Alexis Balayre

High Performance Technical Computing Assignment

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

MSc
Academic Year: 2023 - 2024

Supervisor: Dr Irene Moulitsas
5th February 2024

Abstract

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

Table of Contents

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 or solving structural engineering problems.

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

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

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

The matrix M is defined as:

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

where most elements of M are zeros.

The vector v is defined as:

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

Chapter 2

Methodology

2.1 Data Structures

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

2.1.1 Sparse Matrix

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

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

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

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

This hollow matrix can be visualised as:

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

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

2.1.2 Fat Vector

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

2.2 Sequential Algorithm

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

2.2.1 Algorithm Flow

Algorithm 1 Sequential algorithm

Require: M is an $m \times n$ sparse matrix
Require: v is an $n \times k$ fat vector
Ensure: *Result* is an $m \times k$ matrix
  Result $\leftarrow$ zero matrix of size $m \times k$
  for $i \leftarrow 0$ **to** $m-1$ **do**
    for each non-zero element (j, value) in row i of M **do**
      for $j \leftarrow 0$ **to** $k-1$ **do**
        $Result[i][l] \leftarrow Result[i][l] + (\text{value} \times v[j][l])$
      end for
    end for
  end for
  return *Result*

The algorithmic flow can be more explicitly detailed by:

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

2.2.2 Temporal Complexity Analysis

The time complexity can be broken down into the following components:

1. **Matrix Row Traversal:** Each row of the matrix is traversed once, so the complexity of traversing all the rows is $O(m)$.

2. **Non-zero Element Access:** The CSR format ensures efficient access to non-zero elements, attributing a complexity of $O(z)$ for parsing all such elements across the matrix.

3. **Multiplication et Accumulation:** The key computational step involves multiplying each non-zero element by the corresponding entries in v, across all k columns. Given that each non-zero element undergoes this operation, the complexity is more accurately described as $O(zk)$, highlighting the direct proportionality to both the number of non-zero elements and the vector's column dimension.

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

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.

2.3.1 Algorithm Flow

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

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

The algorithmic flow can be explicitly detailed by:

1. **Initialisation:** Obtain MPI world size and rank to determine each process's role.
2. **Row Distribution:** Assign a subset of rows from the sparse matrix to each process based on the rank, ensuring an even distribution with possible adjustments for any remainder.

3. **Local Computation:** Each process calculates the product of its assigned rows with the fat vector, storing results in a local vector.
4. **Gather Results:** Use `MPI_Gatherv` to collect the local result vectors from all processes into a single vector at the root process.
5. **Final Result Reconstruction:** The root process reconstructs the final result matrix from the gathered vector.

2.3.2 Temporal Complexity Analysis

The time complexity of the line-based parallel algorithm can be broken down 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.

2.4 Column-Wise Parallelism

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

2.4.1 Algorithm Flow

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

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

The algorithmic flow can be explicitly detailed by:

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

3. **Local Computation:** Each process computes a portion of the multiplication result for its assigned columns, iterating through the sparse matrix rows and the relevant columns of the fat vector.

4. **Gather Results:** Use `MPI_Gatherv` to collect the local results from all processes into a single result vector at the root process.

5. **Final Result Reconstruction:** The root process reassembles the gathered results into the final fat vector matrix, ensuring the elements are correctly positioned according to their original indices.

2.4.2 Temporal Complexity Analysis

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 colsPerProcess $= \frac{\text{k}}{\text{worldSize}}$, with some processes handling extra columns if k is not perfectly divisible by 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`):**

   - After computing the local results, processes use `MPI_Gatherv` to gather these results at the root process. The complexity of this operation can vary based on the MPI implementation and network conditions but generally involves logarithmic complexity with respect to the number of processes, $O(\log(worldSize))$, for the gathering operation itself.
   - The size of data being communicated by each process is proportional to its portion of the result matrix, which can be approximated as $O(m \times \text{colsPerProcess})$.

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, $O(m \times k)$.

Combining these components, the overall temporal complexity of the algorithm can be summarised as:

$$O\left(\frac{z}{worldSize} + \log(worldSize) + m \times k\right)$$

This analysis reflects the balanced distribution of computation through column-wise partitioning, the efficiency of parallel computation, and the inherent costs of communication and data assembly in a distributed-memory parallel computing environment. It's important to note that the actual performance may also depend on the specifics of the MPI environment, network bandwidth and latency, and the distribution of non-zero elements in the sparse matrix.

2.5 Non-Zero Element Parallelism

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

2.5.1 Algorithm Flow

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

Require: M is an $m \times n$ sparse matrix
Require: v is an $n \times k$ fat vector
Require: $worldSize$ is the number of processes
Require: $worldRank$ is the rank of the current process
Ensure: $finalResult$ is an $m \times k$ matrix, result of $M \times v$
  Calculate the total number of non-zero elements and distribute them among MPI processes
  Determine $startIdx$ and $endIdx$ for non-zero elements for the current process
  Map non-zero element indices to their corresponding row indices in the sparse matrix
  Initialize $localResult$ with zeros of size $m \times k$
  for $idx \leftarrow startIdx$ **to** $endIdx - 1$ **do**
    Determine row, col, and $value$ for each non-zero element
    for $k \leftarrow 0$ **to** $k - 1$ **do**
      $localResult[row \times k + k] \leftarrow localResult[row \times k + k] + value \times v[col][k]$
    end for
  end for
  Use MPI_Reduce to sum up $localResult$s from all processes to $flatFinalResult$ at the root process
  if worldRank == 0 **then**
    Reconstruct $finalResult$ from $flatFinalResult$
    return $finalResult$
  end if

The algorithmic flow can be explicitly detailed by:

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

2.5.2 Temporal Complexity Analysis

- **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.

2.6 Performance Metrics

In order to evaluate the performance of all the algorithms, they were implemented by the main program (Appendix B.G) and executed on CRESCENT2. The script `run.sh` (Appendix **??**) was used

