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
26th 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:

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

where most elements of M are zeros.

The vector v is defined as:

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

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

Matrix Name	M	NZ	Avg_NZ	Max_NZ	Symmetric
cavity10	2597	76367	29.4	62	False
PR02R	161070	8185136	50.8	92	False
nlpkkt80	1062400	28704672	27.0	28	True
Cube_Coup_dt0	2164760	127206144	58.8	68	True
roadNet-PA	1090920	3083796	2.8	9	True
ML_Laplace	377002	27689972	73.4	74	False
bcsstk17	10974	428650	39.1	150	True
mhda416	416	8562	20.6	33	False
af_1_k101	503625	17550675	34.8	35	True
thermal1	82654	574458	7.0	11	True
thermomech_TK	102158	711558	7.0	10	True
cage4	9	49	5.4	6	False
cant	62451	4007383	64.2	78	True
dc1	116835	766396	6.6	114190	False
raefsky2	3242	294276	90.8	108	False
rdist2	3198	56934	17.8	61	False
mcfe	765	24382	31.9	81	False
olm1000	1000	3996	4.0	6	False
lung2	109460	492564	4.5	8	False
webbase-1M	1000005	3105536	3.1	4700	False
mhd4800a	4800	102252	21.3	33	False
west2021	2021	7353	3.6	12	False
thermal2	1228045	8580313	7.0	11	True
adder_dcop_32	1813	11246	6.2	100	False
mac_econ_fwd500	206500	1273389	6.2	44	False
FEM_3D_thermal1	17880	430740	24.1	27	False
amazon0302	262111	1234877	4.7	5	False
cop20k_A	121192	2624331	21.7	81	True
olafu	16146	1015156	62.9	89	True
af23560	23560	484256	20.6	21	False

Table 2.1: Summary of sparse matrices

2.2.1.1 Compressed Sparse Row (CSR) Format

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

2.2.1.2 ELLPACK

The ELLPACK sparse matrix representation has gained wide usage thanks to its storage and computational efficiency. Within the ELLPACK format, the matrix is stored in arrays that possess two dimensions. Each row of the array contains a set number of non-zero elements, with the non-zero values being stored in the values array, and the corresponding column indices being stored in the column index array as represented in Figure 4. The void entries in the matrix are filled with zeroes in AS and -1 in JA. This format is optimized for parallel computation, particularly for matrices with uniform sparsity patterns, and is well-suited for use with CUDA frameworks.

Compared to the CRS format, the ELLPACK format provides a fixed-length representation for each row, which can significantly improve computational efficiency by using caches and registers. Moreover, the ELLPACK format is more memory-efficient compared to the CSR format, particularly for matrices with uniform sparsity patterns.

However, it's important to note that the ELLPACK format may not be the most efficient representation for matrices with irregular sparsity patterns. This may result in significant storage overhead and inefficient memory access. Additionally, the fixed-length representation for each row may not be optimal for matrices with variable sparsity patterns, where a lot of unnecessary data will be stored.

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 B.A.

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 Sparse Matrix-Dense Vector Multiplication (CRS)

Require: A is an $m \times n$ sparse matrix in CRS format
Require: X is an $n \times k$ fat vector
Ensure: Y is an $m \times k$ fat vector, result of $A \times X$
  for $i = 0$ to $A.numRows - 1$ **do**
    for $j = A.rowPtr[i]$ to $A.rowPtr[i+1] - 1$ **do**
      $colIndex \leftarrow A.colIndices[j]$
      $value \leftarrow A.values[j]$
      for $k = 0$ to $X.numCols - 1$ **do**
        $yIndex \leftarrow i \times X.numCols + k$
        $xIndex \leftarrow colIndex \times X.numCols + k$
        $Y.values[yIndex] \leftarrow Y.values[yIndex] + value \times X.values[xIndex]$
      end for
    end for
  end for

Algorithm 2 Sparse Matrix-Dense Vector Multiplication (ELLPACK)

Require: A is an $m \times n$ sparse matrix in ELLPACK format
Require: X is an $n \times k$ fat vector
Ensure: Y is an $m \times k$ fat vector, result of $A \times X$
  for $i = 0$ to $A.numRows - 1$ **do**
    for $j = 0$ to $A.maxNonZerosPerRow - 1$ **do**
      $colIndex \leftarrow A.colIndices[i \times A.maxNonZerosPerRow + j]$
      $value \leftarrow A.values[i \times A.maxNonZerosPerRow + j]$
      if $colIndex \neq -1$ **then**
        for $k = 0$ to $X.numCols - 1$ **do**
          $yIndex \leftarrow i \times X.numCols + k$
          $xIndex \leftarrow colIndex \times X.numCols + k$
          $Y.values[yIndex] \leftarrow Y.values[yIndex] + value \times X.values[xIndex]$
        end for
      end if
    end for
  end for

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 3 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 ← m/worldSize
extraRows ← m mod worldSize
startRow ← worldRank × rowsPerProcess + min(worldRank, extraRows)
endRow ← startRow + rowsPerProcess
if worldRank < extraRows then
    endRow ← endRow + 1
end if
Initialise localResult with zeros of size (endRow − startRow) × k
for i ← startRow to endRow − 1 do
    for each non-zero element (j, value) in row i of M do
        for k ← 0 to k − 1 do
            localIndex ← (i − startRow) × k + k
            localResult[localIndex] ← localResult[localIndex] + value × 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:

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

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:

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

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:

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

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 4 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 $M \times v$
  $colsPerProcess \leftarrow k/worldSize$
  $extraCols \leftarrow k \mod worldSize$
  $startCol \leftarrow worldRank \times colsPerProcess$
  $endCol \leftarrow startCol + colsPerProcess$
  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.

2.5.2 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:

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

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:

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

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:

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

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 5 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 $localResult$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.

2.6.2 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:

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

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:

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

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:

