



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

5<sup>th</sup> February 2024

# **Abstract**

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

# Table of Contents

<b>Abstract</b>	<b>ii</b>
<b>Table of Contents</b>	<b>iii</b>
<b>List of Figures</b>	<b>v</b>
<b>List of Tables</b>	<b>vi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Problem Description . . . . .	1
<b>2 Methodology</b>	<b>3</b>
2.1 Data Structures . . . . .	3
2.1.1 Sparse Matrix . . . . .	3
2.1.2 fat vector . . . . .	4
2.2 Sequential Algorithm . . . . .	4
2.2.1 Complexity Analysis . . . . .	4
2.2.1.1 Temporal Complexity . . . . .	4
2.2.1.2 Spatial Complexity . . . . .	5
2.3 Line-Based Parallelism . . . . .	6
2.3.1 Complexity Analysis . . . . .	6
2.3.1.1 Temporal Complexity . . . . .	6
2.3.1.2 Spatial Complexity . . . . .	7
2.4 Column-Wise Parallelism . . . . .	9
2.4.1 Complexity Analysis . . . . .	9
2.4.1.1 Temporal Complexity . . . . .	9
2.4.1.2 Spatial Complexity . . . . .	10
2.5 Non-Zero Element Parallelism . . . . .	12
2.5.1 Complexity Analysis . . . . .	12
2.5.1.1 Temporal Complexity . . . . .	12
2.5.1.2 Spatial Complexity . . . . .	13
<b>3 Results and Discussion</b>	<b>14</b>
3.1 HPC Environmental Impact . . . . .	14
<b>4 Conclusion</b>	<b>15</b>

<b>A</b>	<b>Documentation</b>	<b>16</b>
A.A	Project tree . . . . .	16
A.B	Getting Started . . . . .	16
A.C	Detailed Features of Functions . . . . .	17
<b>B</b>	<b>Source Codes</b>	<b>18</b>
B.A	Data Structures . . . . .	18
B.B	Sequential Algorithm . . . . .	19
	B.B.1 Declaration File . . . . .	19
	B.B.2 Implementation File . . . . .	19
B.C	Line-Based Parallelism . . . . .	20
	B.C.1 Declaration File . . . . .	20
	B.C.2 Implementation File . . . . .	20
B.D	Column-Wise Parallelism . . . . .	23
	B.D.1 Declaration File . . . . .	23
	B.D.2 Implementation File . . . . .	23
B.E	Non-Zero Element Parallelism . . . . .	26
	B.E.1 Declaration File . . . . .	26
	B.E.2 Implementation File . . . . .	26
B.F	Utility Functions . . . . .	29
	B.F.1 Declaration File . . . . .	29
	B.F.2 Implementation File . . . . .	30
B.G	Main File . . . . .	34
B.H	Scripts . . . . .	41
	B.H.1 MPI Submission Script . . . . .	41
	B.H.2 Batch Test Script . . . . .	43
	B.H.3 Get CSV Script . . . . .	45

# List of Figures

# List of Tables

# 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, modelling molecules for new medicines or designing aircraft.

Cranfield University has two HPC systems: CRESCENT2 and DELTA. However, this report will focus exclusively on CRESCENT2. The latter is a HPC cluster designed to provide computing power to the university community for teaching and research. The CRESCENT 2 nodes, which are like individual workstations within the cluster, are equipped with Intel Xeon E5 2620 processors. Each node contains two processors (or ‘sockets’), with 16 cores and as many threads, accompanied by 16 gigabytes of RAM, enabling them to handle large workloads. Communication between the nodes is ensured by a 128 Gb/s InfiniBand EDR network, a high-speed connectivity technology that facilitates the rapid exchange of data, a crucial function for parallel processing of computer tasks.

Sparse matrix-vector multiplication is a fundamental operation in numerical linear algebra and has numerous applications in science and engineering.

### 1.1 Problem Description

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 **DenseVector** structure is defined in Appendix B.A.

## 2.2 Sequential Algorithm

Given a sparse matrix  $M$  in CSR format and a fat vector  $v$ , the product  $M \times v$  is computed as follows:

---

### 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  $(j, \text{value})$  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**  $Result$

---

### 2.2.1 Complexity Analysis

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 multiplication  $M \times v$  can be analyzed as follows:

- For each row  $i$  of  $M$ , the algorithm iterates over its non-zero elements.
- For each non-zero element  $M_{ij}$ , the algorithm performs a multiplication with each element of column  $j$  in  $v$ , resulting in  $k$  multiplications.
- These products are accumulated in the corresponding row of the result matrix, which has dimensions  $m \times k$ .

#### 2.2.1.1 Temporal Complexity

The time complexity depends on the number of non-zero elements in the hollow matrix and the dimension of the fat vector.

1. **Hollow Matrix Row Traversal:** Each row of the matrix is traversed once, so this part has complexity  $O(m)$ .

2. **Access to non-zero elements:** For each row, the algorithm traverses the non-zero elements, therefore the complexity of traversing all the non-zero elements in the matrix is  $O(z)$ .
3. **Multiplication et Accumulation:** For each non-zero element, the algorithm performs a multiplication for each column of the fat vector, so this step has complexity  $O(\frac{z}{m} \times k)$ . Thus, for all rows, the complexity becomes  $O(z \times k)$ .