Chapter 3

Results and Discussion

3.1 Results

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

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

Table 3.1: Sparse matrix specifications

3.1.1 Sparse Matrix Impact

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

3.1.1.1 Execution Time

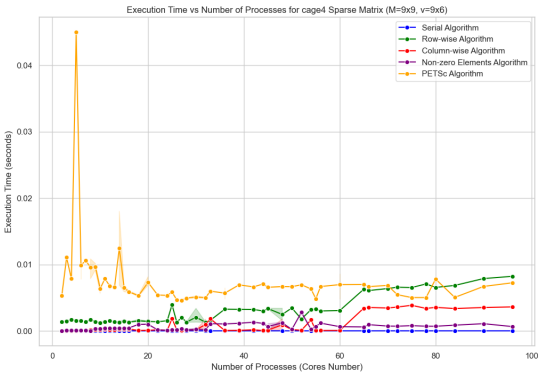

Figure 3.1: Cage4 matrix execution time

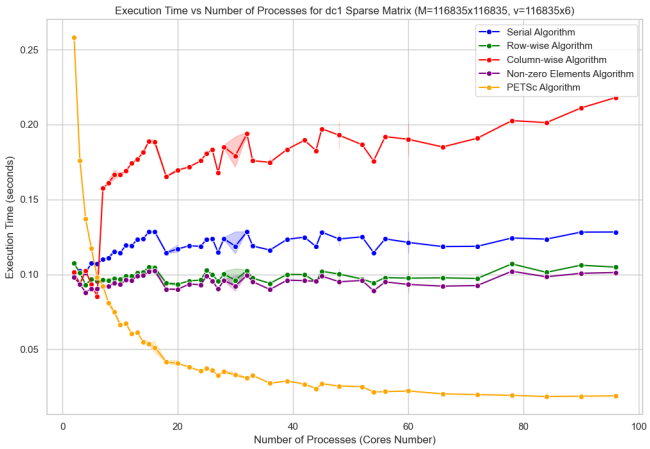

Figure 3.2: DC1 matrix execution time

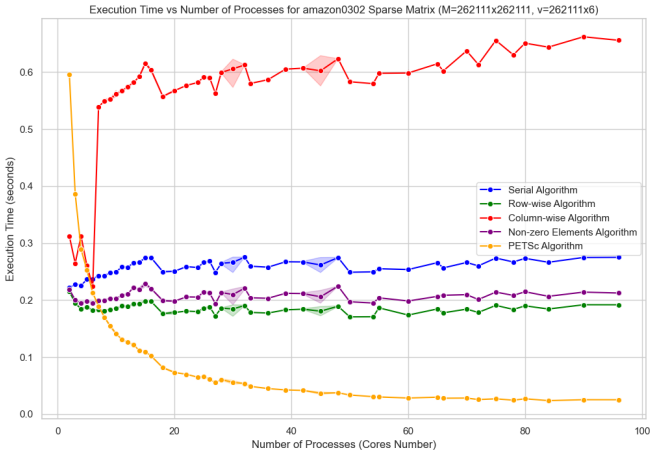

Figure 3.3: Amazon0302 matrix execution time

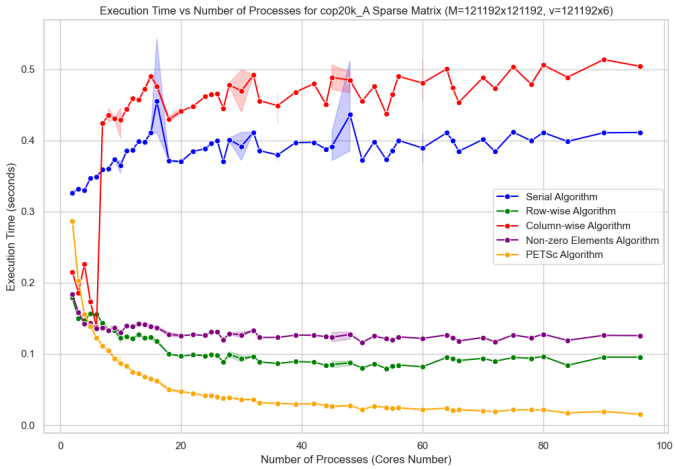

Figure 3.4: Cop20k_A matrix execution time

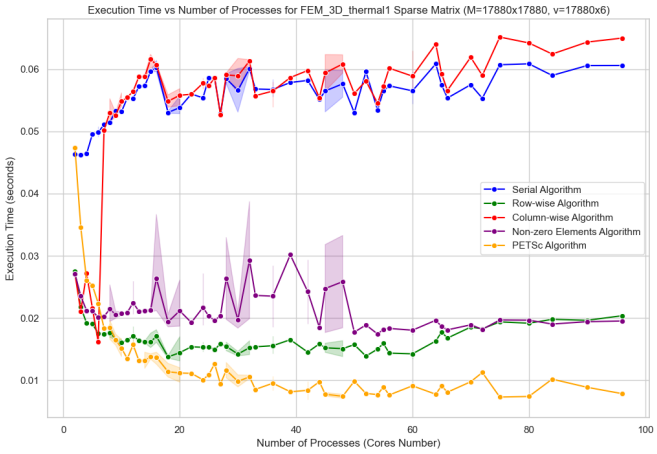

Figure 3.5: FEM_3D_thermal1 matrix execution time

3.1.1.2 Communication Time

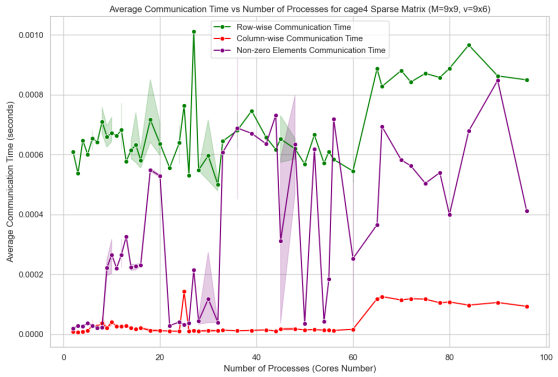

Figure 3.6: Cage4 matrix communication time

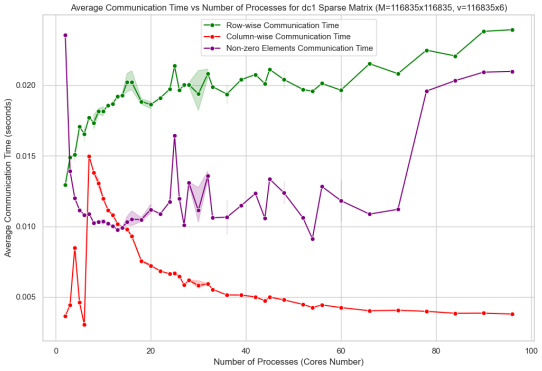

Figure 3.7: DC1 matrix communication time

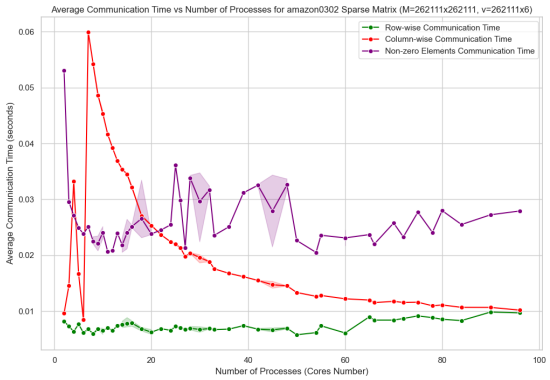

Figure 3.8: Amazon0302 matrix communication time

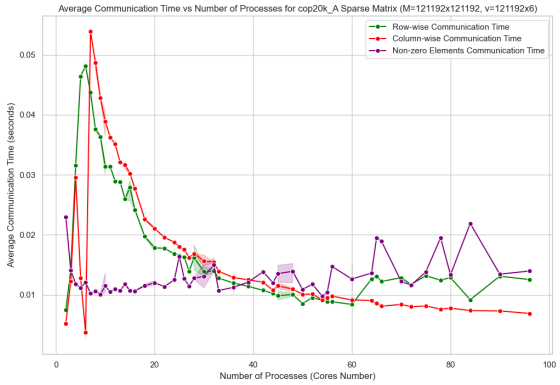

Figure 3.9: Cop20k_A matrix communication time

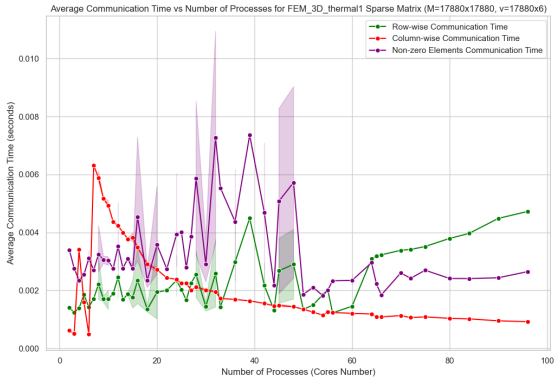

Figure 3.10: FEM_3D_thermal1 matrix communication time

3.1.1.3 Computation Time

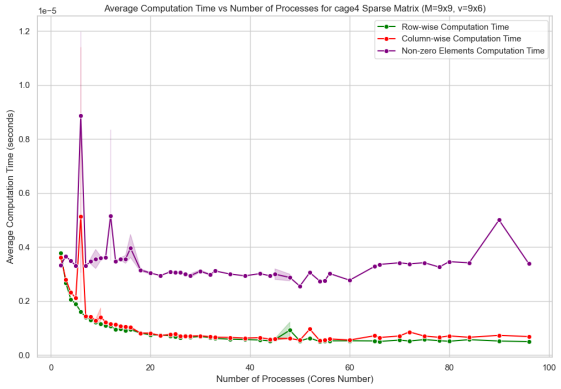

Figure 3.11: Cage4 matrix computation time

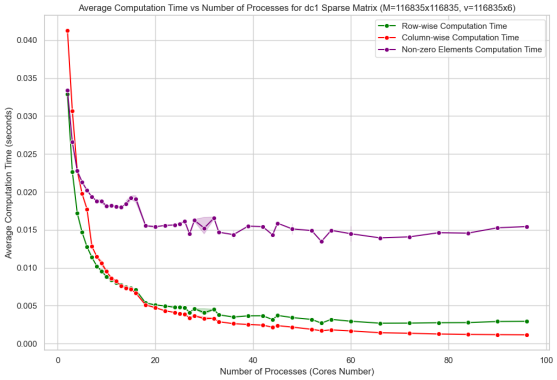

Figure 3.12: DC1 matrix computation time

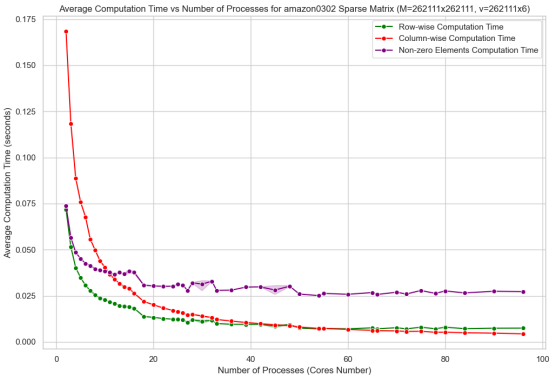

Figure 3.13: Amazon0302 matrix computation time

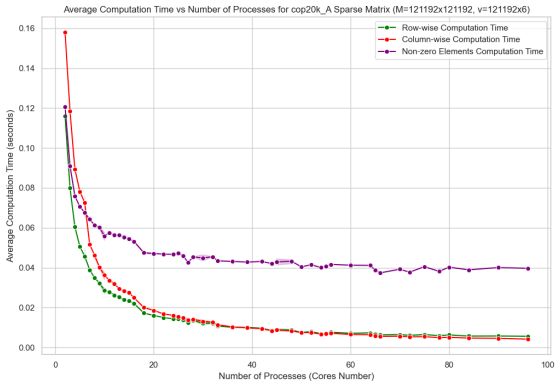

Figure 3.14: Cop20k_A matrix computation time

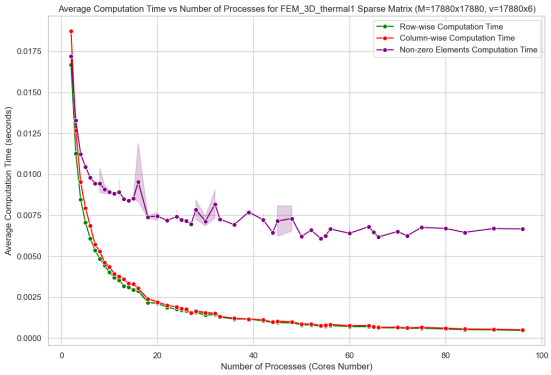

Figure 3.15: FEM_3D_thermal1 matrix computation time

3.1.1.4 Performance

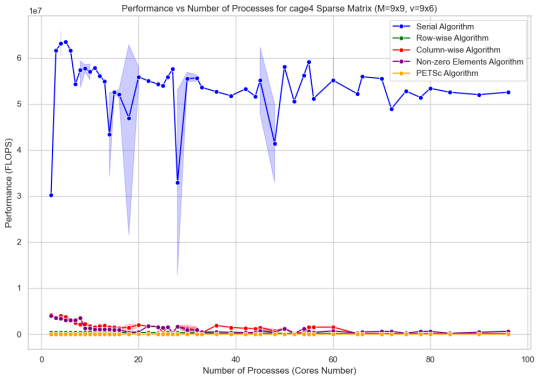