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

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

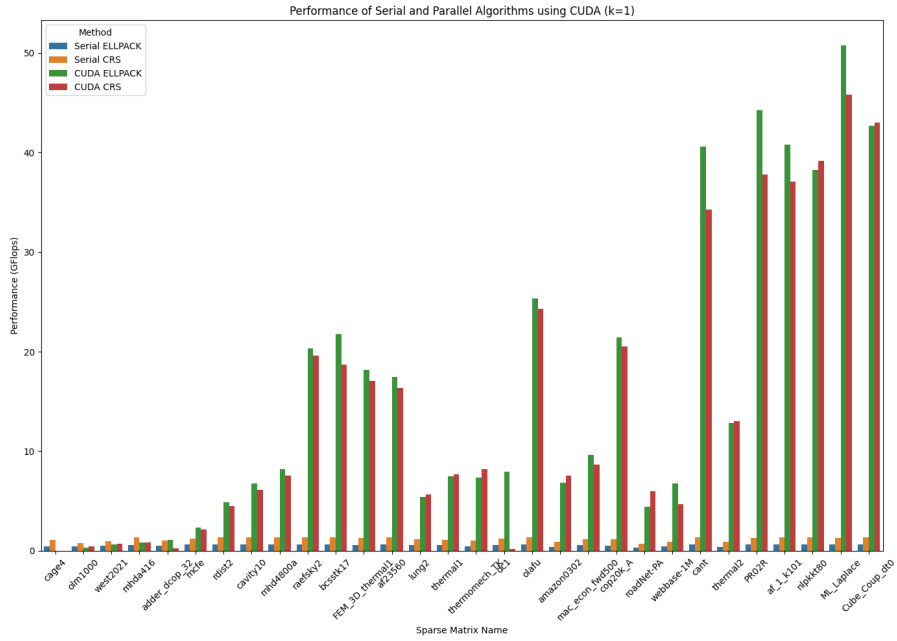

Figure 3.1: Performance of Serial and Parallel Algorithms using CUDA for $k = 1$

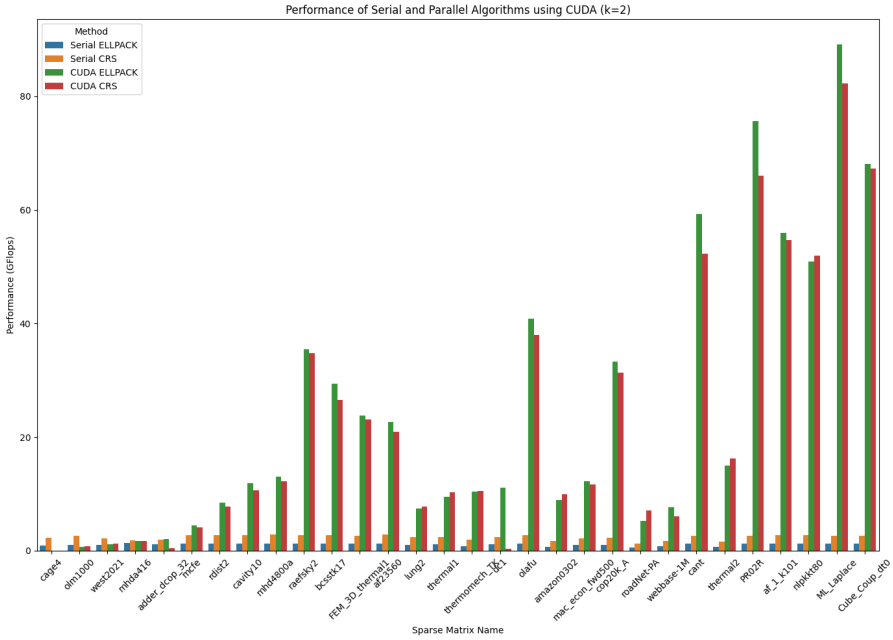

Figure 3.2: Performance of Serial and Parallel Algorithms using CUDA for $k = 2$

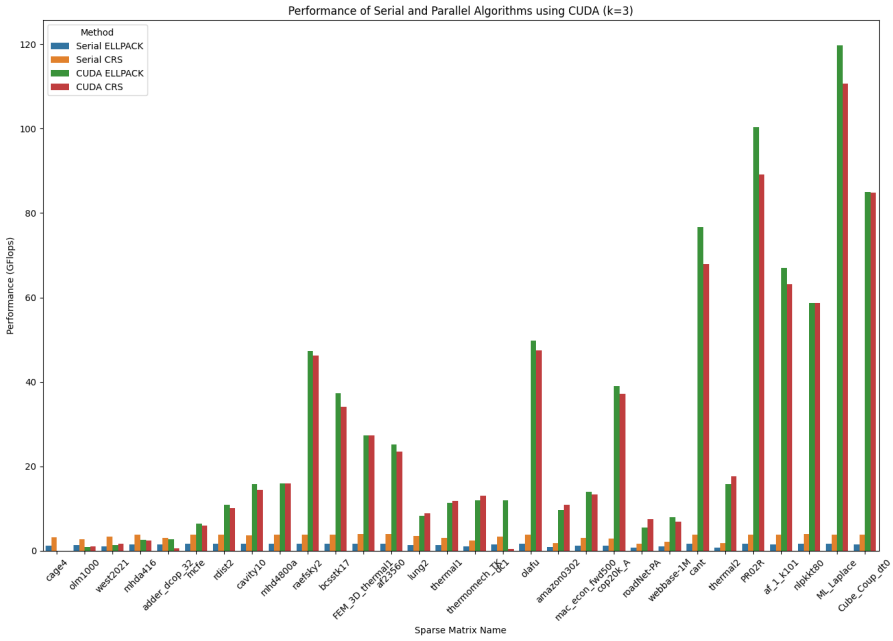

Figure 3.3: Performance of Serial and Parallel Algorithms using CUDA for $k = 3$

