



Alexis Balayre

Small Scale Parallel Programming 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

This paper explores the effectiveness of different parallelization strategies for multiplying sparse matrices by fat vectors, focusing on the use of MPI in High Performance Computing (HPC). It compares the performance of sequential and parallel methods, including row-by-row, column-by-column, and non-zero element approaches. The results highlight the implications of each strategy on execution time and efficiency, while considering the environmental impact of HPC, suggesting avenues towards more sustainable computing systems.

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.

Chapter 2

Methodology

2.1 Problem Statement

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

The matrix M is defined as:

$$M = \begin{pmatrix} m_{11} & m_{12} & \cdots & m_{1n} \\ m_{21} & m_{22} & \cdots & m_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ m_{m1} & m_{m2} & \cdots & m_{mn} \end{pmatrix} \quad (2.1)$$

where most elements of M are zeros.

The vector v is defined as:

$$v = \begin{pmatrix} v_{11} & v_{12} & \cdots & v_{1k} \\ v_{21} & v_{22} & \cdots & v_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n1} & v_{n2} & \cdots & v_{nk} \end{pmatrix} \quad (2.2)$$

2.2 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.

2.2.1 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, 2, 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 ??.

2.2.2 Fat Vector

Unlike a hollow matrix, a fat vector (illustrated by equation 2.2) 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 ??.

2.3 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 ??.

2.3.1 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 $Result[i][l]$.

2.3.2 Temporal Complexity Analysis

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

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

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

$$T(n) = O(zk) \tag{2.3}$$

2.4 Line-Based Parallelism

This algorithm distributes the rows of the sparse matrix across multiple processes for parallel computation in a line-based manner. The implementation is detailed in Appendix ??.

2.4.1 Algorithm Flow

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

Require: M is an $m \times n$ sparse matrix
Require: v is an $n \times k$ fat vector
Require: $worldSize$ is the number of processes
Require: $worldRank$ is the rank of the current process
Ensure: $finalResult$ is an $m \times k$ matrix, result of $M \times v$

```

rowsPerProcess  $\leftarrow m / worldSize$ 
extraRows  $\leftarrow m \bmod worldSize$ 
startRow  $\leftarrow worldRank \times rowsPerProcess + \min(worldRank, extraRows)$ 
endRow  $\leftarrow startRow + rowsPerProcess$ 
if  $worldRank < extraRows$  then
    endRow  $\leftarrow endRow + 1$ 
end if
Initialise  $localResult$  with zeros of size  $(endRow - startRow) \times k$ 
for  $i \leftarrow startRow$  to  $endRow - 1$  do
    for each non-zero element  $(j, value)$  in row  $i$  of  $M$  do
        for  $k \leftarrow 0$  to  $k - 1$  do
            localIndex  $\leftarrow (i - startRow) \times k + k$ 
            localResult[localIndex]  $\leftarrow localResult[localIndex] + value \times v[j][k]$ 
        end for
    end for
end for
if  $worldRank == 0$  then
    Prepare  $recvCounts$  and  $displacements$  for gathering
end if
MPI_Gatherv(localResult, ...)
if  $worldRank == 0$  then
    Reassemble  $finalResult$  from all  $localResults$ 
    return  $finalResult$ 
end if

```

The algorithmic flow can be explicitly detailed by:

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

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

2.4.2 Time Complexity Analysis

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

2.4.2.1 Computational Complexity

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

$$T_{\text{comp}} = O\left(\frac{z \times k}{p}\right) \quad (2.4)$$

2.4.2.2 Communication Complexity

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

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

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

$$T_{\text{comm}} = p \times \alpha + \beta \times \frac{m}{p} \times k \quad (2.5)$$

2.4.2.3 Final Result Reconstruction

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

2.4.2.4 Overall Time Complexity

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

$$T_{\text{overall}} = T_{\text{comp}} + T_{\text{comm}} = O\left(\frac{z \times k}{p}\right) + p \times \alpha + \beta \times \frac{m}{p} \times k + O(m \times k) \quad (2.6)$$

2.4.3 Performance Analysis

2.4.3.1 Performance Dependence

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

2.4.3.2 Expected Performance

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

2.5 Column-Wise Parallelism

This algorithm distributes the columns of the fat vector across multiple processes for parallel computation in a column-based manner. The implementation is detailed in Appendix ??.

2.5.1 Algorithm Flow

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

Require: M is an $m \times n$ sparse matrix
Require: v is an $n \times k$ fat vector
Require: $worldSize$ is the number of processes
Require: $worldRank$ is the rank of the current process
Ensure: $finalResult$ is an $m \times k$ matrix, result of $M \times v$

```

 $colsPerProcess \leftarrow k / worldSize$ 
 $extraCols \leftarrow k \bmod worldSize$ 
 $startCol \leftarrow worldRank \times colsPerProcess$ 
 $endCol \leftarrow startCol + colsPerProcess$ 
if  $worldRank == worldSize - 1$  then
     $endCol \leftarrow endCol + extraCols$ 
end if
 $localSize \leftarrow m \times (endCol - startCol)$ 
Initialise  $localResult$  with zeros of size  $localSize$ 
for  $col \leftarrow startCol$  to  $endCol - 1$  do
    for  $row \leftarrow 0$  to  $m - 1$  do
         $sum \leftarrow 0$ 
        for each non-zero element  $(i, value)$  in row  $row$  of  $M$  do
             $sum \leftarrow sum + value \times v[i][col]$ 
        end for
         $localResult[row][col - startCol] \leftarrow sum$ 
    end for
end for
if  $worldRank == 0$  then
    Initialise  $finalResult$  with zeros of size  $m \times k$ 
end if
Gather  $localResult$  from all processes to  $finalResult$  at root
if  $worldRank == 0$  then
    State Reassemble  $finalResult$  from gathered  $localResults$ 
    return  $finalResult$ 
end if