Figure 3.16: Cage4 matrix performance

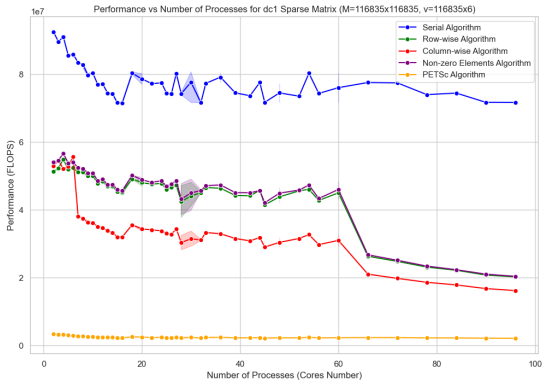

Figure 3.17: DC1 matrix performance

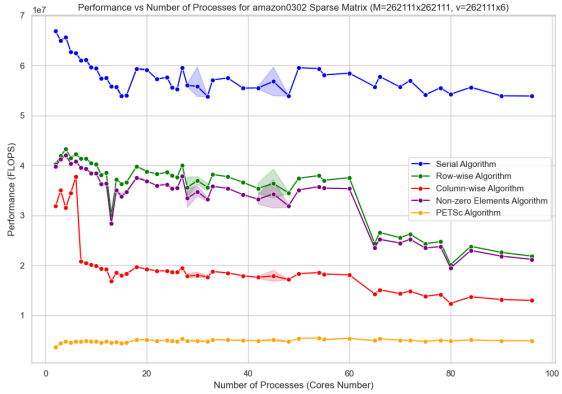

Figure 3.18: Amazon0302 matrix performance

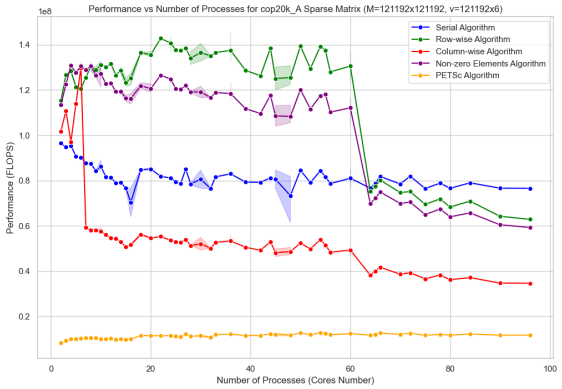

Figure 3.19: Cop20k_A matrix performance

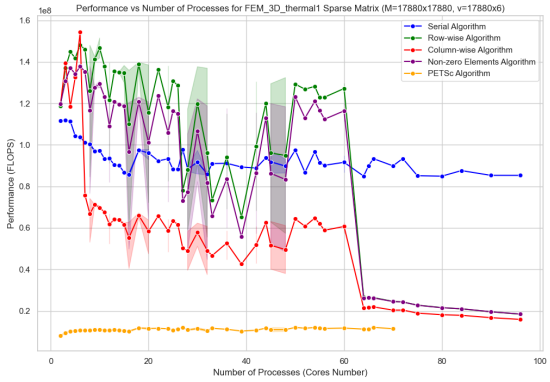

Figure 3.20: FEM_3D_thermal1 matrix performance

3.1.2 Fat Vector Impact

3.1.2.1 Execution Time

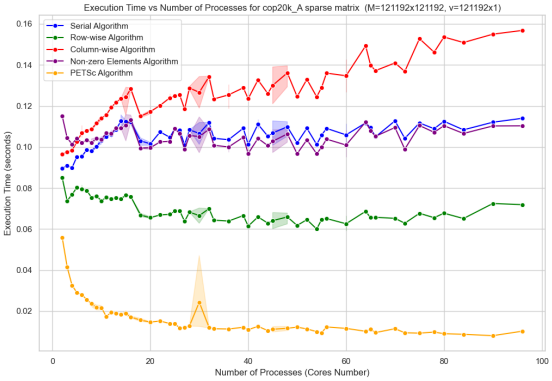

Figure 3.21: Cop20k_A matrix execution time

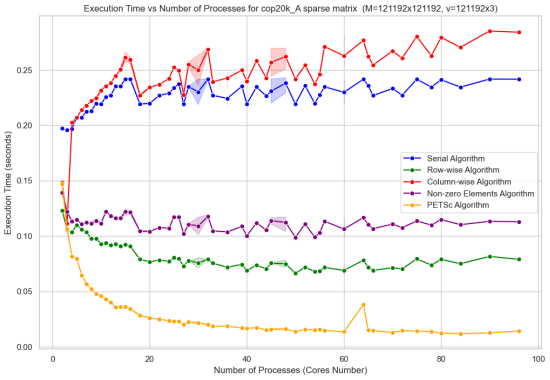

Figure 3.22: Cop20k_A matrix execution time

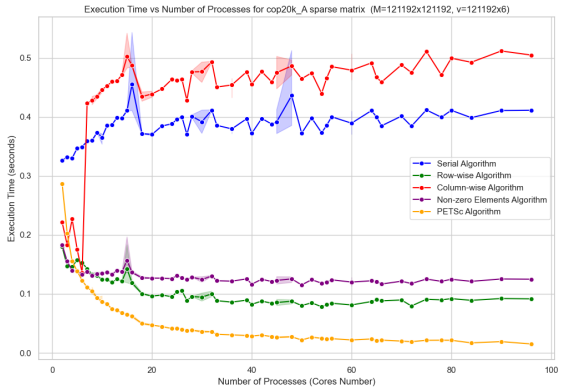

Figure 3.23: Cop20k_A matrix execution time

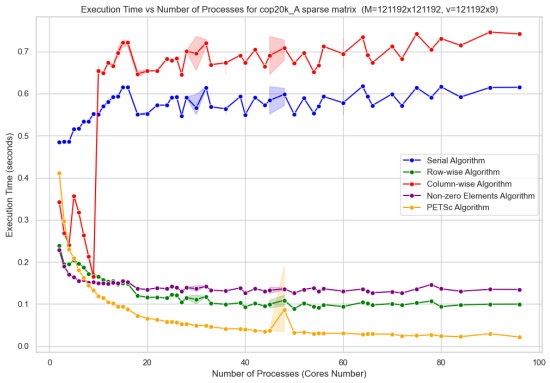

Figure 3.24: Cop20k_A matrix execution time

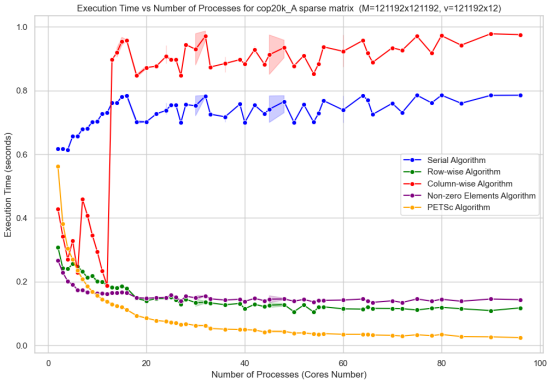

Figure 3.25: Cop20k_A matrix execution time

3.1.2.2 Communication Time

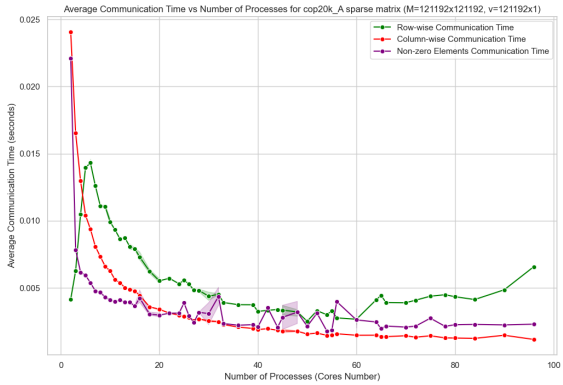

Figure 3.26: Cop20k_A matrix communication time

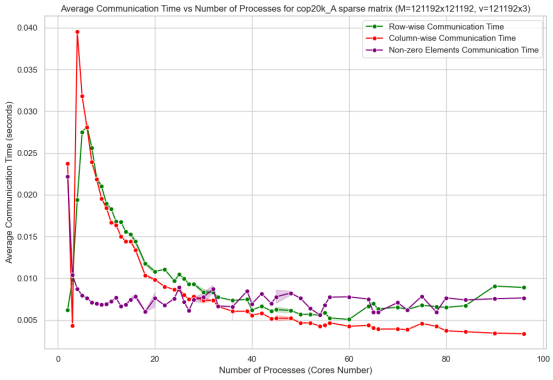

Figure 3.27: Cop20k_A matrix communication time

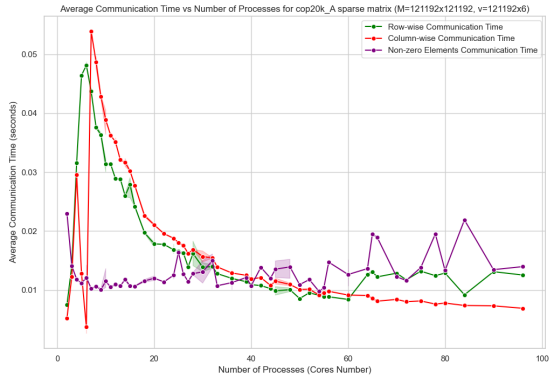

Figure 3.28: Cop20k_A matrix communication time

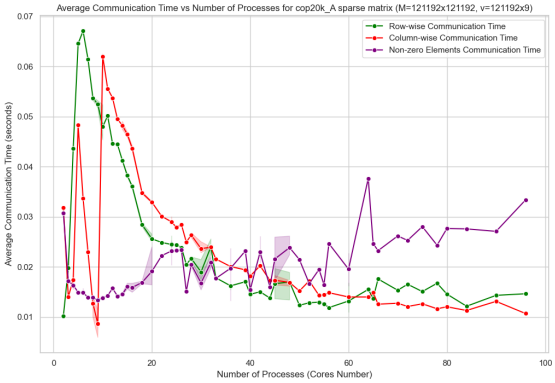

Figure 3.29: Cop20k_A matrix communication time

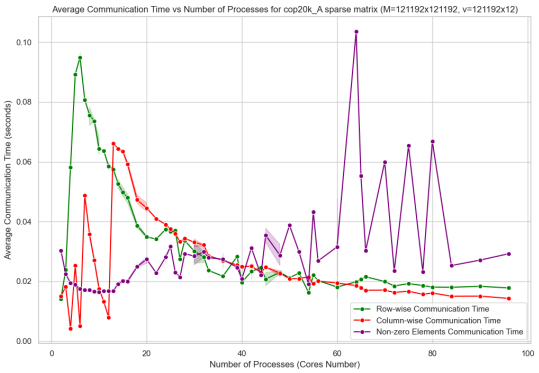

Figure 3.30: Cop20k_A matrix communication time

3.1.2.3 Computation Time

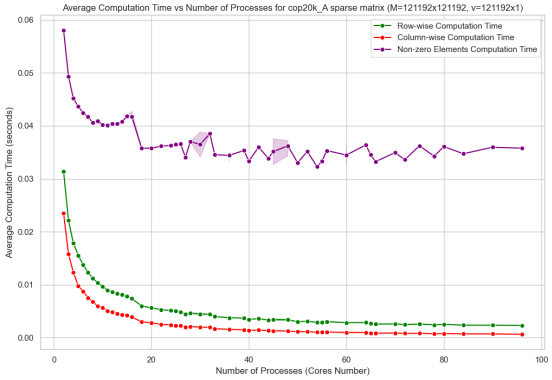

Figure 3.31: Cop20k_A matrix computation time

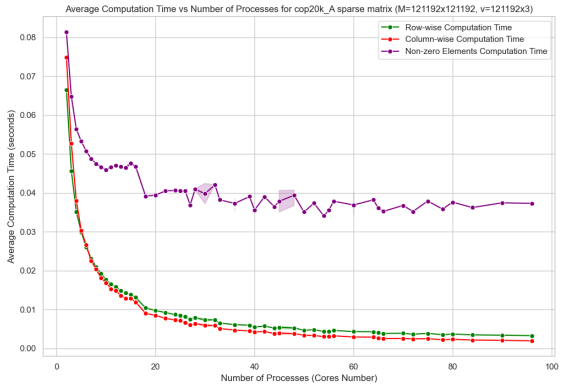

Figure 3.32: Cop20k_A matrix computation time

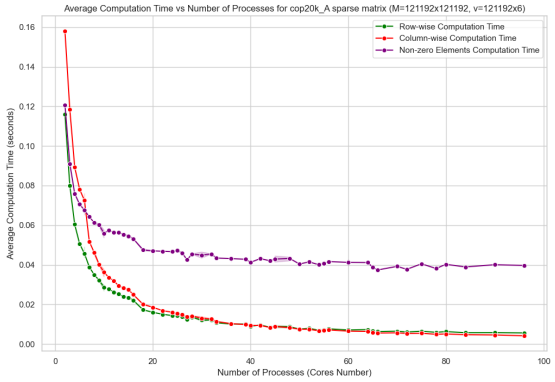

Figure 3.33: Cop20k_A matrix computation time

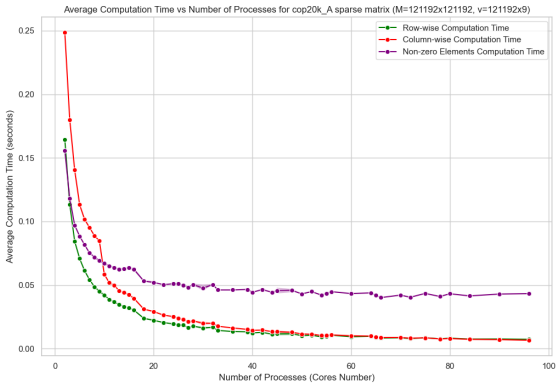

Figure 3.34: Cop20k_A matrix computation time

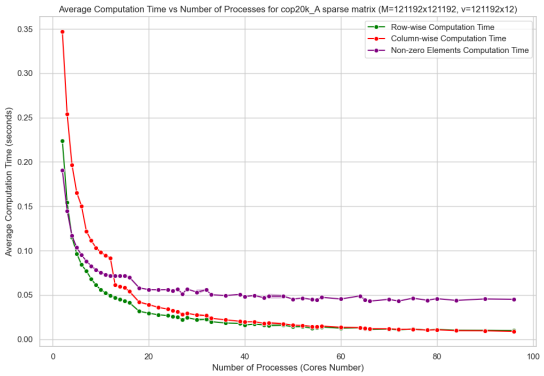