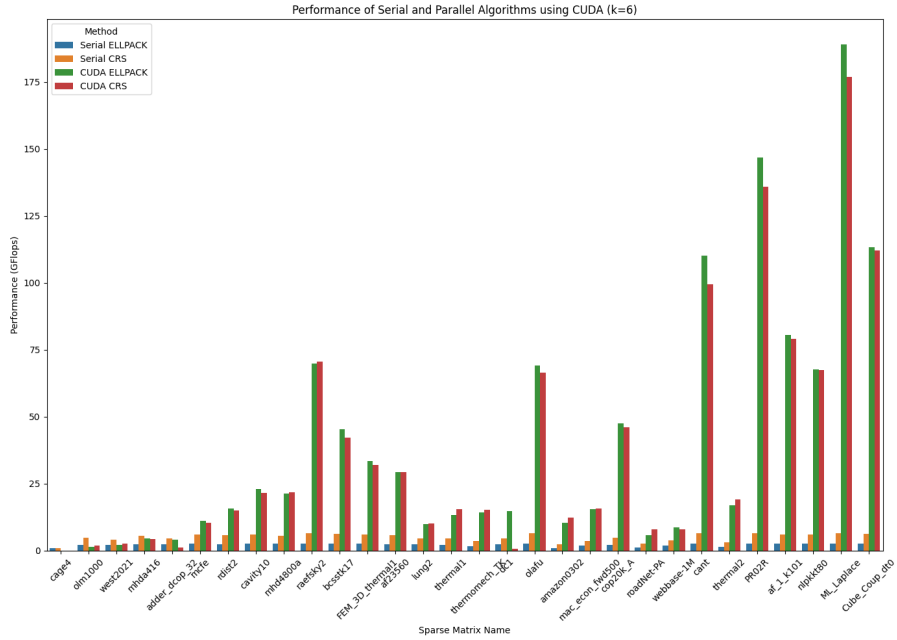

Figure 3.4: Performance of Serial and Parallel Algorithms using CUDA for $k = 6$

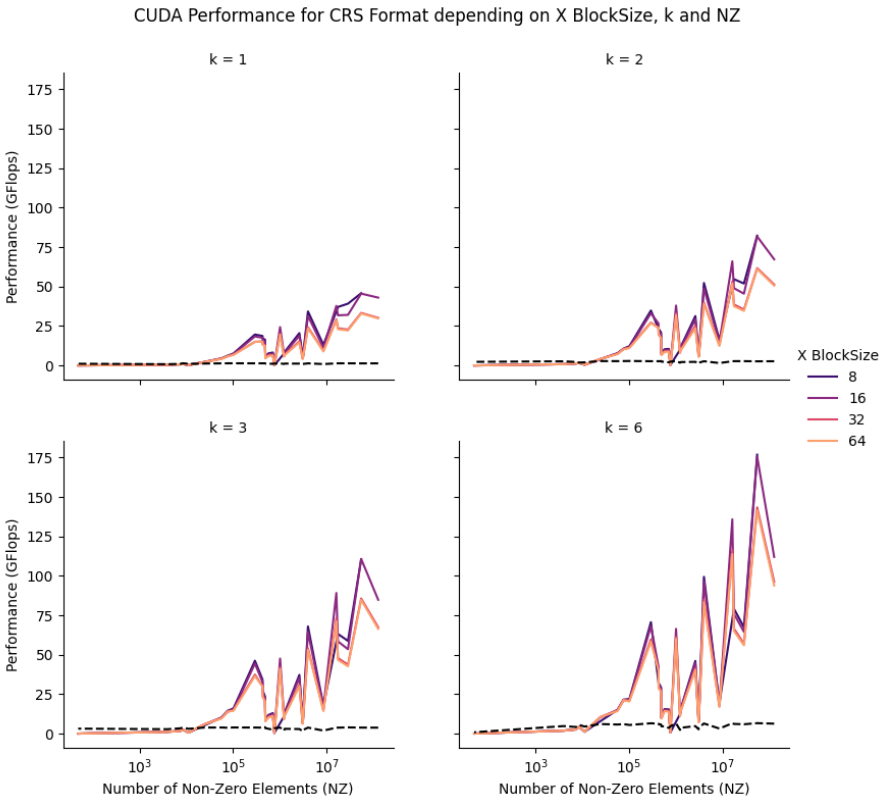

Figure 3.5: CUDA Performance for CRS Format depending on X BlockSize

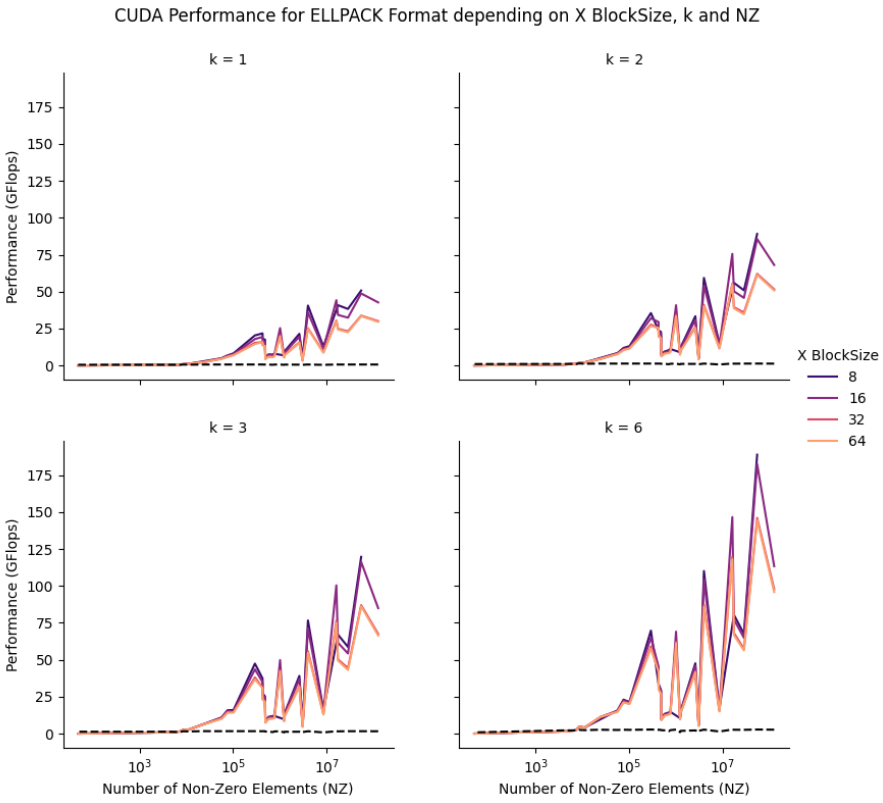

Figure 3.6: CUDA Performance for ELLPACK Format depending on X BlockSize

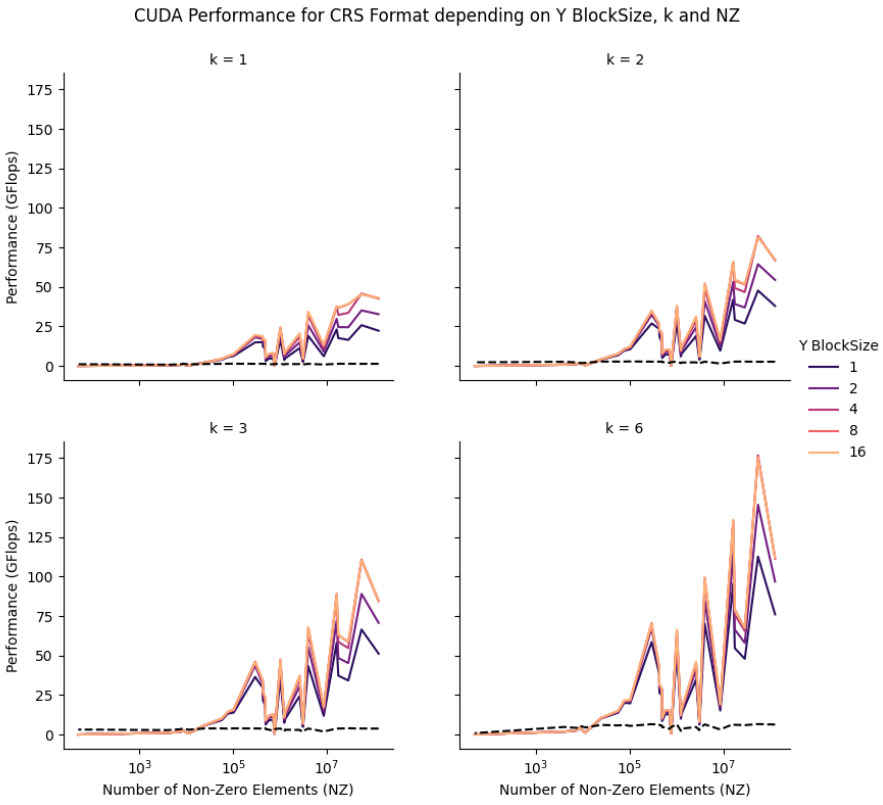

Figure 3.7: CUDA Performance for CRS Format depending on Y BlockSize