```

The algorithmic flow can be explicitly detailed by:

1. **Initialisation:** Obtain MPI world size and rank to determine each process's role.

2. **Column Distribution:** Calculate the number of columns each process will handle, distributing any extra columns to the last processes, and define the start and end column indices for each process.
3. **Local Computation:** Each process computes a portion of the multiplication result for its assigned columns, iterating through the sparse matrix rows and the relevant columns of the fat vector.
4. **Gather Results:** Use `MPI_Gatherv` to collect the local results from all processes into a single result vector at the root process.
5. **Final Result Reconstruction:** The root process reassembles the gathered results into the final fat vector matrix, ensuring the elements are correctly positioned according to their original indices.

2.5.2 Temporal Complexity Analysis

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

2.5.2.1 Computation Time Complexity

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

$$T_{\text{comp}} = O\left(\frac{k \times z}{p}\right) \quad (2.7)$$

2.5.2.2 Communication Time Complexity

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

$$T_{\text{comm}} = p \times \alpha + \beta \times m \times \frac{k}{p} \quad (2.8)$$

2.5.2.3 Final Result Reconstruction

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

2.5.2.4 Overall Time Complexity

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

$$T_{\text{overall}} = T_{\text{comp}} + T_{\text{comm}} = O\left(\frac{k \times z}{p}\right) + p \times \alpha + \beta \times m \times \frac{k}{p} + O(m \times k) \quad (2.9)$$

2.5.3 Performance Analysis

2.5.3.1 Performance Dependence

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

2.5.3.2 Expected Performance

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

2.6 Non-Zero Element Parallelism

This algorithm distributes the non-zero elements of the sparse matrix across multiple processes for parallel computation. The implementation is detailed in Appendix ??.

2.6.1 Algorithm Flow

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

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

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

Require: $worldSize$ is the number of processes

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

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

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

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

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

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

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

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

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

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

end for

end for

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

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

 Reconstruct $finalResult$ from $flatFinalResult$

return $finalResult$

end if

The algorithmic flow can be explicitly detailed by:

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

2.6.2 Temporal Complexity Analysis

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

2.6.3 Computation Complexity

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

$$T_{\text{comp}} = O\left(\frac{z \times k}{p}\right) \quad (2.10)$$

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

2.6.4 Communication Complexity

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

$$T_{\text{comm}} = p \times \alpha + \beta \times m \times \frac{k}{p} \quad (2.11)$$

where:

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

2.6.4.1 Final Result Reconstruction

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

2.6.5 Overall Time Complexity

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

$$T_{\text{overall}} = T_{\text{comp}} + T_{\text{comm}} = O\left(\frac{z \times k}{p}\right) + p \times \alpha + \beta \times m \times \frac{k}{p} + O(m \times k) \quad (2.12)$$

2.6.6 Performance Analysis

2.6.6.1 Performance Dependence

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

2.6.6.2 Expected Performance

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

2.7 Performance Metrics

The evaluation of algorithm performance within the main program (detailed in appendix ??) involves a systematic approach to assess efficiency across various implementations. In addition, the PETSc library (1) is used to compare the performance of the parallel algorithms with a highly optimised parallel sparse matrix-vector multiplication implementation. The program's workflow is structured as follows:

1. **Initialisation:** Initial setup of MPI and PETSc environments to enable distributed computation and matrix operations.
2. **Matrix and Vector Preparation:**
 - *Matrix Reading:* Sparse matrices are sourced from files utilising the `readMatrixMarketFile` method (refer to appendix ??), leveraging the Matrix Market I/O library (2) for efficient data handling.
 - *Fat Vector Generation:* Creation of fat vectors, with dimensions tailored to the corresponding matrices and predetermined column counts, ensuring compatibility for multiplication.
 - *Distribution:* Uniform distribution of matrices and vectors across processes for parallel computation.
3. **Serial Multiplication:** Execution and timing of matrix-vector multiplication in a serial context to establish a performance baseline.
4. **Parallel Multiplication Variants:** Application of distinct parallel multiplication strategies—line-based, column-based, and non-zero element—to measure execution times and identify efficiency variances.
5. **PETSc Implementation:**
 - *Conversion:* Adaptation of matrices and vectors to PETSc formats to utilise its optimized operations.
 - *Execution:* Conducting matrix-vector multiplication within the PETSc framework.
 - *Result Conversion:* Reformatting PETSc output back to Fat Vector for uniform result comparison.
6. **Result Comparison:** Validation of output correctness across serial, parallel, and PETSc implementations to ensure computational integrity.
7. **Finalisation:** Termination of MPI and PETSc environments and release of resources.