Hence, the time complexity of multiplying a sparse matrix in CSR format with a fat matrix is  $O(z \times k)$ .

### 2.2.1.2 Spatial Complexity

The spatial complexity is related to the amount of memory required by the algorithm.

- **Sparse Matrix:** The matrix is stored using three main vectors (*values*, *colIndices*, *rowPtr*). If the matrix has  $z$  non-zero elements, then the spatial complexity of the two vectors *values* and *colIndices* is  $O(z)$ . The vector *rowPtr* has a size of  $m + 1$ , so its spatial complexity is  $O(m)$ . Thus, the space required for  $M$  in CSR format is  $O(z + z + m) = O(2z + m)$ .
- **Fat vector:** The fat vector has a spatial complexity of  $O(n \times k)$ .
- **Result Vector:** The result vector has a spatial complexity of  $O(m \times k)$ .

Considering the storage requirements for the sparse matrix, the dense matrix, and the result matrix, the overall spatial complexity of the multiplication operation is  $O(2z + m + n \times k + m \times k)$ .

## 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.

---

**Algorithm 2** Line-based parallel sparse matrix-vector multiplication
 

---

```

rowsPerProc  $\leftarrow m / \text{numProcs}$ 
extraRows  $\leftarrow m \% \text{numProcs}$ 
startRow  $\leftarrow \text{rank} \times \text{rowsPerProc} + \min(\text{rank}, \text{extraRows})$ 
endRow  $\leftarrow \text{startRow} + \text{rowsPerProc} + (\text{rank} < \text{extraRows})$ 
localRows  $\leftarrow \text{endRow} - \text{startRow}$ 
Result  $\leftarrow$  zero matrix of size  $\text{localRows} \times k$ 
for  $i \leftarrow \text{startRow}$  to  $\text{endRow} - 1$  do
  for each non-zero element  $(j, \text{value})$  in row  $i$  of  $M$  do
    for  $l \leftarrow 0$  to  $k - 1$  do
      Result[ $i - \text{startRow}$ ][ $l$ ]  $\leftarrow$  Result[ $i - \text{startRow}$ ][ $l$ ] + (value  $\times v[j][l]$ )
    end for
  end for
end for
if  $\text{rank} = 0$  then
  FinalResult  $\leftarrow$  zero matrix of size  $m \times k$ 
end if
Define  $\text{recvCounts}[\text{numProcs}]$ ,  $\text{displacements}[\text{numProcs}]$ 
Compute  $\text{recvCounts}$  and  $\text{displacements}$  based on  $\text{localRows}$  for all processes
MPI_Gatherv(Result,  $\text{localRows} \times k$ , MPI_DOUBLE, FinalResult,  $\text{recvCounts}$ ,  $\text{displacements}$ , MPI_DO
if  $\text{rank} = 0$  then
  for  $p \leftarrow 1$  to  $\text{numProcs} - 1$  do
    Integrate received partial Results into FinalResult based on  $\text{displacements}$ 
  end for
return FinalResult
end if

```

---

### 2.3.1 Complexity Analysis

#### 2.3.1.1 Temporal Complexity

The temporal (or time) complexity analysis of the sparse matrix-fat vector multiplication using MPI in a distributed-memory parallel environment involves understanding the computational steps required for each part of the process. The main components are as follows:

1. **Initialisation and Setup:** MPI initialisation and calculation of rows per process have a negligible time complexity compared to the actual computation, so they can be considered as  $O(1)$ .
2. **Local Computation:**
  - Each process computes the multiplication for its assigned subset of rows.

- If the sparse matrix has  $z$  non-zero elements in total, and these elements are evenly distributed across  $m$  rows, each process handles approximately  $\frac{z}{\text{worldSize}}$  non-zero elements.
- The computation for each non-zero element involves accessing the element and its corresponding column in the dense vector, followed by a multiplication and an addition. This operation is  $O(1)$ .
- Thus, the local computation for each process has a time complexity of  $O\left(\frac{z}{\text{worldSize}}\right)$ .

### 3. Communication (MPI\_Gatherv):

- The complexity of the MPI\_Gatherv operation depends on the implementation of the MPI library and the underlying network architecture.
- In general, gathering operations can be assumed to have a logarithmic complexity with respect to the number of processes, i.e.,  $O(\log(\text{worldSize}))$ , but this can vary.
- The amount of data transferred per process is proportional to the size of the local result, which is  $O\left(\frac{m}{\text{worldSize}} \times k\right)$ .

### 4. Final Assembly:

- The root process assembles the final result matrix. This step is essentially a concatenation of the results from each process and has a complexity linear to the total size of the result matrix, which is  $O(m \times k)$ .