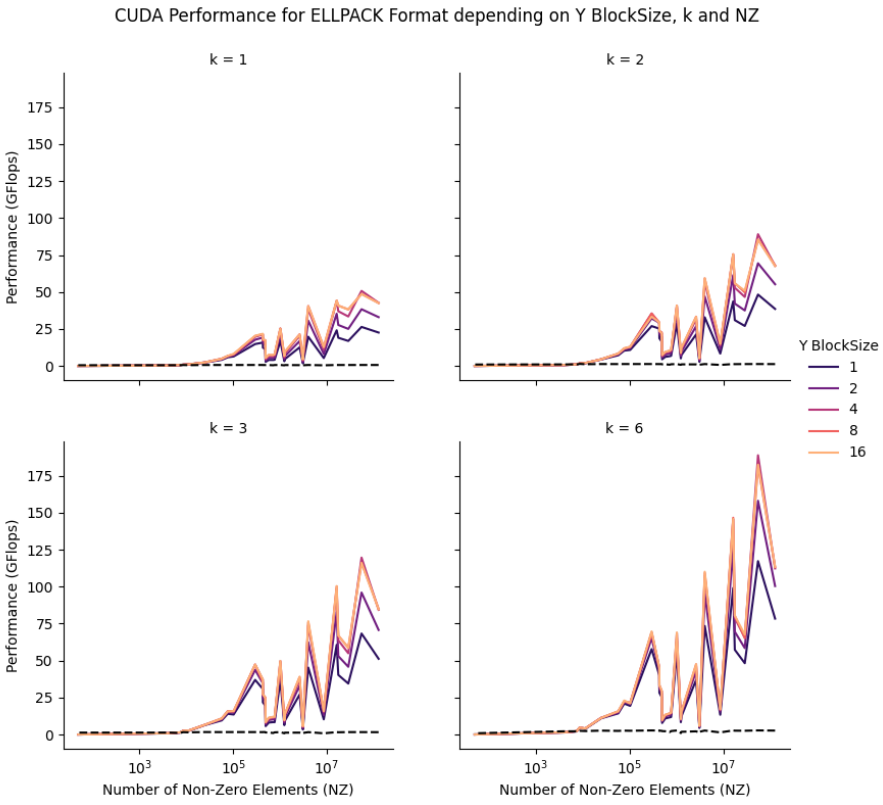

Figure 3.8: CUDA Performance for ELLPACK Format depending on Y BlockSize

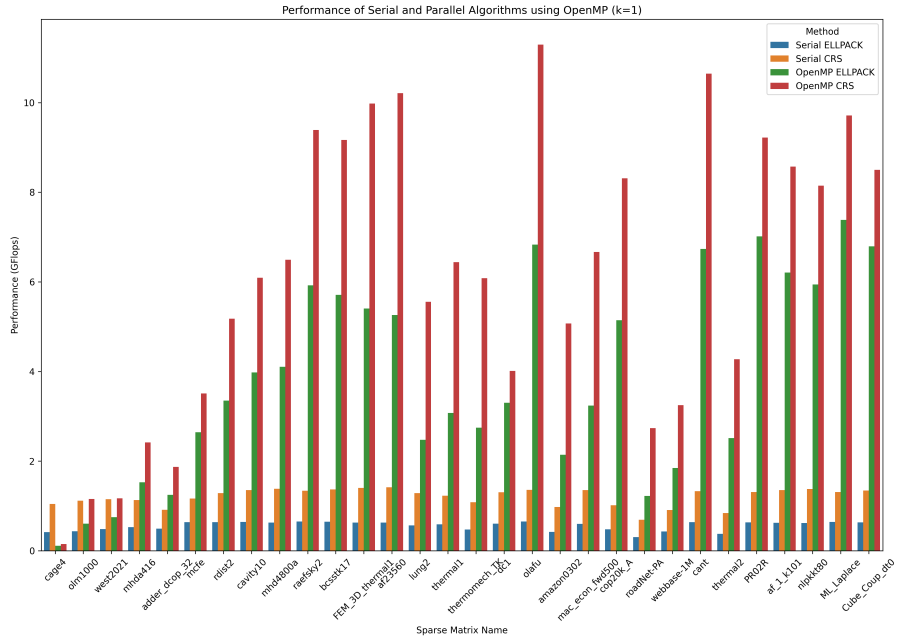

Figure 3.9: Performance of Serial and Parallel Algorithms using OpenMP for $k = 1$

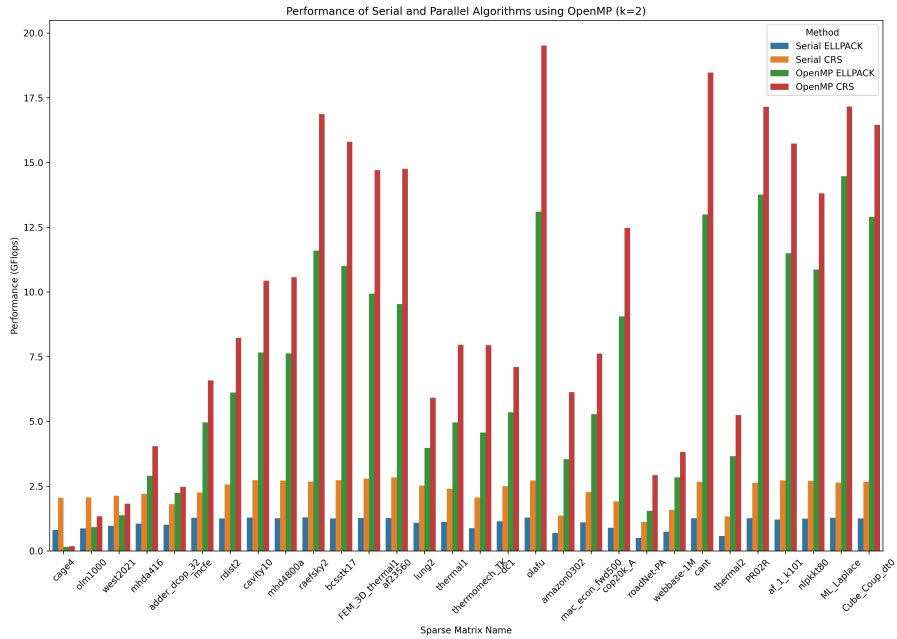

Figure 3.10: Performance of Serial and Parallel Algorithms using OpenMP for $k = 2$

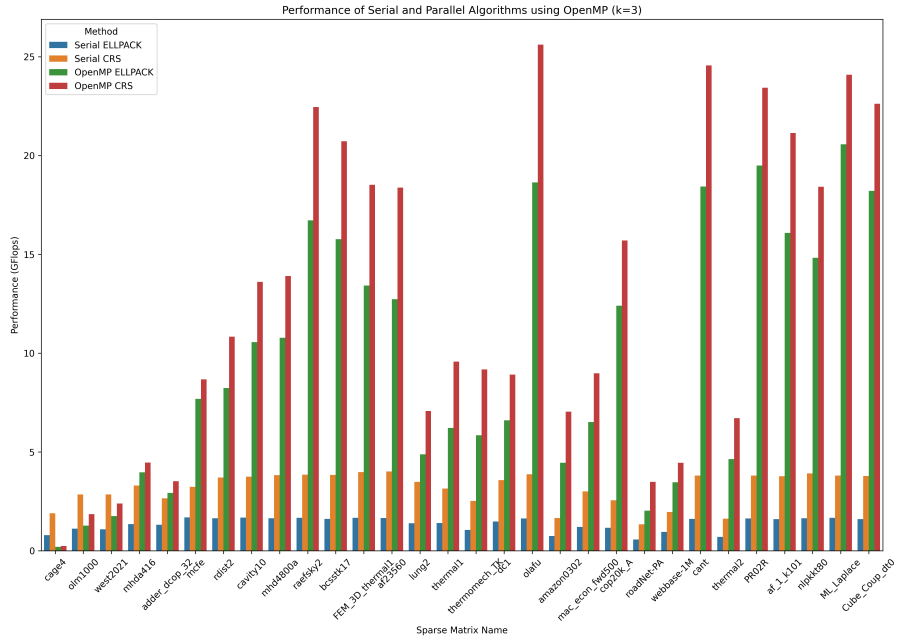

Figure 3.11: Performance of Serial and Parallel Algorithms using OpenMP for $k = 3$