Comprehensive testing, facilitated by the `batch_test.sh` script (see appendix ??), was conducted to evaluate algorithmic performance under varying conditions—spanning different sparse matrix characteristics, fat vector column numbers, and process counts. These tests were categorised into three batches to pinpoint performance influencers:

- **Matrix Impact:** Examining how variations in sparse matrix properties affect algorithm efficiency.
- **Fat Vector Influence:** Assessing the performance implications of altering fat vector column quantities.
- **Execution Time Measurement:** Repeating the second batch tests without tracking average communication and computation times to capture precise execution durations.

This structured approach enables a thorough understanding of each algorithm's behavior under diverse computational scenarios and establishes a foundation for optimising sparse matrix-vector multiplication operations.

Chapter 3

Results and Discussion

3.1 Results

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. The number of columns in the fat vector was fixed at 6. The performance of all algorithms was evaluated using the following five sparse matrix (2, 3):

Matrix Name	Dimensions	Non-Zero Elements	Symmetric	Type
Cage4	9×9	49	No	Real
FEM_3D_thermal1	$17,880 \times 17,880$	430,740	No	Real
DC1	$116,835 \times 116,835$	766,396	No	Real
Cop20k_A	$121,192 \times 121,192$	2,624,331	Yes	Real
Amazon0302	$262,111 \times 262,111$	1,234,877	No	Binary

Table 3.1: Sparse matrix specifications

3.1.1.1 Execution Time Evolution

The following figures illustrate the evolution of the execution time for each of the five test matrices as the number of processes increases.

3.1.1.2 Average Communication Time Evolution

The following figures illustrate the evolution of the average communication time per process for each of the five test matrices as the number of processes increases.

3.1.1.3 Average Computation Time Evolution

The following figures illustrate the evolution of the average computation time per process for each of the five test matrices as the number of processes increases.

3.1.1.4 Performance Evolution

The following figures illustrate the evolution of the performance for each of the five test matrices as the number of processes increases.

3.1.1.5 Performance Summary

Here is a summary of the performance of the algorithms:

1. **Serial Algorithm:** The execution time increases with the size and non-zero elements number of the matrix due to the linear increase in computation. As expected, it remains constant with the number of processes. As shown in the figures ??,?? and ??, the serial algorithm outperforms the parallel algorithms for small matrices or with a lot of non-zero elements non-uniformly distributed like the Cage4 and DC1 matrices. This is due to the overhead associated with coordinating parallel tasks and the communication between processes.
2. **Line-Based Algorithm:** This algorithm performs better with matrices having a balanced distribution of non-zero elements across rows. Matrices with irregular distributions can lead to load imbalance among processes, affecting performance negatively.

For matrices like Amazon0302 or Cop20k_A with a high number of rows, this algorithm perform well due to its ability to parallelize over rows, but struggle with load balancing for matrices like Cage4 that are smaller or less uniformly distributed. This is evident in the figures ??, 3.18 and ??. As shown in the figures 3.8 and 3.13, the communication and computation time per process decreases with the number of processes. However, after about 60 processes, the total execution time starts to increase due to the increased communication overhead.
3. **Column-Based Algorithm:** Similar to the row-wise algorithm, but the performance is more sensitive to the matrix's column distribution. Sparsity patterns that lead to dense columns can result in significant overhead due to increased communication between processes.

4. **Non-Zero Element Algorithm:** The performance heavily depends on the distribution of non-zero elements. For matrices with a large number of non-zero elements scattered across, this algorithm can optimise the use of computational resources by focusing work where it's needed. As shown in the figures 3.18 and ??, it is efficient for matrices with a high density of non-zero elements like Cop20k_A, since it focuses computation only where necessary.
5. **PETSc Algorithm:** Based on the execution time, the PETSc algorithm outperforms all the other algorithms (see ??). However, the total performance remain constant with the number of processes and very low compared to the other algorithms (see ??) as this library is not designed to reconstruct the final result matrix.

3.1.2 Fat Vector Impact

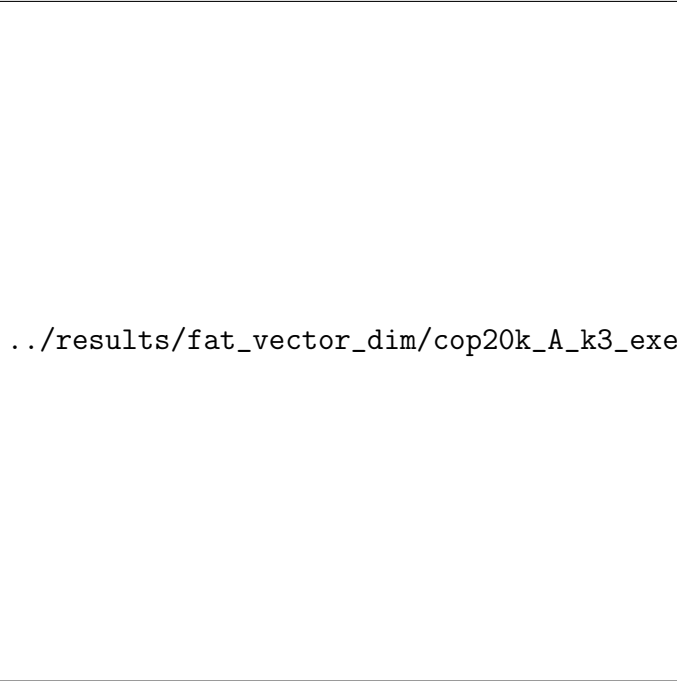
The second and third sets of experiments focused on the impact of the dimensions of the fat vector on the performance of the algorithms. The sparse matrix used was Cop20k_A. The performance of all algorithms was evaluated using the following five fat vector column counts: 1, 3, 6, 9, and 12.

