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

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

```
1:  procedure SparseMatrixVectorMult(M, v)
2:      Initialise a result vector r of appropriate size
3:      for each row i of M do
4:          Determine the start and end of row i in M
5:          for each non-zero element M_ij in row i do
6:              Find the corresponding column index j in M
7:              Compute the product of M_ij and v_j
8:              Add the product to the i-th element of r
9:          end for
10:     end for
11:     return r
12: end procedure
```

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 divides the sparse matrix into chunks of rows, distributing these chunks across multiple threads or processes.

Algorithm 2 Line-based parallel sparse matrix-vector multiplication

```
1:  procedure ParallelLineBasedMult(M, v, numThreads)
2:      Divide M into numThreads row chunks
3:      Initialise a result vector r of appropriate size
4:      for each thread t do
5:          Assign a row chunk to thread t
6:          Thread t computes multiplication for its chunk
7:          Accumulate results in r
8:      end for
9:      return r
10: end procedure
```

2.2.1 Complexity Analysis

2.2.1.1 Temporal Complexity

- **Dividing the Matrix into Chunks:** The complexity of dividing the matrix M into `numThreads` chunks is $O(m)$, where m is the number of rows in the matrix. This step involves assigning row ranges to each thread and is relatively lightweight in terms of computation.
- **Multiplication within Each Thread:** Each thread processes its assigned chunk of the matrix. Assuming n_{nz} is the total number of non-zero elements in the matrix and the elements are evenly distributed, the complexity for this part is approximately $O\left(\frac{n_{nz}}{\texttt{numThreads}}\right)$ in an ideal parallel environment.
- **Accumulation of Results:** The final step of accumulating the results in the vector r can be executed in parallel and does not significantly add to the complexity, assuming ideal conditions.

2.2.1.2 Spatial Complexity

The spatial complexity is influenced by the storage requirements for the sparse matrix and the result vector.

- **Sparse Matrix Storage:** Storing the sparse matrix in CSR (Compressed Sparse Row) format requires $O(n_{nz} + m + n)$ space, where n_{nz} is the number of non-zero elements, m is the number of rows, and n is the number of columns.
- **Result Vector Storage:** The result vector r of size m has a spatial complexity of $O(m)$.
- **Temporary Storage in Threads:** Additional temporary storage for computations within each thread does not significantly affect the overall spatial complexity.

2.2.1.3 Overall Complexity

- **Temporal Complexity:** The overall temporal complexity of the algorithm is $O(m + \frac{n_{nz}}{\texttt{numThreads}})$, under the assumption of an ideal parallel environment.
- **Spatial Complexity:** The overall spatial complexity is $O(n_{nz} + m + n)$, dominated by the storage requirements for the sparse matrix and the result vector.

2. Non-Zero Element-Based Parallelism

This approach distributes the computation based on the non-zero elements of the sparse matrix.

Algorithm 3 Element-based parallel sparse matrix-vector multiplication

```
1:  procedure ParallelElementBasedMult(M, v, numThreads)
2:      Divide non-zero elements of M into numThreads parts
3:      Initialise a result vector r of appropriate size
4:      for each thread t do
5:          Assign a subset of non-zero elements to thread t
6:          Thread t computes contributions to r
7:          Perform atomic addition to accumulate in r
8:      end for
9:      return r
10: end procedure
```

3. Hybrid Parallelism (Lines and Non-Zero Elements)

This method combines the first two approaches: it divides the matrix into row chunks and further parallelizes by distributing the non-zero elements of each chunk.

Algorithm 4 Hybrid parallel sparse matrix-vector multiplication

```
1: procedure HybridParallelMult(M, v, numThreads)
2:     Divide M into row chunks
3:     Initialise a result vector r of appropriate size
4:     for each row chunk do
5:         Further divide non-zero elements among threads
6:         for each thread t do
7:             Assign a subset of non-zero elements to thread t
8:             Thread t computes contributions to r
9:             Perform atomic addition to accumulate in r
10:        end for
11:    end for
12:    return r
13: end procedure
```

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.