Figure 3.35: Cop20k_A matrix computation time

3.1.2.4 Performance

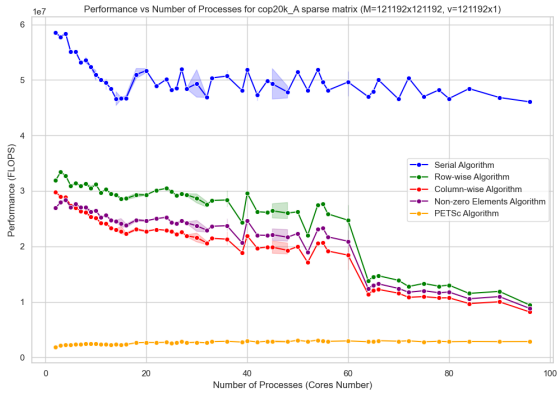

Figure 3.36: Cop20k_A matrix performance

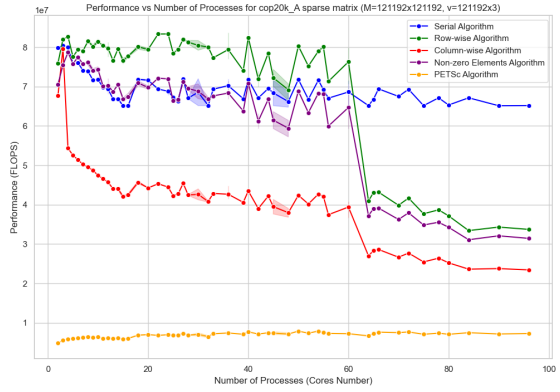

Figure 3.37: Cop20k_A matrix performance

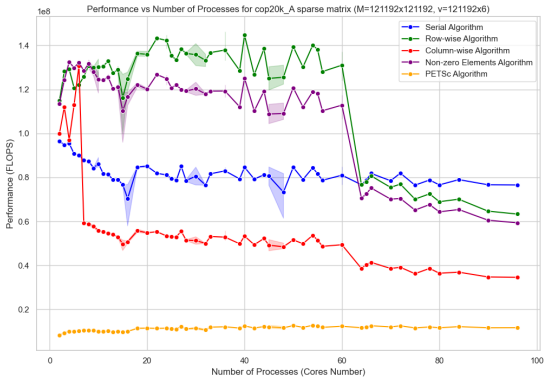

Figure 3.38: Cop20k_A matrix performance

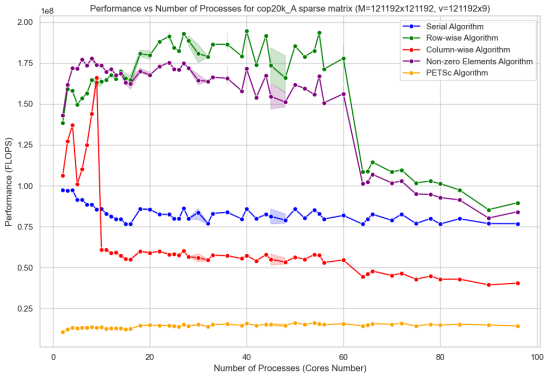

Figure 3.39: Cop20k_A matrix performance

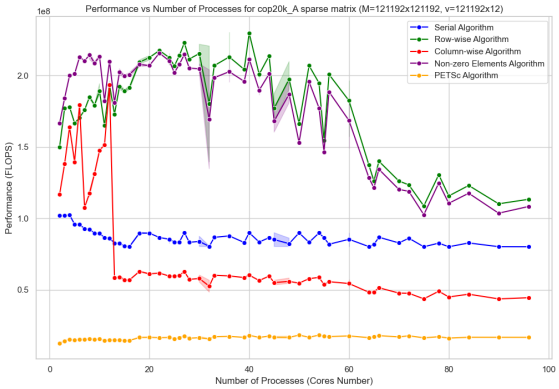

Figure 3.40: Cop20k_A matrix performance

3.2 HPC Environmental Impact

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

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

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

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

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

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

Chapter 4

Conclusion

Appendix A

Documentation

Appendix A.A Project tree

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

Appendix A.B Getting Started

To run the program, follow these steps:

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

```
            mmpicxx -o <executable_name> -I${PETSC_DIR}/include -I${
PETSC_DIR}/${PETSC_ARCH}/include -L${PETSC_DIR}/${PETSC_ARCH}/lib -lpetsc
SparseMatrixDenseVectorMultiply.cpp  main_verify.cpp utils.cpp
SparseMatrixDenseVectorMultiplyColumnWise.cpp
SparseMatrixDenseVectorMultiplyNonZeroElement.cpp
SparseMatrixDenseVectorMultiplyRowWise.cpp
```

3. Run the program using the following command:

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

Appendix A.C Detailed Features of Functions

A.C.1 Utils.h

A.C.1.1 ConvertPETScMatToFatVector

Description: Converts a PETSc matrix to a dense vector representation, termed FatVector.