3.1.2.1 Execution Time Evolution

The following figures illustrate the evolution of the execution time for each of the five test fat vector column counts as the number of processes increases.




Figure 3.1: Execution Time Evolution (k=1)



`../results/fat_vector_dim/cop20k_A_k3_execution_time.png`

Figure 3.2: Execution Time Evolution ($k=3$)



`../results/fat_vector_dim/cop20k_A_k6_execution_time.png`

Figure 3.3: Execution Time Evolution ($k=6$)



Figure 3.4: Execution Time Evolution (k=9)

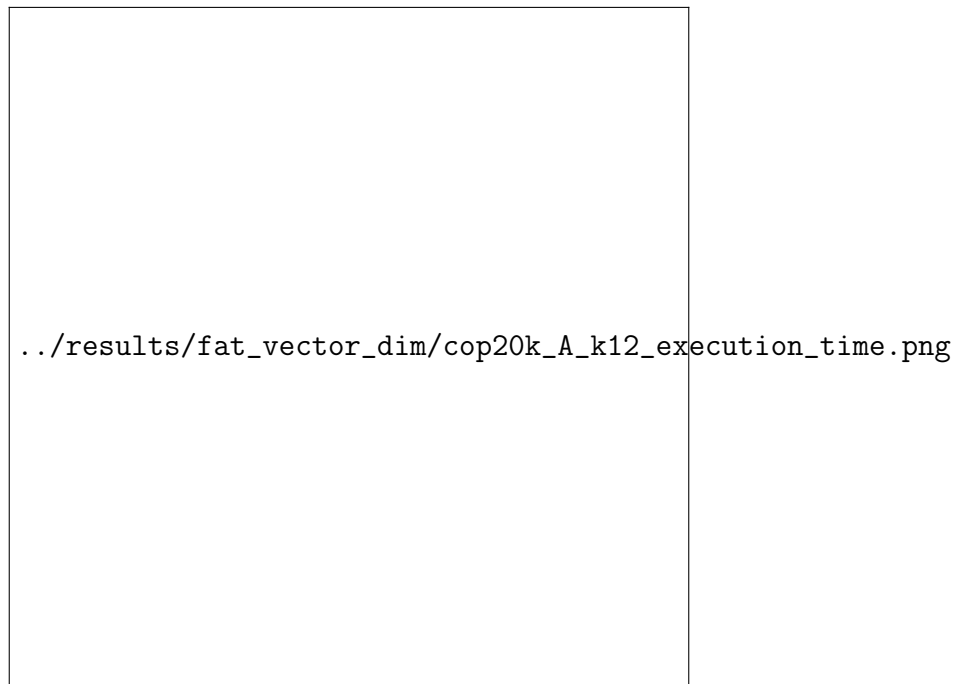


Figure 3.5: Execution Time Evolution (k=12)

3.1.2.2 Average Communication Time Evolution

The following figures illustrate the evolution of the average communication time per process for each of the five test fat vector column counts as the number of processes increases.

Figure 3.6: Communication Time Evolution ($k=1$)Figure 3.7: Communication Time Evolution ($k=3$)

Figure 3.8: Communication Time Evolution ($k=6$)Figure 3.9: Communication Time Evolution ($k=9$)



Figure 3.10: Communication Time Evolution (k=12)

3.1.2.3 Average Computation Time Evolution

The following figures illustrate the evolution of the average computation time per process for each of the five test fat vector column counts as the number of processes increases.

