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.

Acknowledgements

The author would like to thank ...

Table of Contents

Abstract ii

Acknowledgements iii

Table of Contents iv

List of Figures v

List of Tables vi

1 Introduction 1

2 Methodology 2
2.1 Sequential Algorithm . . . 2
2.1.1 Complexity Analysis . . . 2
2.1.1.1 Temporal Complexity . . . 2
2.1.1.2 Spatial Complexity . . . 3
2.1.2 Example . . . 4
2.2 Line-Based Parallelism . . . 5
2.2.1 Complexity Analysis . . . 5
2.2.1.1 Temporal Complexity . . . 5
2.2.1.2 Spatial Complexity . . . 6
2.3 Column-Wise Parallelism . . . 7
2.3.1 Complexity Analysis . . . 7
2.3.1.1 Temporal Complexity . . . 7
2.3.1.2 Spatial Complexity . . . 8
2.4 . . . 9
2.4.1 Complexity Analysis . . . 9
2.4.1.1 Temporal Complexity . . . 9
2.4.1.2 Spatial Complexity . . . 10

3 Conclusion 11

A Documentation 12
A.A Project tree . . . 12
A.B Getting Started . . . 12
A.C Detailed Features of Functions . . . 13

List of Figures

List of Tables

Chapter 1

Introduction

Chapter 2

Methodology

2.1 Sequential Algorithm

The algorithm for multiplying a sparse matrix by a dense vector can be efficiently implemented using the Compressed Sparse Row (CSR) format. The CSR format represents a sparse matrix using three arrays: `values`, `col_indices`, and `row_pointers`. Given a sparse matrix M in CSR format and a dense vector v, the product $M \times v$ is computed as follows:

Algorithm 1 Sequential algorithm

Require: M is an $m \times n$ sparse matrix
Require: v is an $n \times k$ dense 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 $l \leftarrow 0$ **to** $k-1$ **do**
        $Result[i][l] \leftarrow Result[i][l] + (\text{value} \times v[j][l])$
      end for
    end for
  end for
  return $Result$

2.1.1 Complexity Analysis

The complexity analysis of the algorithm for multiplying a sparse matrix by a dense vector focuses on two main aspects: temporal complexity and spatial complexity.

2.1.1.1 Temporal Complexity

The temporal complexity of the algorithm depends on how the sparse matrix is stored and the number of non-zero elements in the matrix.

- **Traversing Rows:** The algorithm traverses each row of the matrix. If the matrix has m rows, this step has a complexity of $O(m)$.

- **Traversing Non-Zero Elements:** Inside each row, the algorithm traverses the non-zero elements. If the total number of non-zero elements in the matrix is n_{nz}, the traversal of all these elements has a complexity of $O(n_{nz})$.

The total temporal complexity is therefore $O(m + n_{nz})$. However, in practice, this complexity is often considered as $O(n_{nz})$, as the number of non-zero elements is usually the dominating factor, especially in very sparse matrices.

2.1.1.2 Spatial Complexity

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

- **Storing the Sparse Matrix:** The way the sparse matrix is stored affects the spatial complexity. Generally, storage formats like CSR or COO allow storing a sparse matrix with a complexity of $O(n_{nz})$, where n_{nz} is the number of non-zero elements.
- **Dense Vector:** The dense vector has a spatial complexity of $O(n)$, where n is the size of the vector.
- **Result Vector:** The result vector also has a size of $O(m)$, where m is the number of rows in the matrix.

The total spatial complexity is therefore $O(n_{nz} + m + n)$, but in practice, the focus is mainly on the $O(n_{nz})$ term as it is generally the most significant.

