nonzero_result=$(if [ $nonzero_result == "same!" ]; then echo "same"; else
echo "different"; fi) # Convert the non-zero elements result to a
boolean

# PETSc Data
    petsc_execution_time=$(grep "PETSc Execution time" $file | awk '{print $4}'
) # Extract the PETSc execution time from the file
    petsc_result=$(grep "PETSc: Results are" $file | awk '{print $5}')
# Extract the PETSc result from the file
    petsc_result=$(if [ $petsc_result == "same!" ]; then echo "same"; else echo
"different"; fi) # Convert the PETSc result to a boolean

# Write the extracted data to the CSV file
    echo "$job_name,$num_cores,$matrix_size,$vector_size,$serial_time,
$row_wise_communication_time,$row_wise_computation_time,
$row_wise_execution_time,$row_wise_result,$col_wise_communication_time,
$col_wise_computation_time,$col_wise_execution_time,$col_wise_result,
$nonzero_communication_time,$nonzero_computation_time,$nonzero_execution_time,
$nonzero_result,$petsc_execution_time,$petsc_result" >>$output_csv
fi
done

echo "The data was successfully written in $output_csv"

```