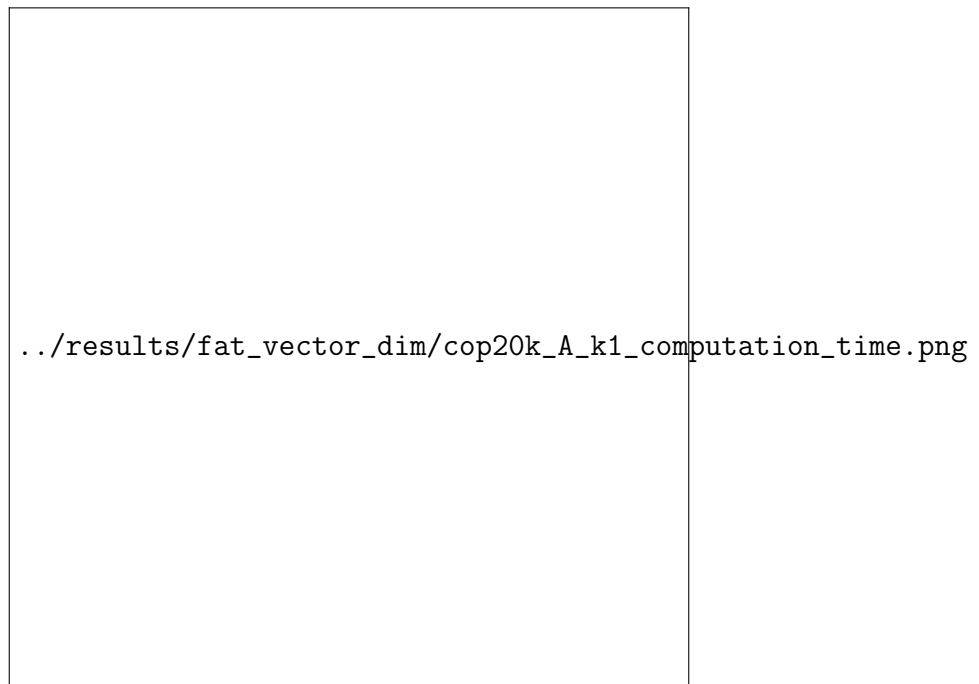


Figure 3.11: Computation Time Evolution (k=1)



Figure 3.12: Computation Time Evolution (k=3)

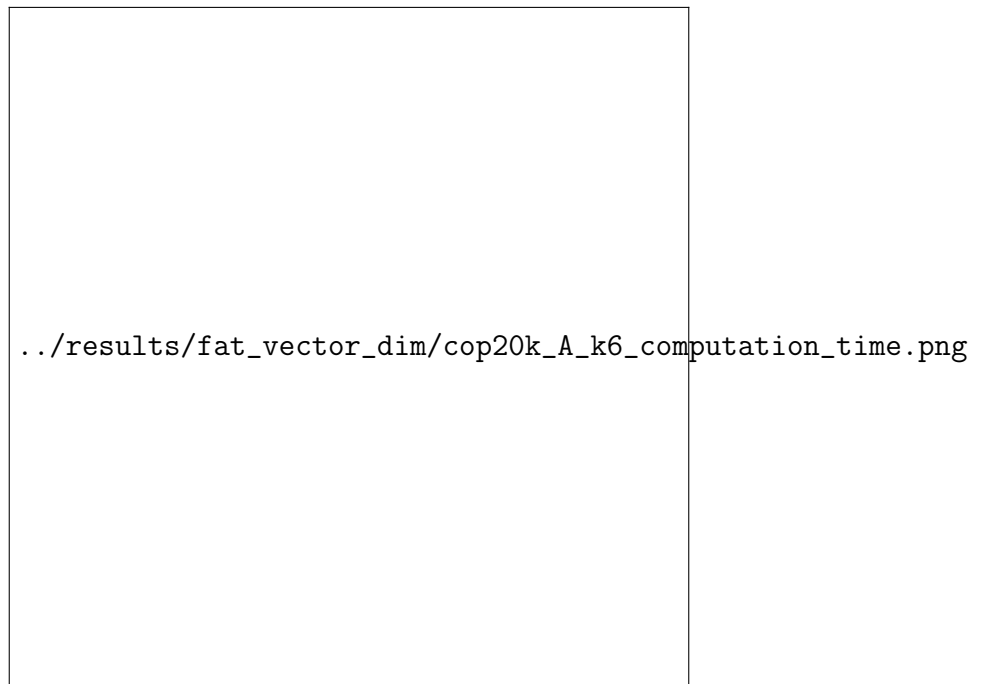


Figure 3.13: Computation Time Evolution (k=6)



Figure 3.14: Computation Time Evolution (k=9)



Figure 3.15: Computation Time Evolution (k=12)

3.1.2.4 Performance Evolution

The following figures illustrate the evolution of the performance for each of the five test fat vector column counts as the number of processes increases.