Considering the parallel nature of the computation, the dominant factor in the time complexity is the local computation performed by each process, which is  $O\left(\frac{z}{\text{worldSize}}\right)$ . The communication step's complexity depends on the MPI implementation and network, but the data volume transferred per process affects this step. The final assembly in the root process is also significant but does not exceed  $O(m \times k)$ . Therefore, the overall time complexity of the algorithm can be approximated as  $O\left(\frac{z}{\text{worldSize}} + \log(\text{worldSize}) + m \times k\right)$ , with the understanding that the actual performance can be influenced by factors like network latency, bandwidth, and the distribution of non-zero elements in the sparse matrix.

#### 2.3.1.2 Spatial Complexity

The spatial complexity analysis of the algorithm performing sparse matrix-fat vector multiplication in a distributed-memory parallel environment using MPI can be understood by considering the memory requirements for storing the sparse matrix, the dense vector, and the final result, as well as the additional storage required for parallel processing. The breakdown is as follows:

##### 1. Storage of Sparse Matrix (sparseMatrix):

- The sparse matrix is stored in CSR format, which includes three arrays:
  - values: Contains all non-zero elements of the matrix. If the matrix has  $z$  non-zero elements, this array takes  $O(z)$  space.

- `colIndices`: Stores column indices for each non-zero element, also taking  $O(z)$  space.
- `rowPtr`: Contains  $m + 1$  elements for an  $m \times n$  matrix, requiring  $O(m + 1)$  space.
- Total space for the sparse matrix is therefore  $O(2z + m)$ .

## 2. Storage of Dense Vector (`denseVector`):

- The dense vector (or fat vector) is a 2D array of size  $n \times k$ . It requires  $O(n \times k)$  space.

## 3. Local Result Vector (`localResult`):

- Each process computes a local result vector of size proportional to the number of rows it handles times the number of columns in the dense vector. In the worst case, this is  $O\left(\frac{m}{\text{worldSize}} + 1 \times k\right)$  per process.

## 4. Gathered Results (`gatheredResults`):

- In the root process, the gathered results from all processes are combined. This requires space equivalent to the size of the final result matrix, which is  $O(m \times k)$ .

## 5. Final Result Matrix (`finalResult`):

- The final result matrix, which is only constructed in the root process, is of size  $m \times k$ , requiring  $O(m \times k)$  space.

## 6. Auxiliary Data for MPI Operations:

- Arrays like `recvCounts` and `displacements` are used for MPI communication. Their sizes are proportional to the number of processes, which is typically much smaller than the size of the matrix or vectors. Therefore, they add a relatively negligible  $O(\text{worldSize})$  space complexity.

The spatial complexity of the algorithm is dominated by the storage requirements for the sparse matrix, the dense vector, and the final result matrix. Thus, the overall spatial complexity is approximately  $O(2z + m + n \times k + m \times k)$ . It is important to note that in a distributed-memory environment, the memory usage is distributed across multiple processes, which can reduce the memory burden on individual nodes.

## 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.

---

**Algorithm 3** Column-wise Parallelization using MPI for Sparse Matrix-Fat Vector Multiplication

---

**Require:**  $M$  is an  $m \times n$  sparse matrix  
**Require:**  $v$  is an  $n \times k$  vector  
**Require:**  $numProcs$  is the number of processes  
**Require:**  $rank$  is the rank of the current process  
**Ensure:**  $PartialResult$  is a part of the  $m \times k$  matrix computed by this process

```

 $colsPerProc \leftarrow k / numProcs$ 
 $startCol \leftarrow rank \times colsPerProc$ 
 $endCol \leftarrow startCol + colsPerProc$ 
 $PartialResult \leftarrow$  zero matrix of size  $m \times colsPerProc$ 
for  $i \leftarrow 0$  to  $m - 1$  do
  for each non-zero element  $(j, value)$  in row  $i$  of  $M$  do
    for  $l \leftarrow startCol$  to  $endCol - 1$  do
       $PartialResult[i][l - startCol] \leftarrow PartialResult[i][l - startCol] + (value \times$ 
 $v[j][l])$ 
    end for
  end for
end for
if  $rank \neq 0$  then
  Send  $PartialResult$  to process 0
else
   $FinalResult \leftarrow$  zero matrix of size  $m \times k$ 
  Copy  $PartialResult$  into appropriate position in  $FinalResult$ 
  for  $p \leftarrow 1$  to  $numProcs - 1$  do
    Receive partial  $PartialResult$  from process  $p$ 
    Copy received  $PartialResult$  into appropriate position in  $FinalResult$ 
  end for
end if
if  $rank = 0$  then return  $FinalResult$ 

```

---

### 2.4.1 Complexity Analysis

#### 2.4.1.1 Temporal Complexity

To analyse the temporal (or time) complexity of the sparse matrix-fat vector multiplication using a column-wise parallel approach with MPI, we need to consider the computation and communication steps involved in the process. The breakdown is as follows:

1. **Local Computation:**

- Each MPI process computes a portion of the final matrix, responsible for a subset of columns. The number of columns processed by each process is roughly  $\text{colsPerProcess} = \frac{\text{vecCols}}{\text{worldSize}}$ , with some processes handling extra columns if  $\text{vecCols}$  is not perfectly divisible by  $\text{worldSize}$ .
- For each column, the process computes the product with every row of the sparse matrix. If the sparse matrix has  $z$  non-zero elements in total, then, on average, each process handles approximately  $\frac{z}{\text{worldSize}}$  non-zero elements.
- The computation involves accessing the element, performing a multiplication, and accumulating the result. These operations for each non-zero element are  $O(1)$ .
- Therefore, the local computation for each process has a time complexity of  $O\left(\frac{z}{\text{worldSize}}\right)$ .