A.C.1.2 areMatricesEqual

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

A.C.1.3 readMatrixMarketFile

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

A.C.1.4 generateLargeFatVector

Description: Generates a random Fat Vector with specified dimensions.

A.C.1.5 serialize and deserialize

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

A.C.2 SparseMatrixFatVectorMultiplyRowWise.h

A.C.2.1 sparseMatrixFatVectorMultiplyRowWise

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

A.C.3 SparseMatrixFatVectorMultiplyNonZeroElement.h

A.C.3.1 sparseMatrixFatVectorMultiplyNonZeroElement

Description: Executes the sparse matrix-Fat Vector multiplication using non-zero element parallel algorithm.

A.C.4 SparseMatrixFatVectorMultiplyColumnWise.h

A.C.4.1 sparseMatrixFatVectorMultiplyColumnWise

Description: Executes the sparse matrix-Fat Vector multiplication using column-wise parallel algorithm.

A.C.5 SparseMatrixFatVectorMultiply.h

A.C.5.1 sparseMatrixFatVectorMultiply

Description: Executes the sparse matrix-fat vector multiplication using a sequential algorithm.

Appendix B

Source Codes

Appendix B.A Data Structures

Data stuctures of the sparse matrix and fat vector.

```
#ifndef MATRIXDEFINITIONS_H
#define MATRIXDEFINITIONS_H

#include <vector>

//
/**
 * @brief Struct to represent a sparse matrix
 *
 * @param values  Non-zero values
 * @param colIndices  Column indices of non-zero values
 * @param rowPtr  Row pointers
 */
struct SparseMatrix
{
    std::vector<double> values;
    std::vector<int> colIndices;
    std::vector<int> rowPtr;
};

// Type definition for a Fat Vector
typedef std::vector<std::vector<double>> FatVector;

#endif
```

Appendix B.B Sequential Algorithm

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

B.B.1 Declaration File

```
#ifndef SPARSEMATRIXFATVECTORMULTIPLY_H
#define SPARSEMATRIXFATVECTORMULTIPLY_H

#include "MatrixDefinitions.h"

/**
 * @brief Function to execute the sparse matrix-fat vector multiplication using
     sequential algorithm
 *
 * @param sparseMatrix  Sparse matrix
 * @param fatVector Fat Vector
 * @param vecCols  Number of columns in the Fat Vector
 * @return FatVector  Result of the multiplication
 */
FatVector sparseMatrixFatVectorMultiply(const SparseMatrix &sparseMatrix,
                                        const FatVector &fatVector, int vecCols
                                            );

#endif
```

B.B.2 Implementation File

```
#include "SparseMatrixFatVectorMultiply.h"

/**
 * @brief Function to execute the sparse matrix-Fat Vector multiplication using
     sequential algorithm
 *
 * @param sparseMatrix  Sparse matrix
 * @param fatVector Fat Vector
 * @param vecCols  Number of columns in the Fat Vector
 * @return FatVector  Result of the multiplication
 */
FatVector sparseMatrixFatVectorMultiply(const SparseMatrix &sparseMatrix,
                                        const FatVector &fatVector, int vecCols
                                            )
{
    // Initialisation of the result vector
    FatVector result(sparseMatrix.numRows, std::vector<double>(vecCols, 0.0));
    // Iterate over the rows of the sparse matrix
    for (int i = 0; i < sparseMatrix.numRows; ++i)
    {
        // Iterate over the non-zero elements in the current row
        for (int j = sparseMatrix.rowPtr[i]; j < sparseMatrix.rowPtr[i + 1]; ++j)
        {
            // Iterate over the columns of the Fat Vector
            for (int k = 0; k < vecCols; ++k)
            {
                result[i][k] += sparseMatrix.values[j] * fatVector[sparseMatrix.
                    colIndices[j]][k]; // Compute the result
            }
        }
    }
    // Return the result
    return result;
}
```

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

```
#ifndef SPARSEMATRIXFATVECTORMULTIPLYROWWIZE_H
#define SPARSEMATRIXFATVECTORMULTIPLYROWWIZE_H

#include "MatrixDefinitions.h"
#include <iostream> // std::cout

/**
 * @brief Function to multiply a sparse matrix with a Fat Vector using row-wise distribution
 *
 * @param sparseMatrix  The sparse matrix to be multiplied
 * @param fatVector  The Fat Vector to be multiplied
 * @param vecCols  Number of columns in the Fat Vector
 * @return FatVector Result of the multiplication
 */
FatVector sparseMatrixFatVectorMultiplyRowWise(const SparseMatrix &sparseMatrix,
                                               const FatVector &fatVector,
                                               int vecCols);

#endif
```

B.C.2 Implementation File

```
#include <mpi.h>
#include "SparseMatrixFatVectorMultiplyRowWise.h"

/**
 * @brief Function to multiply a sparse matrix with a Fat Vector using row-wise distribution
 *
 * @param sparseMatrix  The sparse matrix to be multiplied
 * @param fatVector  The Fat Vector to be multiplied
 * @param vecCols  Number of columns in the Fat Vector
 * @return FatVector Result of the multiplication
 */
FatVector sparseMatrixFatVectorMultiplyRowWise(const SparseMatrix &sparseMatrix,
                                               const FatVector &fatVector,
                                               int vecCols)
{
    // Retrieve the rank and size of the MPI world
    int worldSize, worldRank;
    MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);

    // ============================ FOR DEBUGGING ONLY - START LOCAL COMPUTATION TIMER ================================
    // double computation_start = MPI_Wtime();
    // ============================ FOR DEBUGGING ONLY - START LOCAL COMPUTATION TIMER ================================

    // Distribute rows among processes
    int rowsCountPerProcess = sparseMatrix.numRows / worldSize;
                              // Number of rows per process
    int extraRows = sparseMatrix.numRows % worldSize;
                                    // Number of extra rows to be distributed among processes
```

```
28      int startRow = worldRank * rowsCountPerProcess + std::min(worldRank, extraRows)
            ; // Starting row index for the current process
29      int endRow = startRow + rowsCountPerProcess + (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 Fat 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] * fatVector[
                        colIndex][k]; // Compute the result
48              }
49          }
50      }
51
52      // ============================ FOR DEBUGGING ONLY - STOP LOCAL COMPUTATION
            TIMER ================================
53      //  double computation_end = MPI_Wtime();
54      //  double local_computation_time = computation_end - computation_start;
55      // ============================ FOR DEBUGGING ONLY - STOP LOCAL COMPUTATION
            TIMER ================================
56
57      // ============================ FOR DEBUGGING ONLY - START COMMUNICATION TIMER
            ===================================
58      //  Start timing for communication
59      //  double communication_start = MPI_Wtime();
60      // ============================ FOR DEBUGGING ONLY - START COMMUNICATION TIMER
            ===================================
61
62      // Preparation for Gather operation
63      std::vector<int> recvCounts(worldSize), displacements(worldSize);
64      if (worldRank == 0)
65      {
66          int totalSize = 0; // Total number of elements to be received
67
68          // Compute the number of elements to be received from each process
69          for (int rank = 0; rank < worldSize; ++rank)
70          {
71              int startRowThisRank = rank * rowsCountPerProcess + std::min(rank,
                    extraRows);            // Starting row index for the current
                    process
72              int endRowThisRank = startRowThisRank + rowsCountPerProcess + (rank <
                    extraRows ? 1 : 0); // Ending row index for the current process
73              recvCounts[rank] = (endRowThisRank - startRowThisRank) * vecCols;
                                        // Number of elements to be received from
                    the current process
74              displacements[rank] = totalSize;
                                                                        //
                    Displacement for the current process
75              totalSize += recvCounts[rank];
                                                                          //
                    Update the total number of elements to be received
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
                hold all the results
84      }
85      MPI_Gatherv(localResult.data(), localSize, MPI_DOUBLE,
86                  gatheredResults.data(), recvCounts.data(),
87                  displacements.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD); // Gather the
                     local results in the root process
88
89      // ============================ FOR DEBUGGING ONLY - STOP COMMUNICATION TIMER
            ==================================
90      //  double communication_end = MPI_Wtime();
91      //  double local_communication_time = communication_end - communication_start;
92      // ============================ FOR DEBUGGING ONLY - STOP COMMUNICATION TIMER
            ==================================
93
94      // ============================ FOR DEBUGGING ONLY - COLLECTING AND ANALYSING
            PERFORMANCE DATA ==================
95      //  double total_computation_time = 0.0, total_communication_time = 0.0;
96      //  MPI_Reduce(&local_computation_time, &total_computation_time, 1, MPI_DOUBLE,
             MPI_SUM, 0, MPI_COMM_WORLD);
97      //  MPI_Reduce(&local_communication_time, &total_communication_time, 1,
            MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
98      // ============================ FOR DEBUGGING ONLY - COLLECTING AND ANALYSING
            PERFORMANCE DATA ==================
99
100     // Reconstruct the final result matrix in the root process
101     FatVector finalResult;
102     if (worldRank == 0)
103     {
104         // ============================ FOR DEBUGGING ONLY - PRINTING PERFORMANCE
                DATA ================================
105         //  double avg_computation_time = total_computation_time / worldSize;
106         //  double avg_communication_time = total_communication_time / worldSize;
107         //  std::cout << "Row-wise Average Computation Time: " <<
                avg_computation_time << std::endl;
108         //  std::cout << "Row-wise Average Communication Time: " <<
                avg_communication_time << std::endl;
109         // ============================ FOR DEBUGGING ONLY - PRINTING PERFORMANCE
                DATA ================================
110
111         finalResult.resize(sparseMatrix.numRows, std::vector<double>(vecCols, 0.0))
                ; // Resize the final result matrix
112
113         // Iterate over the rows of the final result
114         for (int i = 0, index = 0; i < sparseMatrix.numRows; ++i)
115         {
116             // Iterate over the columns of the final result
117             for (int j = 0; j < vecCols; ++j, ++index)
118             {
119                 finalResult[i][j] = gatheredResults[index]; // Copy the element of
                        the final result
120             }
121         }
122     }
123
124     // Return the final result
125     return (worldRank == 0) ? finalResult : FatVector{};
126 }
```