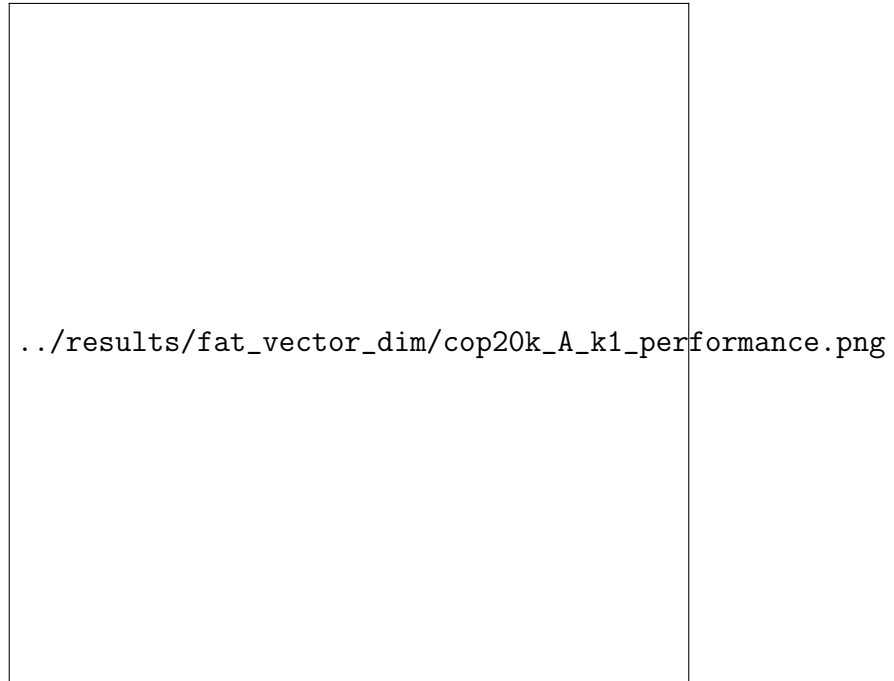


Figure 3.16: Performance Evolution (k=1)

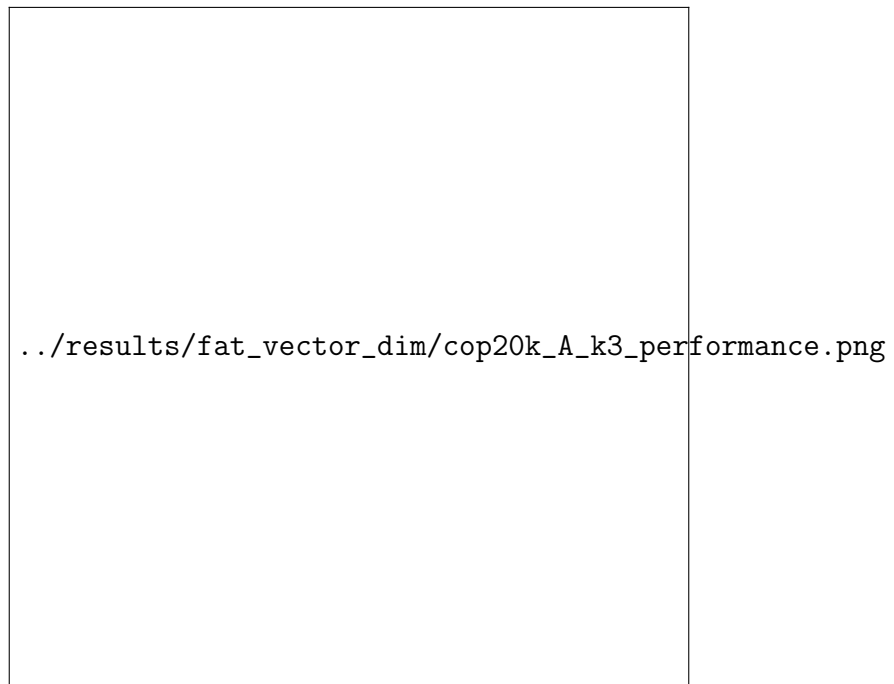


Figure 3.17: Performance Evolution (k=3)

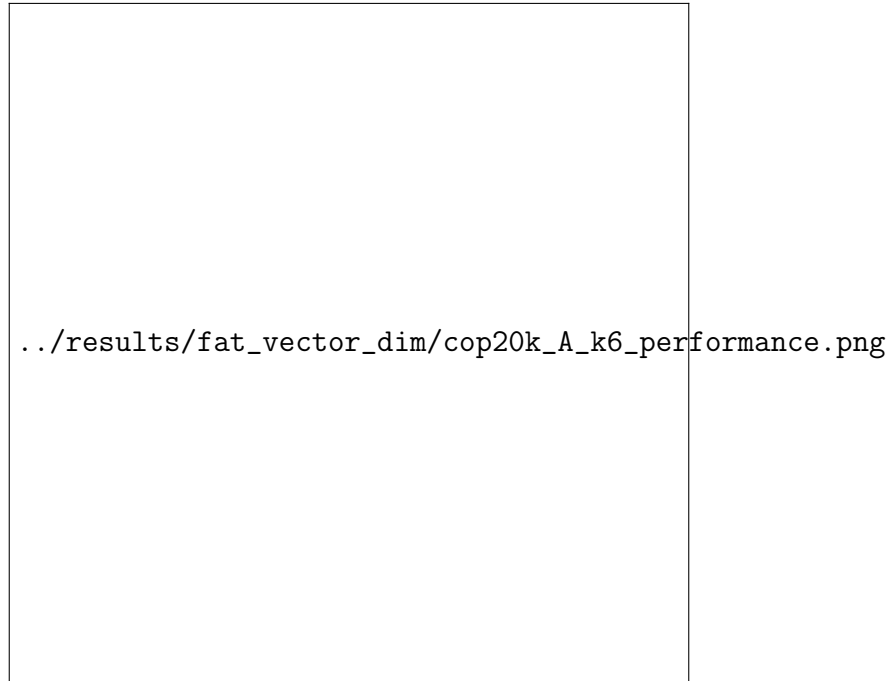


Figure 3.18: Performance Evolution (k=6)