### 2. Communication (MPI\_Gatherv):

- The MPI\_Gatherv operation is used to gather the local results from each process into the root process. The complexity of this operation depends on the implementation of MPI and the underlying network architecture.
- Generally, the gather operation can be assumed to have a complexity that grows with the number of processes and the amount of data being communicated. If the gathered data from each process is large, the communication time can become significant.
- The amount of data transferred per process is proportional to the number of rows in the matrix and the number of columns processed, which is  $O(m \times \text{colsPerProcess})$ .

### 3. Final Assembly:

- The root process assembles the final result matrix. This step is essentially a concatenation of results from each process and is linearly proportional to the size of the final matrix,  $O(m \times k)$ .

The dominant factor in the time complexity is the local computation performed by each process, which is  $O\left(\frac{z}{\text{worldSize}}\right)$ . The communication step can also be significant, especially if the network bandwidth is limited or if the size of the data being communicated is large. The final assembly in the root process is also linearly dependent on the size of the final result matrix. Therefore, the overall time complexity of the algorithm can be approximated as  $O\left(\frac{z}{\text{worldSize}} + \text{communication cost} + m \times k\right)$ , with the understanding that actual performance can vary based on factors like network performance, distribution of non-zero elements in the sparse matrix, and the specific implementation of MPI.

#### 2.4.1.2 Spatial Complexity

The spatial complexity of the given algorithm, which performs sparse matrix-fat vector multiplication in a distributed-memory environment using a column-wise parallel approach, can be analysed by considering the memory requirements for storing the matrix, the vector, and the intermediate and final computational results. The breakdown is as follows:



**1. Sparse Matrix Storage:**

- The sparse matrix is stored in a CSR format, which includes arrays for non-zero values, column indices, and row pointers. If the matrix has  $z$  non-zero elements, then the space required is approximately  $O(z)$  for the values,  $O(z)$  for the column indices, and  $O(m + 1)$  for the row pointers, where  $m$  is the number of rows. Therefore, the total space requirement for the sparse matrix is  $O(2z + m)$ .

**2. Dense Vector Storage:**

- The dense vector (or fat vector) is a 2D array of size  $n \times k$ , requiring  $O(n \times k)$  space.

**3. Local Result Vector:**

- Each process computes a local result vector for its assigned columns. The size of this local result vector is proportional to the number of rows in the sparse matrix and the number of columns processed by each process. The maximum size is  $O(m \times \text{colsPerProcess})$ , where  $\text{colsPerProcess}$  is the number of columns processed by each process.

**4. Gathered Results:**

- In the root process, the final gathered result needs to be stored. This is essentially the entire output matrix, which has a size of  $m \times k$ , hence requiring  $O(m \times k)$  space.

**5. Auxiliary Arrays for MPI Operations:**

- Arrays such as `recvCounts` and `displacements` are used in the MPI Gather operation. The size of these arrays is proportional to the number of processes (i.e., `worldSize`). However, this is generally small compared to the size of the matrices and vectors, so their space contribution is relatively minor.

Considering the storage requirements for the sparse matrix, the dense vector, the local results, and the final gathered results, the overall spatial complexity of the algorithm is approximately  $O(2z + m + n \times k + m \times \text{colsPerProcess} + m \times k)$ . It is important to note that in a distributed-memory setting, this memory usage is spread across multiple processes, reducing the memory load on any single node.

## 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.

---

**Algorithm 4** Non-Zero Element Parallelization using MPI for Sparse Matrix-Fat Vector Multiplication

---

**Require:**  $M$  is an  $m \times n$  sparse matrix stored in a format that allows iterating over non-zero elements (e.g., COO, CSR)

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

**Require:**  $numProcs$  is the number of processes

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

**Ensure:**  $PartialResult$  is a part of the  $m \times k$  matrix computed by this process

$numNonZeroElements \leftarrow$  total number of non-zero elements in  $M$

$elementsPerProc \leftarrow numNonZeroElements / numProcs$

$startIndex \leftarrow rank \times elementsPerProc$

$endIndex \leftarrow startIndex + elementsPerProc$

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

$NonZeroElements \leftarrow$  list of non-zero elements in  $M$  from  $startIndex$  to  $endIndex - 1$

**for** each  $(i, j, value)$  in  $NonZeroElements$  **do**

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

$PartialResult[i][l] \leftarrow PartialResult[i][l] + (value \times v[j][l])$

**end for**

**end for**

**if**  $rank \neq 0$  **then**

    Send  $PartialResult$  to process 0

**else**

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

    Copy  $PartialResult$  into  $FinalResult$

**for**  $p \leftarrow 1$  to  $numProcs - 1$  **do**

        Receive partial  $PartialResult$  from process  $p$

        Add received  $PartialResult$  into  $FinalResult$

**end for**

**end if**

**if**  $rank = 0$  **then return**  $FinalResult$

---

### 2.5.1 Complexity Analysis

#### 2.5.1.1 Temporal Complexity

- **MPI Initialisation and Rank and Size Determination:** As with other MPI-based algorithms, this step has a complexity of approximately  $O(1)$ .
- **Scattering Chunks of Rows of  $M$  to Each Process:** This step distributes parts of the matrix to different processes. Its complexity depends on the number of rows

and the distribution method, typically around  $O(\frac{m}{p})$ , where  $m$  is the number of rows and  $p$  is the number of processes.