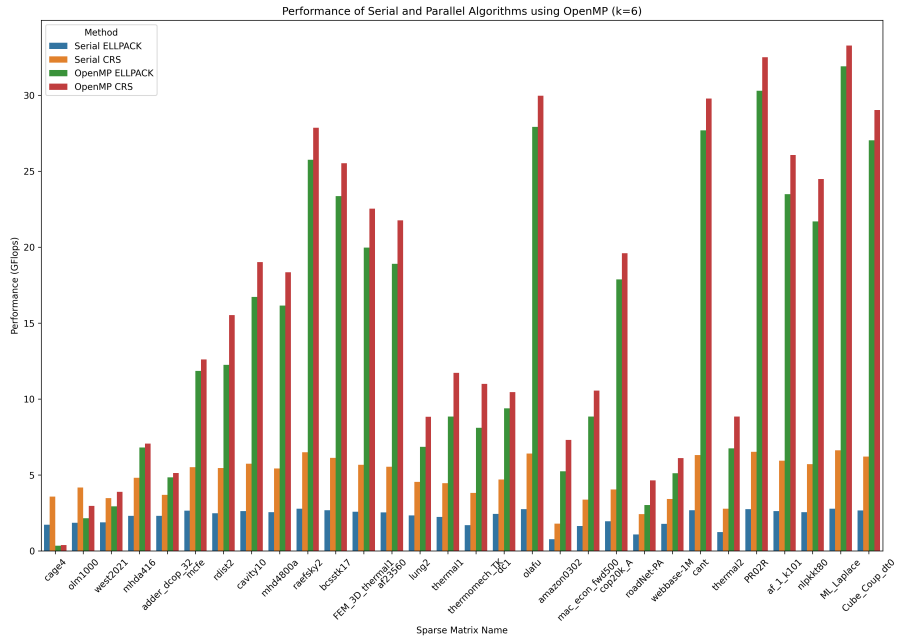

Figure 3.12: Performance of Serial and Parallel Algorithms using OpenMP for $k = 6$