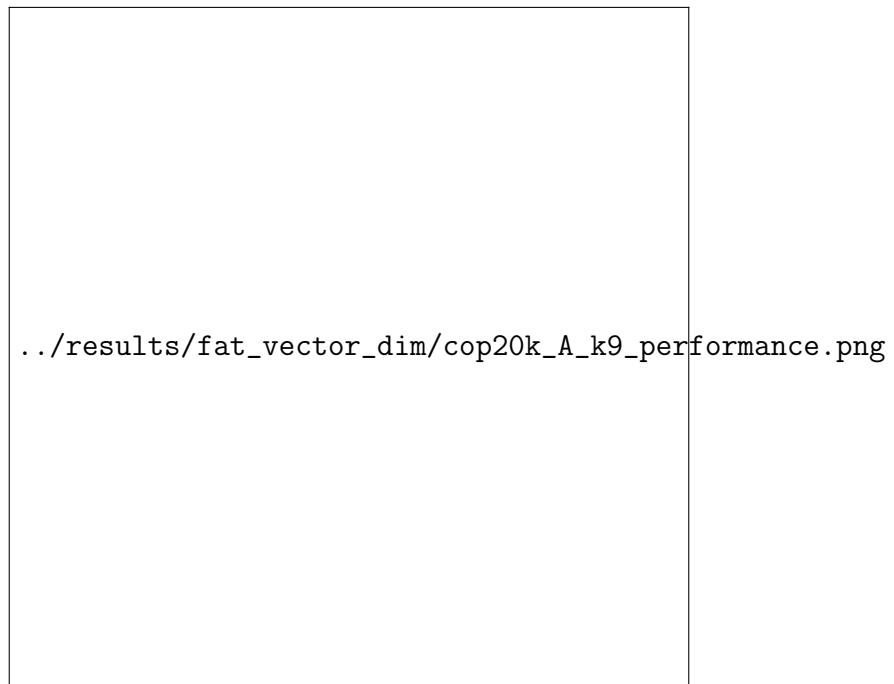


Figure 3.19: Performance Evolution (k=9)



Figure 3.20: Performance Evolution (k=12)

3.1.2.5 Performance Summary

Here is a summary of the performance of the algorithms:

1. **Serial Algorithm:** As shown in figures 3.16 and 3.20, the serial algorithm outperforms the parallel algorithms for small dimensions but is outperformed by the parallel algorithms for larger dimensions.
2. **Line-Based Algorithm:** As shown in the figures 3.16 and 3.20, the performance of the line-based algorithm increases with the number of processes and the number of columns in the fat vector. However, there is a drop in performance when the number of processes is too large. This is due to the increased communication overhead as shown in the figures 3.6 and 3.10.
3. **Column-Based Algorithm:** As shown in the figures 3.16 and 3.20, the performance of the column-based algorithm increases with the number of processes and the number of columns in the fat vector. However, there is a large drop in performance when the number of processes is greater than the number of columns in the fat vector. This is due to the increased communication overhead as shown in the figures 3.1 and 3.5.
4. **Non-Zero Element Algorithm:** As shown in the figures 3.7 and 3.10, the communication time per process increase proportionally with the number of processes and the number of columns in the fat vector. As the complexity of the computation increases with the number of columns in the fat vector, the performance becomes the same as the row-wise algorithm. This is evident in the figures 3.17 and 3.20.

5. **PETSc Algorithm:** As expected, the execution time decreases with the number of processes but stay constant with the number of columns in the fat vector as shown in the figures 3.1 and 3.5. This is due to the highly optimized parallel sparse matrix-vector multiplication implementation in the PETSc library. However, the performance is not good compared to the custom implementations as it not optimised to reconstruct the final result from the gathered vector.

3.2 HPC's Environmental Impact

Although HPC systems are essential for complex calculations and simulations in a variety of fields, their impact on the environment is not negligible. Energy consumption and heat generation require the use of large cooling systems, which further increases their ecological footprint. However, advances in energy-efficient cooling technologies and methods are paving the way for more sustainable HPC operations.

Irina Kupiainen's article highlights the environmental considerations involved in HPC operations, highlighting the exemplary case of LUMI in integrating sustainability. LUMI is one of the most powerful supercomputers in the world, while having a neutral, or even positive, impact on the environment. Located in Finland, it benefits from the country's cool climate and access to renewable energy sources, making it possible to reconcile high performance with environmental sustainability (4).

New innovations are emerging in the field of green HPC. For example, it is possible to use new cooling systems that significantly lower junction temperatures and require less pumping power (5). In addition, Green Cloud Computing initiatives aim to optimise the use of resources, by exploiting renewable energy sources and adopting energy-saving technologies such as virtualisation (6, 7). These advances demonstrate a commitment to sustainability, underlining the move towards reducing energy consumption and emissions in high-performance computing operations, while maintaining energy efficiency.

Finally, the adoption of metrics and standards plays a pivotal role in guiding HPC towards sustainability. The Green Index (TGI) exemplifies this by providing a comprehensive metric for evaluating the energy efficiency of HPC systems (8). It aggregates various benchmarks into a singular measure, facilitating comparisons of system-wide efficiency. Additionally, the Green500 list ranks supercomputers based on their energy efficiency, promoting an industry standard that encourages the development of environmentally friendly HPC technologies. These metrics and rankings not only spotlight the most energy-efficient systems but also drive competition and innovation towards greener HPC solutions.