2.1.2 Example

Dans le format CSR, la matrice est représentée par trois vecteurs : `values`, `rows`, et `cols`. Pour notre exemple, ces vecteurs sont définis comme suit:

- `values` = $\{1, 2, 3, 4\}$
- `rows` = $\{0, 2, 3, 3, 4\}$
- `cols` = $\{0, 2, 1, 3\}$

La matrice creuse correspondante peut être visualisée comme:

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

Le vecteur dense est simplementq:

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

La multiplication de la matrice creuse par le vecteur dense est effectuée ligne par ligne. Le résultat peut être visualisé comme :

$$\begin{bmatrix} 1 & 0 & 2 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1\times1+0\times2+2\times3+0\times4 \\ 0\times1+0\times2+3\times3+0\times4 \\ 0\times1+0\times2+0\times3+0\times4 \\ 0\times1+0\times2+0\times3+4\times4 \end{bmatrix} = \begin{bmatrix} 7 \\ 9 \\ 0 \\ 16 \end{bmatrix}$$

2.2 Line-Based Parallelism

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

Algorithm 2 Line-based parallel sparse matrix-vector multiplication

Require: M is an $m \times n$ sparse matrix
Require: v is an $n \times k$ vector
Require: *numProcs* is the number of processes
Require: *rank* is the rank of the current process
Ensure: *Result* is a part of the $m \times k$ matrix computed by this process
  $rowsPerProc \leftarrow m/numProcs$
  $startRow \leftarrow rank \times rowsPerProc$
  $endRow \leftarrow startRow + rowsPerProc$
  $Result \leftarrow$ zero matrix of size $rowsPerProc \times k$
  for $i \leftarrow startRow$ **to** $endRow - 1$ **do**
    for each non-zero element (j, value) in row i of M **do**
      for $l \leftarrow 0$ **to** $k - 1$ **do**
        $Result[i - startRow][l] \leftarrow Result[i - startRow][l] + (\text{value} \times v[j][l])$
      end for
    end for
  end for
  if $rank \neq 0$ **then**
    Send *Result* to process 0
  else
    $FinalResult \leftarrow$ zero matrix of size $m \times k$
    Copy *Result* into $FinalResult$
    for $p \leftarrow 1$ **to** $numProcs - 1$ **do**
      Receive partial *Result* from process p
      Copy received *Result* into appropriate position in $FinalResult$
    end for
  end if
  if $rank = 0$ **then return** $FinalResult$

2.2.1 Complexity Analysis

2.2.1.1 Temporal Complexity

- **MPI Initialisation and Rank and Size Determination:** This step is generally fast, with a complexity close to $O(1)$, as it mainly involves setup operations.
- **Calculation of the Number of Rows per Process:** This operation also has a complexity of $O(1)$ as it requires only simple arithmetic based on the total size of the matrix and the number of processes.
- **Calculation of Start and End Indices for Each Process:** Again, this step has a complexity of $O(1)$ as it involves simple arithmetic calculations.

- **Scatter of Rows of M and Vector v:** The complexity of this step depends on the MPI implementation and data distribution. In general, it can be considered as $O(\frac{n_{nz}}{p})$, where n_{nz} is the total number of non-zero elements in the matrix and p is the number of processes.
- **Local Computation in Each Process:** Each process performs the matrix-vector product computation for its assigned portion of the matrix. The complexity of this step is $O(\frac{n_{nz}}{p})$ in the ideal case where the non-zero elements are evenly distributed among the processes.
- **Gather of Local Results r_{local}:** The gather operation can vary in complexity, but generally, it is proportional to the total number of elements to be gathered and depends on the efficiency of communication between processes.

2.2.1.2 Spatial Complexity

- **Storage of Sparse Matrix and Dense Vector:** The storage of the matrix and vector remains $O(n_{nz}+m+n)$, where m and n are the dimensions of the matrix.
- **Local Result Vectors r_{local}:** Each process stores a local result vector of a size proportional to the portion of the matrix it processes, approximately $O(\frac{m}{p})$.