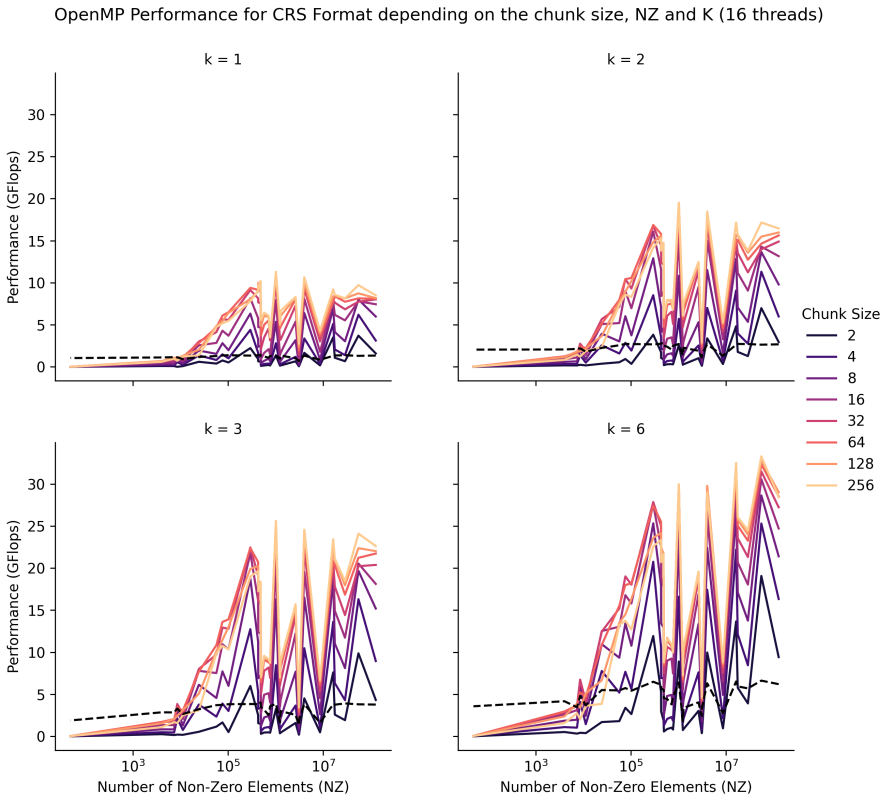

Figure 3.13: OpenMP Performance for CRS Format depending on Chunk Size

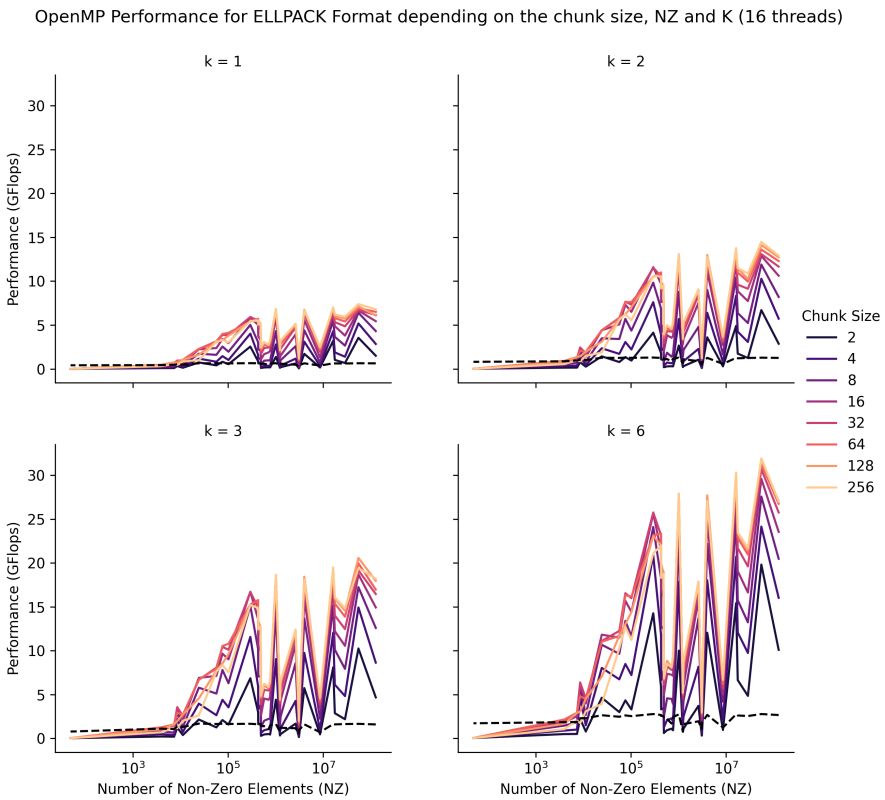

Figure 3.14: OpenMP Performance for ELLPACK Format depending on Chunk Size

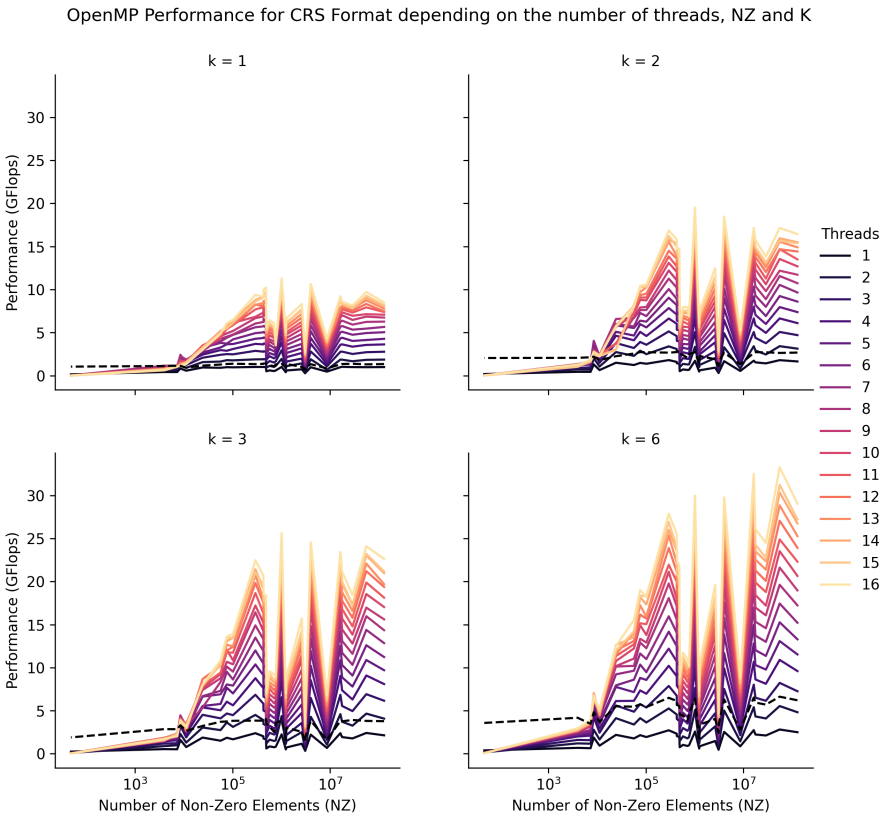

Figure 3.15: OpenMP Performance for CRS Format depending on Threads

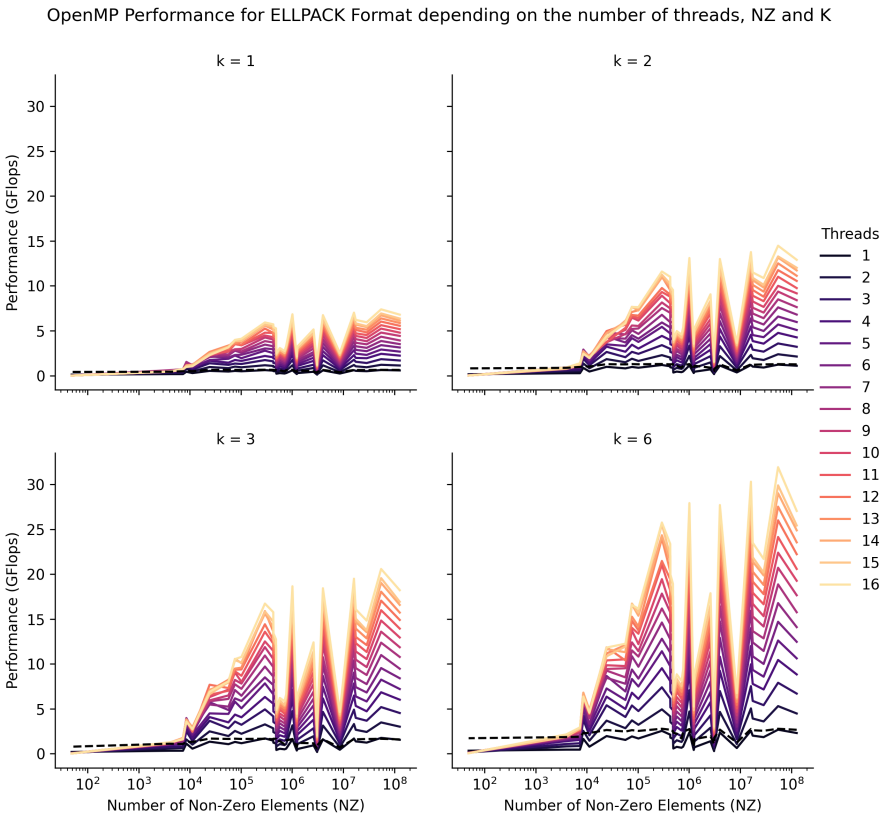

Figure 3.16: OpenMP Performance for ELLPACK Format depending on Threads

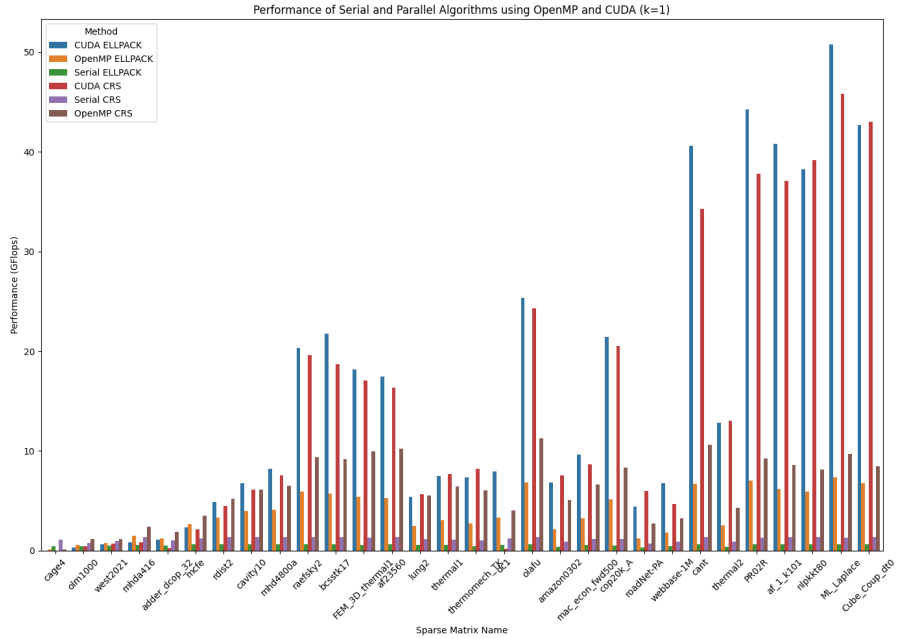

Figure 3.17: OpenMP vs CUDA Performance for $k = 1$

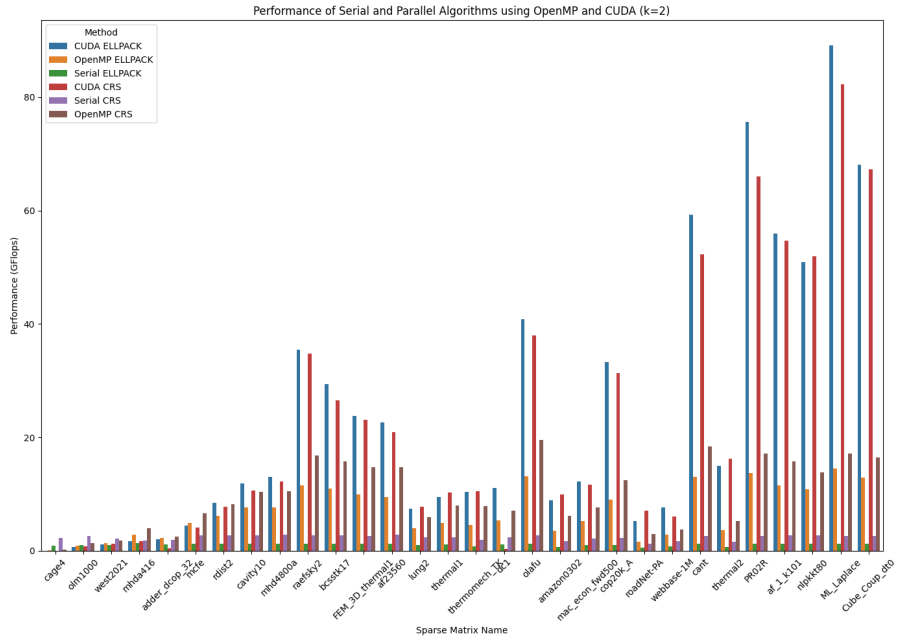

Figure 3.18: OpenMP vs CUDA Performance for $k = 2$

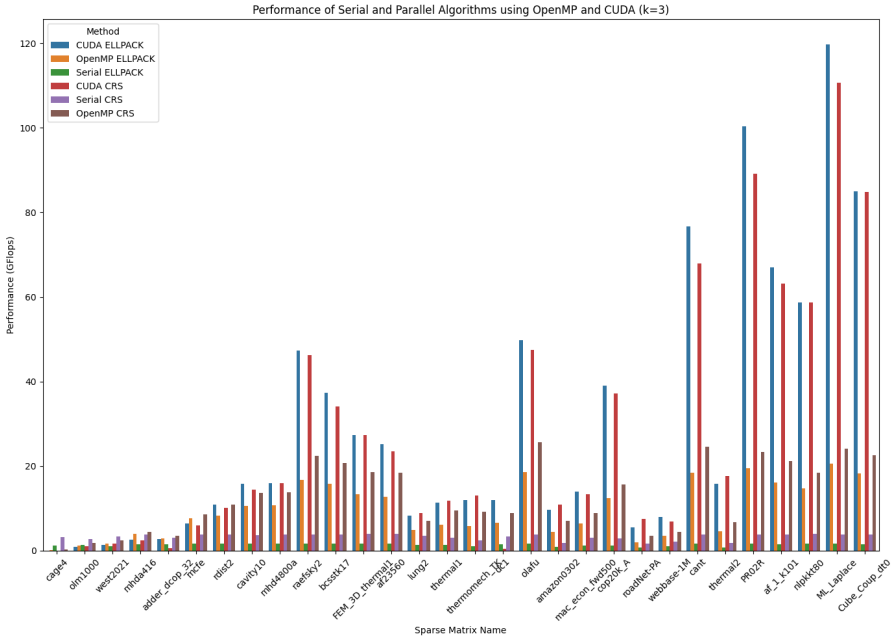

Figure 3.19: OpenMP vs CUDA Performance for $k = 3$

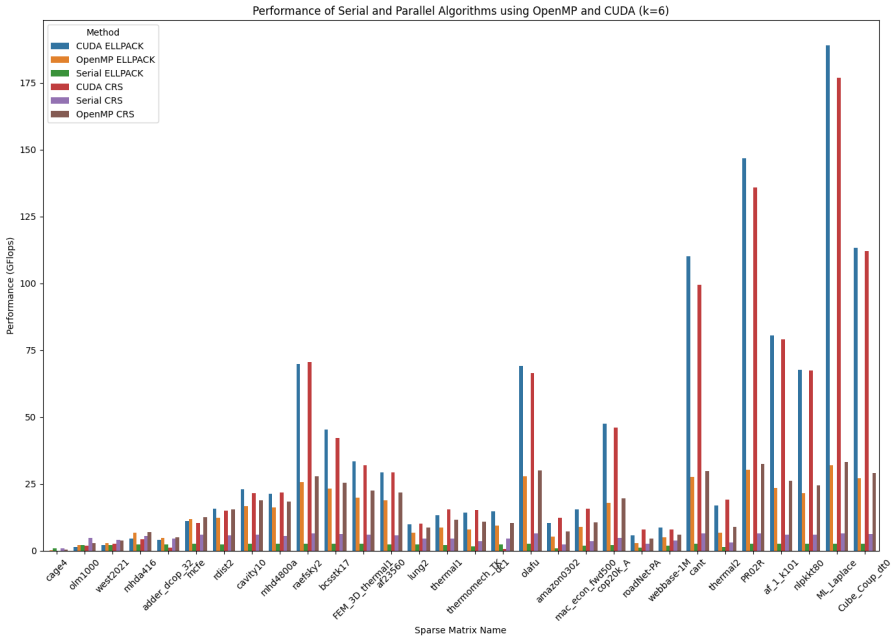

Figure 3.20: OpenMP vs CUDA Performance for $k = 6$

Table 3.1: Performance Comparison of CRS and ELLPACK Sequential Algorithms

Matrix	**CRS**	**ELLPACK**	**Best Structure**