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

```
#ifndef SPARSEMATRIXFATVECTORMULTIPLYCOLUMNWIZE_H
#define SPARSEMATRIXFATVECTORMULTIPLYCOLUMNWIZE_H

#include "MatrixDefinitions.h"
#include <iostream> // std::cout

/**
 * @brief Function to execute the sparse matrix-Fat Vector multiplication using column-wise parallel algorithm
 *
 * @param sparseMatrix Sparse matrix
 * @param fatVector Fat Vector
 * @param vecCols Number of columns in the Fat Vector
 * @return FatVector Result of the multiplication
 */
FatVector sparseMatrixFatVectorMultiplyColumnWise(const SparseMatrix &sparseMatrix, const FatVector &fatVector, int vecCols);

#endif
```

B.D.2 Implementation File

```
#include <mpi.h>
#include "SparseMatrixFatVectorMultiplyColumnWise.h"
#include <numeric> // std::accumulate

/**
 * @brief Function to execute the sparse matrix-Fat Vector multiplication using column-wise parallel algorithm
 *
 * @param sparseMatrix Sparse matrix
 * @param fatVector Fat Vector
 * @param vecCols Number of columns in the Fat Vector
 * @return FatVector Result of the multiplication
 */
FatVector sparseMatrixFatVectorMultiplyColumnWise(const SparseMatrix &sparseMatrix, const FatVector &fatVector, int vecCols)
{
    // Retrieve the rank and size of the MPI world
    int worldSize, worldRank;
    MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);

    // ========================== FOR DEBUGGING ONLY - START LOCAL COMPUTATION TIMER ================================
    // double computation_start = MPI_Wtime();
    // ========================== FOR DEBUGGING ONLY - START LOCAL COMPUTATION TIMER ================================

    // Distribute columns among processes
    int colsPerProcess = vecCols / worldSize;                                                                    // Number of columns per process
    int extraCols = vecCols % worldSize;

        // Number of extra columns to be distributed among processes
```

```
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] * fatVector[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
            TIMER ================================
51      // double computation_end = MPI_Wtime();
52      // double local_computation_time = computation_end - computation_start;
53      // ============================ FOR DEBUGGING ONLY - STOP LOCAL COMPUTATION
            TIMER ================================
54
55      // ============================ FOR DEBUGGING ONLY - START COMMUNICATION TIMER
            ==================================
56      // Start timing for communication
57      // double communication_start = MPI_Wtime();
58      // ============================ FOR DEBUGGING ONLY - START COMMUNICATION TIMER
            ==================================
59
60      // Preparation for Gather operation
61      std::vector<int> recvCounts(worldSize), displacements(worldSize); // Number of
            elements to be received from each process, Displacement for each process
62      if (worldRank == 0)
63      {
64          // Compute the number of elements to be received from each process
65          int displacement = 0;
66          for (int i = 0; i < worldSize; ++i)
67          {
68              int startColThisRank = i * colsPerProcess;

                    // Starting column index for the current process
69              int endColThisRank = (i != worldSize - 1) ? startColThisRank +
                    colsPerProcess : startColThisRank + colsPerProcess + extraCols; //
                    Ending column index for the current process
70              recvCounts[i] = sparseMatrix.numRows * (endColThisRank -
                    startColThisRank);
                                                                        // Number of
                    elements to be received from the current process
71              displacements[i] = displacement;

                    // Displacement for the current process
72              displacement += recvCounts[i];
```

```
                displacement += recvCounts[rank - 1]; // Update the displacement
            }
        }

        // Gather all local results into the root process
        std::vector<double> gatheredResults;
        if (worldRank == 0)
        {
            gatheredResults.resize(std::accumulate(recvCounts.begin(), recvCounts.end()
                , 0)); // Resize the vector to hold the final result
        }
        MPI_Gatherv(localResult.data(), localSize, MPI_DOUBLE,
                    gatheredResults.data(), recvCounts.data(),
                    displacements.data(), MPI_DOUBLE, 0, MPI_COMM_WORLD); // Gather the
                         local results in the root process

        // ============================ FOR DEBUGGING ONLY - STOP COMMUNICATION TIMER
            ====================================
        // double communication_end = MPI_Wtime();
        // double local_communication_time = communication_end - communication_start;
        // ============================ FOR DEBUGGING ONLY - STOP COMMUNICATION TIMER
            ====================================

        // ============================ FOR DEBUGGING ONLY - COLLECTING AND ANALYSING
            PERFORMANCE DATA ==================
        // double total_computation_time = 0.0, total_communication_time = 0.0;
        // MPI_Reduce(&local_computation_time, &total_computation_time, 1, MPI_DOUBLE,
            MPI_SUM, 0, MPI_COMM_WORLD);
        // MPI_Reduce(&local_communication_time, &total_communication_time, 1,
            MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
        // ============================ FOR DEBUGGING ONLY - COLLECTING AND ANALYSING
            PERFORMANCE DATA ==================

        // Reconstruct the final result matrix in the root process
        FatVector finalResult;
        if (worldRank == 0)
        {
            // ============================ FOR DEBUGGING ONLY - PRINTING PERFORMANCE
                DATA ================================
            // double avg_computation_time = total_computation_time / worldSize;
            // double avg_communication_time = total_communication_time / worldSize;
            // std::cout << "Column-wise Average Computation Time: " <<
                avg_computation_time << std::endl;
            // std::cout << "Column-wise Average Communication Time: " <<
                avg_communication_time << std::endl;
            // ============================ FOR DEBUGGING ONLY - PRINTING PERFORMANCE
                DATA ================================

            // Reconstruct the final result matrix
            finalResult.resize(sparseMatrix.numRows, std::vector<double>(vecCols, 0.0))
                ; // Resize the final result matrix
            int resultIndex = 0;
            // Iterate over the processes
            for (int rank = 0; rank < worldSize; ++rank)
            {
                int numColsThisRank = (rank != worldSize - 1) ? colsPerProcess :
                    colsPerProcess + extraCols; // Number of columns assigned to the
                    current process
                int startColThisRank = rank * colsPerProcess;
                                                                    // Starting column
                    index for the current process

                // Iterate over the rows of the sparse matrix
                for (int row = 0; row < sparseMatrix.numRows; ++row)
                {
                    // Iterate over the columns assigned to the current process
                    for (int col = 0; col < numColsThisRank; ++col)
                    {
                        finalResult[row][startColThisRank + col] = gatheredResults[
                            resultIndex++]; // Reconstruct the final result matrix
```

```
                }
            }
        }
    }

    // Return the final result
    return (worldRank == 0) ? finalResult : FatVector{};
}
```

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

```
#ifndef SPARSEMATRIXFATVECTORMULTIPLYNONZEROELEMENT_H
#define SPARSEMATRIXFATVECTORMULTIPLYNONZEROELEMENT_H

#include "MatrixDefinitions.h"
#include <iostream> // std::cout

/**
 * @brief Function to execute the sparse matrix-Fat Vector multiplication using non-zero element parallel algorithm
 *
 * @param sparseMatrix  Sparse matrix
 * @param fatVector  Fat Vector
 * @param vecCols  Number of columns in the Fat Vector
 * @return FatVector  Result of the multiplication
 */
FatVector sparseMatrixFatVectorMultiplyNonZeroElement(const SparseMatrix &sparseMatrix, const FatVector &fatVector, int vecCols);

#endif
```

B.E.2 Implementation File

```
#include "SparseMatrixFatVectorMultiplyNonZeroElement.h"
#include <mpi.h>

/**
 * @brief Function to execute the sparse matrix-Fat Vector multiplication using non-zero element parallel algorithm
 *
 * @param sparseMatrix  Sparse matrix
 * @param fatVector  Fat Vector
 * @param vecCols  Number of columns in the Fat Vector
 * @return FatVector  Result of the multiplication
 */
FatVector sparseMatrixFatVectorMultiplyNonZeroElement(const SparseMatrix &sparseMatrix, const FatVector &fatVector, int vecCols)
{
    // Retrieve the rank and size of the MPI world
    int worldSize, worldRank;
    MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
    MPI_Comm_rank(MPI_COMM_WORLD, &worldRank);

    // ============================ FOR DEBUGGING ONLY - START LOCAL COMPUTATION TIMER ================================
```

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

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

Appendix B.F Utility Functions

Utility functions used by the main file.