- **Scatter of Vector  $v$  to All Processes:** This operation generally has a complexity of  $O(n)$ , where  $n$  is the size of the vector.
- **Local Computations for Non-Zero Elements:** Each process computes the products for the non-zero elements in its assigned rows. Assuming an even distribution of non-zero elements, the complexity for each process is approximately  $O(\frac{n_{nz}}{p})$ .
- **Gather of Local Results  $r_{local}$  into Final Result Vector  $r$ :** This step combines the partial results from all processes and typically has a complexity proportional to the total number of elements in  $r$ .

### 2.5.1.2 Spatial Complexity

- **Storage of Sparse Matrix and fat vector:** The overall storage requirements remain  $O(n_{nz} + m + n)$ , as in other sparse matrix-vector multiplication methods.
- **Local Result Vectors  $r_{local}$ :** Each process stores a local result vector for its chunk of rows, with the size depending on the distribution of rows and non-zero elements.

## Complexity and Considerations

Each of these parallel algorithms aims to exploit different aspects of parallelism, with the primary goal of reducing the overall computation time. The actual performance gain depends on the characteristics of the sparse matrix, the number of available processing units, and the specific implementation details. Moreover, care must be taken to manage concurrency issues, such as race conditions and proper synchronization, to ensure correct and efficient execution.

# Chapter 3

## Results and Discussion

### 3.1 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**

# Appendix A

## Documentation

### Appendix A.A Project tree

```
lib /
    collecting.py
    processing.py
    storing.py
scripts /
    get_iam_credentials.sh
    start_spark_job.sh
services /
    get_iam_credentials.service
    spark_python_job.service
test /
    artillery_load_test.yml
    monitoring.py
    metrics.csv
    results.json
    visualisation_load_test.ipynb
main.py
README.md
requirements.txt
```

### Appendix A.B Getting Started

To run the program, follow these steps:

1. Create a virtual environment using `python3 -m venv venv`.
2. Activate the virtual environment using `source venv/bin/activate`.
3. Install the required dependencies using `pip3 install -r requirements.txt`.
4. Run the program using `python3 main.py`.
5. Visualise the results using `visualisation.ipynb` (Jupyter Notebook).

## Appendix A.C Detailed Features of Functions

`collecting.py`

- `fetch_sensors_data(sparkSession)`: Function to ingest the latest data from the sensors and returns it as a Spark DataFrame.

`processing.py`

- `get_aqi_value_p25(value)`: Function for calculating the AQI value for PM2.5.
- `get_aqi_value_p10(value)`: Function for calculating the AQI value for PM10.
- `computeAQI(df)`: Function for calculating the AQI value for each particulate matter sensor and returning the DataFrame with the AQI column.

`storing.py`

- `keepOnlyUpdatedRows(database_name, table_name, df)`: Function for keeping only the rows that have been updated in the DataFrame.
- `_print_rejected_records_exceptions(err)`: Internal function for printing the rejected records exceptions.
- `write_records(database_name, table_name, client, records)`: Internal function for writing a batch of records to the Timestream database.
- `writeToTimestream(database_name, table_name, partitioned_df)`: Function for writing the DataFrame to the Timestream database.

# 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 dense vector
22  typedef std::vector<std::vector<double>> DenseVector;
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 SPARSEMATRIXDENSEVECTORMULTIPLY_H
2 #define SPARSEMATRIXDENSEVECTORMULTIPLY_H
3
4 #include "MatrixDefinitions.h"
5
6 /**
7  * @brief Function to execute the sparse matrix-dense vector multiplication using
8  * sequential algorithm
9  *
10  * @param sparseMatrix Sparse matrix
11  * @param denseVector Dense vector
12  * @param vecCols Number of columns in the dense vector
13  * @return DenseVector Result of the multiplication
14  */
15 DenseVector sparseMatrixDenseVectorMultiply(const SparseMatrix &sparseMatrix,
16                                             const DenseVector &denseVector, int
17                                             vecCols);
18 #endif

```

### B.B.2 Implementation File

```

1 #include "SparseMatrixDenseVectorMultiply.h"
2
3 /**
4  * @brief Function to execute the sparse matrix-dense vector multiplication using
5  * sequential algorithm
6  *
7  * @param sparseMatrix Sparse matrix
8  * @param denseVector Dense vector
9  * @param vecCols Number of columns in the dense vector
10  * @return DenseVector Result of the multiplication
11  */
12 DenseVector sparseMatrixDenseVectorMultiply(const SparseMatrix &sparseMatrix,
13                                             const DenseVector &denseVector, int
14                                             vecCols)
15 {
16     // Initialisation of the result vector
17     DenseVector 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 dense vector
25             for (int k = 0; k < vecCols; ++k)
26             {
27                 result[i][k] += sparseMatrix.values[j] * denseVector[sparseMatrix.
28                 colIndices[j]][k]; // Compute the result
29             }
30         }
31     }
32     // Return the result
33     return result;
34 }

```

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