Cube_Coup_dt0	6.313540	2.659870	CRS
FEM_3D_thermal1	5.939650	2.543850	CRS
ML_Laplace	6.629070	2.775640	CRS
PR02R	6.577900	2.755250	CRS
adder_dcop_32	4.481260	2.360730	CRS
af23560	5.804440	2.509250	CRS
af_1_k101	6.139610	2.608660	CRS
amazon0302	2.344020	0.968039	CRS
bcsstk17	6.256790	2.654560	CRS
cage4	0.871628	0.849791	CRS
cant	6.419710	2.687030	CRS
cavity10	6.032170	2.625250	CRS
cop20k_A	4.771770	2.155300	CRS
dc1	4.686860	2.349130	CRS
lung2	4.634780	2.319800	CRS
mac_econ_fwd500	3.684580	1.851230	CRS
mcfe	6.069120	2.634280	CRS
mhd4800a	5.474340	2.549000	CRS
mhda416	5.524470	2.516310	CRS
nlpkkt80	5.932140	2.567120	CRS
olafu	6.481010	2.724440	CRS
olm1000	4.775320	2.192310	CRS
raefsky2	6.556030	2.767780	CRS
rdist2	5.762990	2.465300	CRS
roadNet-PA	2.721200	1.240050	CRS
thermal1	4.599780	2.228430	CRS
thermal2	3.206760	1.437750	CRS
thermomech_TK	3.558870	1.592880	CRS
webbase-1M	3.750440	1.888800	CRS
west2021	4.184400	2.181630	CRS