B.F.1 Declaration File

```
#ifndef UTILS_H
#define UTILS_H

#include <iostream> // std::cout
#include <vector>   // std::vector
#include <cstdlib>  // rand() and srand()
#include <ctime>    // time()
#include <mpi.h>
#include <petsc.h>
#include <fstream>   // std::ifstream
#include <string>    // std::string
#include <sstream>   // std::stringstream
#include <utility>   // std::pair
#include <algorithm> // std::sort
#include <stdexcept> // std::runtime_error
#include <cmath>     // std::fabs
#include "MatrixDefinitions.h"

/**
 * Method to convert a PETSc matrix to a fat vector
 * @param C PETSc matrix
 * @return FatVector Dense vector
 */
FatVector ConvertPETScMatToFatVector(Mat C);

/**
 * Method to compare two matrices
 * @param mat1 First matrix
 * @param mat2 Second matrix
 * @param tolerance Tolerance for comparison
 * @return bool True if the matrices are equal, false otherwise
 */
bool areMatricesEqual(const FatVector &mat1, const FatVector &mat2, double
    tolerance);

/**
 * Method to read a matrix from a Matrix Market file
 * @param filename Name of the file
 * @return SparseMatrix Sparse matrix
 */
SparseMatrix readMatrixMarketFile(const std::string &filename);

/**
 * Method to generate a random fat vector
 * @param n Number of rows
 * @param m Number of columns
 * @return FatVector Dense vector
 */
FatVector generateLargeFatVector(int n, int k);

/**
 * @brief Method to serialize a FatVector to a flat array
 * @param denseVec Dense vector to serialize
 * @return std::vector<double> Flat array containing the serialized data
 */
std::vector<double> serialize(const FatVector &denseVec);

/**
 * @brief Method to deserialize a flat array to a FatVector
 * @param flat Flat array to deserialize
 * @param rows Number of rows in the fat vector
```

```
 * @param cols Number of columns in the fat vector
 * @return FatVector Dense vector
 */
FatVector deserialize(const std::vector<double> &flat, int rows, int cols);

#endif
```

B.F.2 Implementation File

```
#include "utils.h"

/**
 * Method to convert a PETSc matrix to a fat vector
 * @param C PETSc matrix
 * @return FatVector Dense vector
 */
FatVector ConvertPETScMatToFatVector(Mat C)
{
    PetscInt m, n;          // Number of rows and columns in the matrix
    MatGetSize(C, &m, &n); // Get the number of rows and columns in the matrix

    FatVector denseVec(m, std::vector<double>(n, 0.0)); // Dense vector to hold the
        matrix

    // Iterate over the rows of the matrix
    for (int i = 0; i < m; ++i)
    {
        // Iterate over the columns of the matrix
        for (int j = 0; j < n; j++)
        {
            PetscScalar value;                      // Value of the element
            MatGetValue(C, i, j, &value);           // Get the value of the element
            denseVec[i][j] = PetscRealPart(value); // Copy the value of the element
        }
    }

    // Return the fat vector
    return denseVec;
}

/**
 * Method to compare two matrices
 * @param mat1 First matrix
 * @param mat2 Second matrix
 * @param tolerance Tolerance for comparison
 * @return bool True if the matrices are equal, false otherwise
 */
bool areMatricesEqual(const FatVector &mat1, const FatVector &mat2, double
    tolerance)
{
    // Check if the matrices have the same dimensions
    if (mat1.size() != mat2.size())
        return false;

    // Iterate over the rows of the matrices
    for (size_t i = 0; i < mat1.size(); ++i)
    {
        // Check if the rows have the same dimensions
        if (mat1[i].size() != mat2[i].size())
            return false;

        // Iterate over the columns of the matrices
        for (size_t j = 0; j < mat1[i].size(); ++j)
        {
            // Check if the elements are equal
            if (std::fabs(mat1[i][j] - mat2[i][j]) > tolerance)
            {
```

```
                return false; // Matrices are not equal
            }
        }
    }

    return true; // Matrices are equal
}

/**
 * Method to read a matrix from a Matrix Market file
 * @param filename Name of the file
 * @return SparseMatrix Sparse matrix
 */
SparseMatrix readMatrixMarketFile(const std::string &filename)
{
    std::ifstream file(filename); // Input file stream

    // Check if the file was opened successfully
    if (!file.is_open())
    {
        throw std::runtime_error("Unable to open file: " + filename);
    }

    std::string line;                           // String to hold the current line
    bool isSymmetric = false, isPattern = false; // Flags to indicate if the matrix
        is symmetric or pattern only

    // Skip the comments
    while (std::getline(file, line))
    {
        // Check if the line is a comment
        if (line[0] == '%')
        {
            // Check if the line contains the word "symmetric"
            if (line.find("symmetric") != std::string::npos)
            {
                isSymmetric = true; // Set the symmetric flag
            }

            // Check if the line contains the word "pattern"
            if (line.find("pattern") != std::string::npos)
            {
                isPattern = true; // Set the pattern flag
            }
        }
        else
        {
            break; // First non-comment line reached, break out of the loop
        }
    }

    // Read the matrix dimensions
    int numRows, numCols, nonZeros;                          // Number of rows,
        columns and non-zero elements in the matrix
    std::stringstream(line) >> numRows >> numCols >> nonZeros; // Read the
        dimensions from the line

    // Check if the file was read successfully
    if (!file)
    {
        throw std::runtime_error("Failed to read matrix dimensions from file: " +
            filename);
    }

    SparseMatrix matrix;                                                  // Sparse
        matrix to hold the data
    matrix.rowPtr.resize(numRows + 1, 0);                                 // Resize
        the row pointer vector
    std::vector<std::vector<std::pair<int, double>>> tempRows(numRows); //
        Temporary vector to hold the data
```

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

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

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

Appendix B.G Main File

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

```
1  #include "utils.h"                                          // Utility functions
2  #include "SparseMatrixFatVectorMultiply.h"                 // Sequential algorithm
3  #include "SparseMatrixFatVectorMultiplyRowWise.h"          // Parallel algorithm (row
       -wise)
4  #include "SparseMatrixFatVectorMultiplyColumnWise.h"       // Parallel algorithm (
       column-wise)
5  #include "SparseMatrixFatVectorMultiplyNonZeroElement.h" // Parallel algorithm (non
       -zero element)
6
7  int main(int argc, char *argv[])
8  {
9      //
           ------------------------------------------------------------------------------------------
10     // ============================================== 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 Fat Vector
37     SparseMatrix M;
38     FatVector v;
39
40     // Declare the result of the serial multiplication
41     FatVector resultSerial;
42
43     // Declare the data for broadcasting the sparse matrix and Fat 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 Fat
            Vector ============================
52      //
            ----------------------------------------------------------------------------------------------
53      if (worldRank == 0)
54      {
55          std::cout << "World size: " << worldSize << std::endl;   // Print the
                number of processes
56          std::cout << "Sparse matrix: " << filename << std::endl; // Print the name
                of the Matrix Market file
57
58          // Read the sparse matrix from the Matrix Market file
59          M = readMatrixMarketFile(filename);
60          std::cout << "Matrix size: " << M.numRows << "x" << M.numCols << std::endl;
61
62          // Generate a random Fat Vector
63          v = generateLargeFatVector(M.numCols, k);
64          std::cout << "Vector size: " << M.numCols << "x" << k << std::endl;
65
66          // Prepare the data for broadcasting
67          flatData = serialize(v);    // Serialize the Fat Vector
68          dataSize = flatData.size(); // Size of the serialized data
69      }
70
71      //
            ----------------------------------------------------------------------------------------------
72      // ==================================== EXECUTE THE SERIAL MULTIPLICATION
            ======================================
73      //
            ----------------------------------------------------------------------------------------------
74      if (worldRank == 0)
75      {
76          // Execute the serial multiplication
77          startTime = MPI_Wtime();
78          resultSerial = sparseMatrixFatVectorMultiply(M, v, k);
79          endTime = MPI_Wtime();
80          std::cout << "Serial Algo Execution time: " << (endTime - startTime)
81                    << std::endl;
82
83          // FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS OF THE RESULT
84          // std::cout << "Result: " << std::endl;
85          // for (int i = 0; i < 10; ++i)
86          // {
87          //     for (int j = 0; j < k; ++j)
88          //     {
89          //         std::cout << resultSerial[i][j] << " ";
90          //     }
91          //     std::cout << std::endl;
92          // }
93      }
94
95      //
            ----------------------------------------------------------------------------------------------
96      // ================================ BROADCAST THE SPARSE MATRIX AND Fat Vector
            ================================
97      //
            ----------------------------------------------------------------------------------------------
98
99      // Wait for the main process to finish the serial multiplication
100     MPI_Barrier(MPI_COMM_WORLD);
```