```

### B.C.2 Implementation File

```

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

```

```

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

```

```

76         }
77     }
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    DenseVector 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 : DenseVector{};
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 SPARSEMATRIXDENSEVECTORMULTIPLYCOLUMNWISE_H
2 #define SPARSEMATRIXDENSEVECTORMULTIPLYCOLUMNWISE_H
3
4 #include "MatrixDefinitions.h"
5 #include <iostream> // std::cout
6
7 /**
8  * @brief Function to execute the sparse matrix-dense vector multiplication using
9  *        column-wise parallel algorithm
10  *
11  * @param sparseMatrix Sparse matrix
12  * @param denseVector Dense vector
13  * @param vecCols Number of columns in the dense vector
14  * @return DenseVector Result of the multiplication
15  */
16 DenseVector sparseMatrixDenseVectorMultiplyColumnWise(const SparseMatrix &
17   sparseMatrix, const DenseVector &denseVector, int vecCols);
18 #endif

```

### B.D.2 Implementation File

```

1 #include <mpi.h>
2 #include "SparseMatrixDenseVectorMultiplyColumnWise.h"
3 #include <numeric> // Pour std::accumulate
4
5 /**
6  * @brief Function to execute the sparse matrix-dense vector multiplication using
7  *        column-wise parallel algorithm
8  *
9  * @param sparseMatrix Sparse matrix
10  * @param denseVector Dense vector
11  * @param vecCols Number of columns in the dense vector
12  * @return DenseVector Result of the multiplication
13  */
14 DenseVector sparseMatrixDenseVectorMultiplyColumnWise(const SparseMatrix &
15   sparseMatrix, const DenseVector &denseVector, int vecCols)
16 {
17     // MPI Initialisation
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;
//
// Starting column index for the current process
28     int endCol = (worldRank != worldSize - 1) ? startCol + colsPerProcess :
        startCol + colsPerProcess + extraCols; // Ending column index for the
        current process
29
30     // Local computation
31     int localSize = sparseMatrix.numRows * (endCol - startCol); // Number of
        elements in the local result vector
32     std::vector<double> localResult(localSize, 0.0);
33     // Iterate over the columns assigned to the current process
34     for (int col = startCol; col < endCol; ++col)
35     {
36         // Iterate over the rows of the sparse matrix
37         for (int i = 0; i < sparseMatrix.numRows; ++i)
38         {
39             // Iterate over the non-zero elements in the current row
40             double sum = 0.0;
41             for (int j = sparseMatrix.rowPtr[i]; j < sparseMatrix.rowPtr[i + 1]; ++
                j)
42             {
43                 int sparseCol = sparseMatrix.colIndices[j]; //
                Column index of the non-zero element
44                 sum += sparseMatrix.values[j] * denseVector[sparseCol][col]; //
                Compute the result
45             }
46             localResult[i * (endCol - startCol) + (col - startCol)] = sum; // Store
                the result in the local result vector
47         }
48     }
49
50     // ===== FOR DEBUGGING ONLY - STOP LOCAL COMPUTATION
51     TIMER =====
52     // double computation_end = MPI_Wtime();
53     // double local_computation_time = computation_end - computation_start;
54     // ===== FOR DEBUGGING ONLY - STOP LOCAL COMPUTATION
55     TIMER =====
56
57     // ===== FOR DEBUGGING ONLY - START COMMUNICATION TIMER
58     =====
59     // Start timing for communication
60     // double communication_start = MPI_Wtime();
61     // ===== FOR DEBUGGING ONLY - START COMMUNICATION TIMER
62     =====
63
64     // Preparation for Gather operation
65     std::vector<int> recvCounts(worldSize), displacements(worldSize); // Number of
        elements to be received from each process, Displacement for each process
66     if (worldRank == 0)
67     {
68         // Compute the number of elements to be received from each process
69         int displacement = 0;
70         for (int i = 0; i < worldSize; ++i)
71         {
72             int startColThisRank = i * colsPerProcess;
73
74             // Starting column index for the current process
75             int endColThisRank = (i != worldSize - 1) ? startColThisRank +
                colsPerProcess : startColThisRank + colsPerProcess + extraCols; //
                Ending column index for the current process
76             recvCounts[i] = sparseMatrix.numRows * (endColThisRank -
                startColThisRank);
77
78             // Number of
79             elements to be received from the current process
80             displacements[i] = displacement;
81
82             // Displacement for the current process
83             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 DenseVector 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
129 // Return the final result
130 return (worldRank == 0) ? finalResult : DenseVector{};
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 SPARSEMATRIXDENSEVECTORMULTIPLYNONZEROELEMENT_H
2  #define SPARSEMATRIXDENSEVECTORMULTIPLYNONZEROELEMENT_H
3
4  #include "MatrixDefinitions.h"
5  #include <iostream> // std::cout
6
7  /**
8   * @brief Function to execute the sparse matrix-dense vector multiplication using
9   *        non-zero element parallel algorithm
10   *
11   * @param sparseMatrix Sparse matrix
12   * @param denseVector Dense vector
13   * @param vecCols Number of columns in the dense vector
14   * @return DenseVector Result of the multiplication
15   */
16 DenseVector sparseMatrixDenseVectorMultiplyNonZeroElement(const SparseMatrix &
17     sparseMatrix, const DenseVector &denseVector, int vecCols);
18
19 #endif