Chapter 4

Conclusion

In conclusion, this report has meticulously analysed various parallelization strategies for fat matrix vector multiplication, based on the MPI and PETSc frameworks, and through extensive experimentation and comparison, highlights the importance of choosing an appropriate parallelization approach based on matrix characteristics and computational resources. The results highlight the trade-offs between computational and communication overheads, emphasising the need for a balanced distribution of workloads to maximise efficiency. As expected the PETSc library outperforms the custom implementations in terms of execution time but not in terms of performance, as the matrices stay distributed even after the multiplication. The custom implementations are more efficient to reconstruct the final result from the gathered vector. In addition, the discussion of the environmental impact of HPC practices, with a nod to sustainable computing, reflects a broader perspective on the implications of computational research. Future directions could explore adaptive parallelization techniques, further optimizing specific models and matrix sizes, possibly integrating AI for dynamic algorithm selection based on real-time performance metrics. This study not only contributes to the field of HPC by providing insight into efficient algorithmic implementations, but also paves the way for more energy-efficient HPC systems, in line with global sustainable development goals.

References

1. Balay S, Abhyankar S, Adams MF, Benson S, Brown J, Brune P, et al.. PETSc Web page; 2023. <https://petsc.org/>. Available at: <https://petsc.org/>. (Accessed: December 28, 2023).
2. Lugowski A. fast matrix market: Fast and Full-Featured Matrix Market I/O Library; 2023. Available at: https://github.com/alugowski/fast_matrix_market. (Accessed: December 28, 2023).
3. Kolodziej SP, Aznaveh M, Bullock M, David J, Davis TA, Henderson M, et al. The SuiteSparse Matrix Collection Website Interface. *Journal of Open Source Software*. 2019;4(35):1244-8. Available at: <https://sparse.tamu.edu/>. (Accessed: December 28, 2023).
4. Kupiainen I. HPC at the core of green and digital transition; 2021. Available at: <https://www.scientific-computing.com/viewpoint/hpc-core-green-and-digital-transition>. (Accessed: January 02, 2024).
5. Karwa N. Ultra-Low Global Warming Potential Heat Transfer Fluids for Pumped Two-Phase Cooling in HPC Data Centers; 2020. Available at: <https://ieeexplore.ieee.org/document/9190269>. (Accessed: December 28, 2023).
6. Geetanjali, Quraishi SJ. Energy Savings using Green Cloud Computing; 2022. Available at: <https://ieeexplore.ieee.org/document/9917654>. (Accessed: December 28, 2023).
7. Guyon D, Orgerie AC, Morin C, Agarwal D. How Much Energy Can Green HPC Cloud Users Save?; 2017. Available at: <https://ieeexplore.ieee.org/document/7912681>. (Accessed: December 28, 2023).
8. Subramaniam B, Feng Wc. The Green Index: A Metric for Evaluating System-Wide Energy Efficiency in HPC Systems; 2012. Available at: <https://ieeexplore.ieee.org/document/6270748>. (Accessed: December 28, 2023).

Appendix A

Documentation

Appendix A.A Project tree

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

Appendix A.B Getting Started

To run the program, follow these steps:

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

```

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

```

3. Run the program using the following command:

```

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

```

Appendix A.C Methods Overview

A.C.1 Utils.h

A.C.1.1 ConvertPETScMatToFatVector

Description: Converts a PETSc matrix to a FatVector stucture.

Parameters:

- Mat C - PETSc matrix to be converted.

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

A.C.1.2 areMatricesEqual

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

Parameters:

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

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

A.C.1.3 readMatrixMarketFile

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

Parameters:

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

Returns: SparseMatrix - Sparse matrix read from the file.

A.C.1.4 generateLargeFatVector

Description: Generates a random Fat Vector with specified dimensions.

Parameters:

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

Returns: `FatVector` - Generated fat vector.

A.C.1.5 serialize and deserialize

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

Parameters for serialize:

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

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

Parameters for deserialize:

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

Returns: `FatVector` - Deserialized fat vector.

A.C.2 SparseMatrixFatVectorMultiply.h

A.C.2.1 sparseMatrixFatVectorMultiply

Description: Executes the multiplication using a sequential algorithm.

Parameters:

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

Returns: `FatVector` - Result of the multiplication.

A.C.3 SparseMatrixFatVectorMultiplyRowWise.h

A.C.3.1 sparseMatrixFatVectorMultiplyRowWise

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

Parameters:

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

Returns: FatVector - Result of the multiplication.

A.C.4 SparseMatrixFatVectorMultiplyColumnWise.h

A.C.4.1 sparseMatrixFatVectorMultiplyColumnWise

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

Parameters:

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

Returns: FatVector - Result of the multiplication.

A.C.5 SparseMatrixFatVectorMultiplyNonZeroElement.h

A.C.5.1 sparseMatrixFatVectorMultiplyNonZeroElement

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

Parameters:

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

Returns: FatVector - Result of the multiplication.

Appendix B

Source Codes

Appendix B.A Data Structures

Data stuctures of the sparse matrix and fat vector.