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

```
157     // ======================== EXECUTE THE PARALLEL MULTIPLICATION (ROW-WISE)
            ====================================
158     //
            ----------------------------------------------------------------------------------------
159
160     // Execute the parallel multiplication (row-wise)
161     startTime = MPI_Wtime();
162     FatVector resultRowWise = sparseMatrixFatVectorMultiplyRowWise(M, v, k);
163     endTime = MPI_Wtime();
164
165     // Only the main process prints the parallel execution time
166     if (worldRank == 0)
167     {
168         std::cout << "Row-wise Execution time: " << (endTime - startTime)
169                   << std::endl;
170
171         // ============================ FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS
                ==============================
172         // std::cout << "Result: " << std::endl;
173         // for (int i = 0; i < 10; ++i)
174         // {
175         //     for (int j = 0; j < k; ++j)
176         //     {
177         //         std::cout << resultRowWise[i][j] << " ";
178         //     }
179         //     std::cout << std::endl;
180         // }
181         // ============================ FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS
                ==============================
182
183         // Compare the results of the serial and parallel multiplications
184         if (areMatricesEqual(resultSerial, resultRowWise, 1e-6)) // Tolerance = 1e
                -6
185         {
186             std::cout << "Row-wise: Results are the same!"
187                       << std::endl;
188         }
189         else
190         {
191             std::cout << "Row-wise: Results are different!"
192                       << std::endl;
193         }
194     }
195
196     //
            ----------------------------------------------------------------------------------------
197     // ======================== EXECUTE THE PARALLEL MULTIPLICATION (COLUMN-WISE)
            ====================================
198     //
            ----------------------------------------------------------------------------------------
199
200     // Wait for all processes to finish the parallel multiplication (row-wise)
201     MPI_Barrier(MPI_COMM_WORLD);
202
203     // Execute the parallel multiplication (column-wise)
204     startTime = MPI_Wtime();
205     FatVector resultColumnWise = sparseMatrixFatVectorMultiplyColumnWise(M, v, k);
206     endTime = MPI_Wtime();
207
208     // Only the main process prints the parallel execution time
209     if (worldRank == 0)
210     {
211         std::cout << "Column-wise Execution time: " << (endTime - startTime)
212                   << std::endl;
213
214         // ============================ 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 =
                1e-6
228         {
229             std::cout << "Column-wise: Results are the same!"
230                       << std::endl;
231         }
232         else
233         {
234             std::cout << "Column-wise: Results are different!"
235                       << std::endl;
236         }
237     }
238
239     //
            ----------------------------------------------------------------------------------------------------
240     // ============================ EXECUTE THE PARALLEL MULTIPLICATION (NON-ZERO
            ELEMENT) ============================
241     //
            ----------------------------------------------------------------------------------------------------
242
243     // Wait for all processes to finish the parallel multiplication (column-wise)
244     MPI_Barrier(MPI_COMM_WORLD);
245
246     // Execute the parallel multiplication (non-zero element)
247     startTime = MPI_Wtime();
248     FatVector resultNonZeroElement = sparseMatrixFatVectorMultiplyNonZeroElement(M,
            v, k);
249     endTime = MPI_Wtime();
250
251     // Only the main process prints the parallel execution time
252     if (worldRank == 0)
253     {
254         std::cout << "Non-zero Elements Execution time: " << (endTime - startTime)
255                   << std::endl;
256
257         // ============================ FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS
                ==============================
258         // std::cout << "Result: " << std::endl;
259         // for (int i = 0; i < 10; ++i)
260         // {
261         //     for (int j = 0; j < k; ++j)
262         //     {
263         //         std::cout << resultNonZeroElement[i][j] << " ";
264         //     }
265         //     std::cout << std::endl;
266         // }
267         // ============================ FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS
                ==============================
268
269         // Compare the results of the serial and parallel multiplications
270         if (areMatricesEqual(resultSerial, resultNonZeroElement, 1e-6)) //
                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 (
        PETSc) ==============================
284     //
        ---------------------------------------------------------------------------------------------
285
286     // Wait for all processes to finish the parallel multiplication (non-zero
        element)
287     MPI_Barrier(MPI_COMM_WORLD);
288
289     // Declare the PETSc matrix
290     Mat A, B, C;
291
292     // ========================== FOR DEBUGGING ONLY - START PETSCS SETUP TIMER
        ========================================
293     // startTime = MPI_Wtime();
294     // ========================== FOR DEBUGGING ONLY - START PETSCS SETUP TIMER
        ========================================
295
296     // Create a parallel matrix to store the sparse matrix
297     MatCreate(PETSC_COMM_WORLD, &A);
298     MatSetSizes(A, PETSC_DECIDE, PETSC_DECIDE, M.numRows, M.numCols);
299     MatSetType(A, MATMPIAIJ);
300     MatSetUp(A);
301     // Fill the PETSc matrix with the values from the sparse matrix
302     if (worldRank == 0)
303     {
304         for (int i = 0; i < M.numRows; ++i)
305         {
306             for (int j = M.rowPtr[i]; j < M.rowPtr[i + 1]; ++j)
307             {
308                 MatSetValue(A, i, M.colIndices[j], M.values[j], INSERT_VALUES);
309             }
310         }
311     }
312     // Assemble the PETSc matrix
313     MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
314     MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
315
316     // Create a parallel matrix to store the Fat Vector
317     MatCreate(PETSC_COMM_WORLD, &B);
318     MatSetSizes(B, PETSC_DECIDE, PETSC_DECIDE, M.numCols, k);
319     MatSetType(B, MATDENSE);
320     MatSetUp(B);
321     // Fill the PETSc matrix B with values from the dense matrix v
322     if (worldRank == 0)
323     {
324         for (int i = 0; i < M.numCols; ++i)
325         {
326             for (int j = 0; j < k; ++j)
327             {
328                 MatSetValue(B, i, j, v[i][j], INSERT_VALUES);
329             }
330         }
331     }
332     // Assemble the PETSc matrix
333     MatAssemblyBegin(B, MAT_FINAL_ASSEMBLY);
334     MatAssemblyEnd(B, MAT_FINAL_ASSEMBLY);
335
```

```
336     // ============================ FOR DEBUGGING ONLY - STOP PETSCS SETUP TIMER
            ========================================
337     // 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
355     // ============================ FOR DEBUGGING ONLY - START PETSCS CONVERSION
            TIMER ==================================
356     // startTime = MPI_Wtime();
357     // ============================ FOR DEBUGGING ONLY - START PETSCS CONVERSION
            TIMER ==================================
358
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 FatVector
366         FatVector globalMatrix = ConvertPETScMatToFatVector(CSeq);
367
368         // ============================ FOR DEBUGGING ONLY - STOP PETSCS CONVERSION
                TIMER ==================================
369         // endTime = MPI_Wtime();
370         // std::cout << "PETSc Conversion time: " << (endTime - startTime) << std::
                endl;
371         // ============================ FOR DEBUGGING ONLY - STOP PETSCS CONVERSION
                TIMER ==================================
372
373         // ============================ FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS
                ==============================
374         // std::cout << "Result: " << std::endl;
375         // for (int i = 0; i < 10; ++i)
376         // {
377         //     for (int j = 0; j < k; ++j)
378         //     {
379         //         std::cout << globalMatrix[i][j] << " ";
380         //     }
381         //     std::cout << std::endl;
382         // }
383         // ============================ FOR DEBUGGING ONLY - PRINT 10 FIRST ELEMENTS
                ==============================
384
385         // Compare the results of the serial and PETSc multiplications
386         if (areMatricesEqual(resultSerial, globalMatrix, 1e-6)) // Tolerance = 1e-6
387         {
388             std::cout << "PETSc: Results are the same!"
389                       << std::endl;
390         }
391         else
392         {
393             std::cout << "PETSc: Results are different!"
394                       << std::endl;
```

```
        }
    }

    // Free the memory
    MatDestroy(&A);
    MatDestroy(&B);
    MatDestroy(&C);
    MatDestroy(&CSeq);

    // Finalise MPI and PETSc
    PetscFinalize();
    MPI_Finalize();

    return 0;
}
```

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 mpiprocs MUST be the same value as ncpus.
##
## If more than 16 cpus are required then select multiple chunks of 16
## e.g. 16 CPUs: select=1:ncpus=16:mpiprocs=16
##      32 CPUs: select=2:ncpus=16:mpiprocs=16
##      ..etc..
##
#PBS -l select=2:ncpus=16:mpiprocs=16
##
## STEP 3:
##
## Select the correct queue by modifying the #PBS -q line below
##
## half_hour    -  30 minutes
## one_hour     -   1 hour
## three_hour   -   3 hours
## six_hour     -   6 hours
## half_day     -  12 hours
## one_day      -  24 hours
## two_day      -  48 hours
## five_day     - 120 hours
## ten_day      - 240 hours (by special arrangement)
##
#PBS -q half_hour
```

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

mpirun -genvall -hostfile mpi_nodes.$$ -np ${cpus} ../my_program_final_debug ${
    K_VALUE} ${MATRIX_PATH}

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

B.H.2 Batch Test Script

Bash script to submit multiple MPI jobs to the cluster.

```
#!/bin/bash

# Script to submit a batch of jobs to the cluster

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

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

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

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

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

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

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

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

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

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

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

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

                        # Wait for 1 second
                        sleep 1
                    done

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

B.H.3 Get CSV Script

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

```
#!/bin/bash

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

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

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

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

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

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

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

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

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

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

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

echo "The data was successfully written in $output_csv"
```