```

### B.E.2 Implementation File

```

1  #include "SparseMatrixDenseVectorMultiplyNonZeroElement.h"
2  #include <mpi.h>
3
4  /**
5   * @brief Function to execute the sparse matrix-dense vector multiplication using
6   *        non-zero element parallel algorithm
7   *
8   * @param sparseMatrix Sparse matrix
9   * @param denseVector Dense vector
10   * @param vecCols Number of columns in the dense vector
11   * @return DenseVector Result of the multiplication
12   */
13 DenseVector sparseMatrixDenseVectorMultiplyNonZeroElement(const SparseMatrix &
14     sparseMatrix, const DenseVector &denseVector, int vecCols)
15 {
16     // MPI Initialisation
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 dense vector
63     for (int k = 0; k < vecCols; ++k)
64     {
65         localResult[row * vecCols + k] += value * denseVector[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 DenseVector 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 : DenseVector{};
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 dense vector
21  * @param C PETSc matrix
22  * @return DenseVector Dense vector
23  */
24 DenseVector ConvertPETScMatToDenseVector(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 DenseVector &mat1, const DenseVector &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 dense vector
44  * @param n Number of rows
45  * @param m Number of columns
46  * @return DenseVector Dense vector
47  */
48 DenseVector generateLargeDenseVector(int n, int k);
49
50 /**
51  * @brief Method to serialize a DenseVector to a flat array
52  * @param denseVec Dense vector to serialize
53  * @return std::vector<double> Flat array containing the serialized data
54  */
55 std::vector<double> serialize(const DenseVector &denseVec);
56
57 /**
58  * @brief Method to deserialize a flat array to a DenseVector
59  * @param flat Flat array to deserialize
60  * @param rows Number of rows in the dense vector

```

```

61  * @param cols Number of columns in the dense vector
62  * @return DenseVector Dense vector
63  */
64  DenseVector 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 dense vector
5   * @param C PETSc matrix
6   * @return DenseVector Dense vector
7   */
8  DenseVector ConvertPETScMatToDenseVector(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     DenseVector denseVec(m, std::vector<double>(n, 0.0)); // Dense vector to hold
14     // the 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             MatGetValue(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 dense 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 DenseVector &mat1, const DenseVector &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 dense vector
189  * @param n Number of rows
190  * @param m Number of columns
191  * @return DenseVector Dense vector
192  */
193 DenseVector generateLargeDenseVector(int n, int k)
194 {
195     DenseVector denseVector(n, std::vector<double>(k)); // Dense vector to hold the
196     // random values
197
198     // Iterate over the rows of the dense vector
199     for (int i = 0; i < n; ++i)
200     {
201         // Iterate over the columns of the dense 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 dense vector
210     return denseVector;
211 }
212
213 /**
214  * @brief Method to serialize a DenseVector to a flat array
215  * @param denseVec Dense vector to serialize
216  * @return std::vector<double> Flat array containing the serialized data
217  */
218 std::vector<double> serialize(const DenseVector &denseVec)
219 {
220     std::vector<double> flat; // Flat array to hold the serialized data
221
222     // Iterate over the rows of the dense 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 DenseVector
234  * @param flat Flat array to deserialize
235  * @param rows Number of rows in the dense vector
236  * @param cols Number of columns in the dense vector
237  * @return DenseVector Dense vector
238  */
239 DenseVector deserialize(const std::vector<double> &flat, int rows, int cols)
240 {
241     DenseVector denseVec(rows, std::vector<double>(cols)); // Dense vector to hold
242     // the deserialized data
243
244     // Iterate over the rows of the dense vector
245     for (int i = 0; i < rows; ++i)
246     {
247         // Iterate over the columns of the dense 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 dense 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 "SparseMatrixDenseVectorMultiply.h" // Sequential algorithm
3  #include "SparseMatrixDenseVectorMultiplyRowWise.h" // Parallel algorithm (
   row-wise)
4  #include "SparseMatrixDenseVectorMultiplyColumnWise.h" // Parallel algorithm (
   column-wise)
5  #include "SparseMatrixDenseVectorMultiplyNonZeroElement.h" // Parallel algorithm (
   non-zero element)
6
7  int main(int argc, char *argv[])
8  {
9      //
10     // ===== INITIALISATION
11     // =====
12
13     // Initialise MPI and PETSc
14     MPI_Init(&argc, &argv);
15     PetscInitialize(&argc, &argv, NULL, NULL);
16
17     // Retrieve the rank and size of the world communicator
18     int worldRank, worldSize;
19     MPI_Comm_rank(PETSC_COMM_WORLD, &worldRank);
20     MPI_Comm_size(PETSC_COMM_WORLD, &worldSize);
21
22     // Check if the correct number of arguments is provided
23     if (argc != 3)
24     {
25         if (worldRank == 0)
26         {
27             std::cerr << "Usage: " << argv[0] << " <number of columns> <matrix file
               path>" << std::endl;
28         }
29         MPI_Abort(PETSC_COMM_WORLD, 1);
30     }
31
32     // Parse the command-line arguments
33     int k = std::atoi(argv[1]); // Convert the first argument to an integer
34     std::string filename = argv[2]; // The second argument is the filename
35
36     // Declare the sparse matrix and dense vector
37     SparseMatrix M;
38     DenseVector v;
39
40     // Declare the result of the serial multiplication
41     DenseVector resultSerial;
42
43     // Declare the data for broadcasting the sparse matrix and dense vector
44     std::vector<double> flatData;
45     int dataSize = 0;
46
47     // Declare the variables for timing the execution of the algorithms