Table 3.2: CRS vs ELLPACK avec OpenMP

Matrix	**CRS**	**ELLPACK**	**Best Structure**	**Speedup**
Cube_Coup_dt0	29.0243	27.0415	CRS	4.597152
FEM_3D_thermal1	22.5277	19.9711	CRS	3.792766
ML_Laplace	33.2766	31.9121	CRS	5.019799
PR02R	32.5086	30.2968	CRS	4.942094
adder_dcop_32	5.12204	4.83417	CRS	1.142991
af23560	21.7606	18.9078	CRS	3.748958
af_1_k101	26.067	23.4946	CRS	4.245709
amazon0302	7.29883	5.23835	CRS	3.113809
bcsstk17	25.5293	23.3597	CRS	4.080255
cage4	0.385164	0.334653	CRS	0.441890
cant	29.7794	27.6956	CRS	4.638745
cavity10	19.0148	16.7258	CRS	3.152232
cop20k_A	19.5918	17.8729	CRS	4.105772
dc1	10.4524	9.39085	CRS	2.230150
lung2	8.832	6.85328	CRS	1.905592
mac_econ_fwd500	10.5515	8.84879	CRS	2.863691
mcfe	12.6097	11.8456	CRS	2.077682
mhd4800a	18.3501	16.1582	CRS	3.352021
mhda416	7.06099	6.80262	CRS	1.278130
nlpkkt80	24.4853	21.694	CRS	4.127566
olafu	29.966	27.9111	CRS	4.623662
olm1000	2.95895	2.1558	CRS	0.619634
raefsky2	27.8548	25.7504	CRS	4.248730
rdist2	15.5283	12.2527	CRS	2.694487
roadNet-PA	4.63584	3.01227	CRS	1.703601
thermal1	11.7254	8.83925	CRS	2.549122
thermal2	8.84723	6.75174	CRS	2.758931
thermomech_TK	10.9908	8.10205	CRS	3.088284
webbase-1M	6.11469	5.10783	CRS	1.630393
west2021	3.89256	2.92762	CRS	0.930255

Table 3.3: Comparaison des Performances des Algorithmes CRS et ELLPACK avec CUDA

Matrix	**CRS**	**ELLPACK**	**Best Structure**	**Speedup**
amazon0302	12.3120	10.4822	CRS	5.252515
cage4	0.033586	0.031649	CRS	0.038533
lung2	10.2714	9.81548	CRS	2.216157
mac_econ_fwd500	15.6609	15.4014	CRS	4.250389
mhd4800a	21.9362	21.4274	CRS	4.007095
olm1000	1.89504	1.48348	CRS	0.396840
raefsky2	70.6624	69.7615	CRS	10.778230
roadNet-PA	8.07317	5.84257	CRS	2.966768
thermal1	15.5753	13.318	CRS	3.386097
thermal2	19.2541	17.0523	CRS	6.004222
thermomech_TK	15.3241	14.2953	CRS	4.305889
west2021	2.66929	2.26954	CRS	0.637915
Cube_Coup_dt0	111.963	113.348	ELLPACK	17.953161
FEM_3D_thermal1	32.0555	33.4671	ELLPACK	5.634524
ML_Laplace	176.774	188.886	ELLPACK	28.493590
PR02R	135.828	146.651	ELLPACK	22.294501
adder_dcop_32	1.20824	4.12316	ELLPACK	0.920089
af23560	29.2923	29.4083	ELLPACK	5.066518
af_1_k101	79.1365	80.4176	ELLPACK	13.098161
bcsstk17	42.2982	45.3715	ELLPACK	7.251562
cant	99.3766	110.127	ELLPACK	17.154513
cavity10	21.571	23.0165	ELLPACK	3.815625
cop20k_A	46.062	47.6264	ELLPACK	9.980867
dc1	0.733577	14.7532	ELLPACK	3.147779
mcfe	10.5186	11.2408	ELLPACK	1.852130
mhda416	4.46372	4.6875	ELLPACK	0.848498
nlpkkt80	67.4344	67.5926	ELLPACK	11.394303
olafu	66.3819	69.052	ELLPACK	10.654512
rdist2	15.0779	15.7503	ELLPACK	2.733008
webbase-1M	8.04237	8.68042	ELLPACK	2.314507

Table 3.4: Comparaison des Performances entre CUDA, OpenMP, et Séquentiel

Matrix	**Best Performance**	**Best Structure**	**Best Method**	**Speedup**
amazon0302	12.3120	CRS	CUDA	5.252515
lung2	10.2714	CRS	CUDA	2.216157
mac_econ_fwd500	15.6609	CRS	CUDA	4.250389
mhd4800a	21.9362	CRS	CUDA	4.007095
raefsky2	70.6624	CRS	CUDA	10.778230
roadNet-PA	8.07317	CRS	CUDA	2.966768
thermal1	15.5753	CRS	CUDA	3.386097
thermal2	19.2541	CRS	CUDA	6.004222
thermomech_TK	15.3241	CRS	CUDA	4.305889
Cube_Coup_dt0	113.348	ELLPACK	CUDA	17.953161
FEM_3D_thermal1	33.4671	ELLPACK	CUDA	5.634524
ML_Laplace	188.886	ELLPACK	CUDA	28.493590
PR02R	146.651	ELLPACK	CUDA	22.294501
af23560	29.4083	ELLPACK	CUDA	5.066518
af_1_k101	80.4176	ELLPACK	CUDA	13.098161
bcsstk17	45.3715	ELLPACK	CUDA	7.251562
cant	110.127	ELLPACK	CUDA	17.154513
cavity10	23.0165	ELLPACK	CUDA	3.815625
cop20k_A	47.6264	ELLPACK	CUDA	9.980867
dc1	14.7532	ELLPACK	CUDA	3.147779
nlpkkt80	67.5926	ELLPACK	CUDA	11.394303
olafu	69.0520	ELLPACK	CUDA	10.654512
rdist2	15.7503	ELLPACK	CUDA	2.733008
webbase-1M	8.68042	ELLPACK	CUDA	2.314507
adder_dcop_32	5.12204	CRS	OpenMP	1.142991
mcfe	12.6097	CRS	OpenMP	2.077682
mhda416	7.06099	CRS	OpenMP	1.278130
cage4	0.871628	CRS	Serial	1.000000
olm1000	4.77532	CRS	Serial	1.000000
west2021	4.1844	CRS	Serial	1.000000

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).

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.