2.3 Column-Wise Parallelism

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

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

Require: M is an $m \times n$ sparse matrix
Require: v is an $n \times k$ vector
Require: *numProcs* is the number of processes
Require: *rank* is the rank of the current process
Ensure: $PartialResult$ is a part of the $m \times k$ matrix computed by this process
  $colsPerProc \leftarrow k/numProcs$
  $startCol \leftarrow rank \times colsPerProc$
  $endCol \leftarrow startCol + colsPerProc$
  $PartialResult \leftarrow$ zero matrix of size $m \times colsPerProc$
  for $i \leftarrow 0$ **to** $m-1$ **do**
    for each non-zero element (j, value) in row i of M **do**
      for $l \leftarrow startCol$ **to** $endCol - 1$ **do**
        $PartialResult[i][l - startCol] \leftarrow PartialResult[i][l - startCol] + (\text{value} \times v[j][l])$
      end for
    end for
  end for
  if $rank \neq 0$ **then**
    Send $PartialResult$ to process 0
  else
    $FinalResult \leftarrow$ zero matrix of size $m \times k$
    Copy $PartialResult$ into appropriate position in $FinalResult$
    for $p \leftarrow 1$ **to** $numProcs - 1$ **do**
      Receive partial $PartialResult$ from process p
      Copy received $PartialResult$ into appropriate position in $FinalResult$
    end for
  end if
  if $rank = 0$ **then return** $FinalResult$

2.3.1 Complexity Analysis

2.3.1.1 Temporal Complexity

- **MPI Initialisation and Rank and Size Determination:** Similar to the line-based approach, this step has a complexity of approximately $O(1)$, involving basic setup operations.
- **Distribution of Non-Zero Elements of M:** This step involves distributing the non-zero elements among the processes. The complexity depends on the distribution

mechanism but is generally proportional to the number of non-zero elements, n_{nz}, and the efficiency of the distribution algorithm used.

- **Scatter of Vector v to All Processes:** Scattering the vector v to all processes can be done efficiently in MPI and typically has a complexity proportional to the size of the vector, $O(n)$.
- **Computation of Products for Assigned Non-Zero Elements:** Each process computes the products for its assigned non-zero elements. Assuming an even distribution, the complexity for each process would be $O(\frac{n_{nz}}{p})$, where p is the number of processes.
- **MPI Atomic Operations and Reduction:** The use of atomic operations and reduction to form the final result vector can introduce additional complexity, depending on the implementation and efficiency of these operations in MPI.

2.3.1.2 Spatial Complexity

- **Storage of Sparse Matrix and Dense Vector:** The storage requirements remain the same as in the line-based approach, $O(n_{nz}+m+n)$.
- **Local Result Vectors r_{local}:** Each process maintains a local result vector, but since the computation is based on non-zero elements, the storage requirement for each r_{local} might be smaller, depending on the distribution of non-zero elements.

2.4 Non-Zero Element Parallelism

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

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

Require: M is an $m \times n$ sparse matrix stored in a format that allows iterating over non-zero elements (e.g., COO, CSR)
Require: v is an $n \times k$ vector
Require: *numProcs* is the number of processes
Require: *rank* is the rank of the current process
Ensure: *PartialResult* is a part of the $m \times k$ matrix computed by this process
  $numNonZeroElements \leftarrow$ total number of non-zero elements in M
  $elementsPerProc \leftarrow numNonZeroElements / numProcs$
  $startIndex \leftarrow rank \times elementsPerProc$
  $endIndex \leftarrow startIndex + elementsPerProc$
  $PartialResult \leftarrow$ zero matrix of size $m \times k$
  $NonZeroElements \leftarrow$ list of non-zero elements in M from $startIndex$ to $endIndex - 1$
  for each (i, j, value) in $NonZeroElements$ **do**
    for $l \leftarrow 0$ **to** $k - 1$ **do**
      $PartialResult[i][l] \leftarrow PartialResult[i][l] + (\text{value} \times v[j][l])$
    end for
  end for
  if $rank \neq 0$ **then**
    Send *PartialResult* to process 0
  else
    $FinalResult \leftarrow$ zero matrix of size $m \times k$
    Copy *PartialResult* into *FinalResult*
    for $p \leftarrow 1$ **to** $numProcs - 1$ **do**
      Receive partial *PartialResult* from process p
      Add received *PartialResult* into *FinalResult*
    end for
  end if
  if $rank = 0$ **then return** *FinalResult*

2.4.1 Complexity Analysis

2.4.1.1 Temporal Complexity

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

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

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

2.4.1.2 Spatial Complexity

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

Complexity and Considerations

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

Chapter 3

Conclusion

Appendix A

Documentation

Appendix A.A Project tree

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

Appendix A.B Getting Started

To run the program, follow these steps:

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

Appendix A.C Detailed Features of Functions

`collecting.py`

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

`processing.py`

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

`storing.py`

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