```



```

48     double startTime, endTime;
49
50     //
51     // ===== READ THE SPARSE MATRIX AND GENERATE THE DENSE
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 dense vector
66         v = generateLargeDenseVector(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 dense 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 = sparseMatrixDenseVectorMultiply(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 DENSE
101    // VECTOR =====
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 Dense 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 DenseVector resultRowWise = sparseMatrixDenseVectorMultiplyRowWise(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 DenseVector resultColumnWise = sparseMatrixDenseVectorMultiplyColumnWise(M, v,
210 k);
211 endTime = MPI_Wtime();
212
213 // Only the main process prints the parallel execution time
214 if (worldRank == 0)
215 {
216     std::cout << "Column-wise Execution time: " << (endTime - startTime)
217     << std::endl;
218
219     // ===== FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS

```

```

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     DenseVector resultNonZeroElement =
248         sparseMatrixDenseVectorMultiplyNonZeroElement(M, 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
290 // Declare the PETSc matrix
291 Mat A, B, C;
292
293 // ===== FOR DEBUGGING ONLY - START PETSCS SETUP TIMER
294 // startTime = MPI_Wtime();
295 // ===== FOR DEBUGGING ONLY - START PETSCS SETUP TIMER
296 // =====
297
298 // Create a parallel matrix to store the sparse matrix
299 MatCreate(PETSC_COMM_WORLD, &A);
300 MatSetSizes(A, PETSC_DECIDE, PETSC_DECIDE, M.numRows, M.numCols);
301 MatSetType(A, MATMPIAIJ);
302 MatSetUp(A);
303 // Fill the PETSc matrix with the values from the sparse matrix
304 if (worldRank == 0)
305 {
306     for (int i = 0; i < M.numRows; ++i)
307     {
308         for (int j = M.rowPtr[i]; j < M.rowPtr[i + 1]; ++j)
309         {
310             MatSetValue(A, i, M.colIndices[j], M.values[j], INSERT_VALUES);
311         }
312     }
313 // Assemble the PETSc matrix
314 MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
315 MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
316
317 // Create a parallel matrix to store the dense vector
318 MatCreate(PETSC_COMM_WORLD, &B);
319 MatSetSizes(B, PETSC_DECIDE, PETSC_DECIDE, M.numCols, k);
320 MatSetType(B, MATDENSE);
321 MatSetUp(B);
322 // Fill the PETSc matrix B with values from the dense matrix v
323 if (worldRank == 0)
324 {
325     for (int i = 0; i < M.numCols; ++i)
326     {
327         for (int j = 0; j < k; ++j)
328         {
329             MatSetValue(B, i, j, v[i][j], INSERT_VALUES);
330         }
331     }
332 // Assemble the PETSc matrix
333 MatAssemblyBegin(B, MAT_FINAL_ASSEMBLY);
334 MatAssemblyEnd(B, MAT_FINAL_ASSEMBLY);

```

```

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

```

```

394         << std::endl;
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@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.$$
export I_MPI_HYDRA_IFACE=ib0
export I_MPI_HYDRA_BOOTSTRAP=ssh
export I_MPI_HYDRA_RM=pbs
export K_VALUE=1
export MATRIX_PATH=/mnt/beegfs/home/s425500/hpc/assignment/sparse-matrix/cop20k_A.
    mtx
## Debug options - only enable when instructed by HPC support
##export I_MPI_HYDRA_DEBUG=1
##export I_MPI_DEBUG=6
##export I_MPI_DEBUG_OUTPUT=%h-%r-%p-debug.out
## set some MPI tuning parameters to use the correct transport
## =====
## AND HERE
## =====
##
## STEP 5:
##
## Load the default application environment
## For a specific version add the version number, e.g.
## module load intel/2016b
##
module use /apps/modules/all
module load intel/2021b
##
## STEP 6:
##
## Run MPI code
##
## The main parameter to modify is your mpi program name
## - change YOUR_EXECUTABLE to your own filename
##
mpirun -genval1 -hostfile mpi_nodes.$$ -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 dense 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,Dense 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"

```