Matrix Multiplication with Map Reduce

Joel Clemente López Cabrera

January 3, 2024

Abstract

Matrix multiplication is a fundamental operation in linear algebra with numerous applications across various domains. In this paper, we present an efficient MapReduce-based algorithm for matrix multiplication. Leveraging the capabilities of the MRJob library, the algorithm is designed to scale horizontally, making it suitable for large-scale distributed computing environments.

1 Introduction

Matrix multiplication is a computationally intensive task that plays a pivotal role in scientific computing, machine learning, and graph algorithms. The advent of big data has necessitated the development of scalable algorithms capable of handling massive datasets. MapReduce, a programming model for processing and generating large datasets, provides an attractive framework for tackling such problems.

This paper introduces a MapReduce-based matrix multiplication algorithm implemented using the MRJob library. The algorithm takes advantage of the inherent parallelism in the MapReduce paradigm, distributing the computation across multiple nodes for improved efficiency.

2 Implementation and Methodology

The algorithm is implemented in Python using the MRJob library, a powerful framework for building MapReduce algorithms. The code defines a MatrixMultiplication class, encapsulating the map and reduce functions, as well as the necessary configuration parameters.

2.1 Configuration Parameters

The algorithm takes two configuration parameters:

- `row_a`: Number of rows in matrix A.
- `col_b`: Number of columns in matrix B.

Having these two parameters, to execute the code we must write the following command in the console:

- python MatrixMultiplication.py input.txt `--row_a` 4 `--col_b` 4

2.2 Map Function

The map function reads input lines representing matrix entries, extracts matrix identifiers, row and column indices, and values. If the matrix identifier is 'A,' the function emits key-value pairs for each element in the resulting matrix C, where the key is a composite of the row index and column index, and the value is a tuple containing the column index and value of the corresponding element in A. If the matrix identifier is 'B,' the function emits key-value pairs in a similar fashion, using the row index instead.

```python
def mapper(self, _, line):
    matrix, row, col, value = line.strip().split(",")

    if matrix == "A":
        for i in range(self.options.col_b):
            key = f"{row},{i}"
            yield key, (col, value)
    else:
        for j in range(self.options.row_a):
            key = f"{j},{col}"
            yield key, (row, value)
```

Figure 1: Mapper

2.3 Reduce Function

The reduce function processes the intermediate key-value pairs by grouping them based on the composite key. It then sorts the grouped values based on the column index. The function iterates through the sorted values, multiplying adjacent elements with the same column index and summing the results to compute the final value for the corresponding element in matrix C. The final key-value pair is emitted, representing a unique element in the resulting matrix C.

```python
def reducer(self, key, values):
    value_list = list(values)
    value_list = sorted(value_list, key=itemgetter(0))
    i = 0
    result = 0

    while i < len(value_list) - 1:
        if value_list[i][0] == value_list[i + 1][0]:
            result += int(value_list[i][1]) * int(value_list[i + 1][1])
            i += 2
        else:
            i += 1

    yield key, result
```

Figure 2: Reducer

3 Results and Performance Analysis

The algorithm exhibits promising results in terms of scalability and computational efficiency. The use of MapReduce allows for distributed processing, enabling the algorithm to handle large matrices that may not fit into the memory of a single machine. Additionally, the algorithm's parallel nature enhances its performance, making it suitable for deployment on cloud-based computing platforms.

Performance metrics, such as execution time and scalability, were evaluated using synthetic datasets of varying sizes. The algorithm demonstrated linear scalability with an increasing number of nodes in the MapReduce cluster. The results indicate that the proposed algorithm is a viable solution for large-scale matrix multiplication tasks.

4 Conclusion

This paper presented a MapReduce-based algorithm for matrix multiplication, showcasing its implementation using the MRJob library. The algorithm leverages the parallelism inherent in the MapReduce paradigm, making it well-suited for large-scale distributed computing environments.

The results of the performance analysis demonstrate the algorithm's efficiency and scalability. Future work could focus on further optimizations and adaptations for specific distributed computing frameworks, as well as exploring potential applications in real-world scenarios.

5 Future Work

The proposed algorithm lays the foundation for future research in several directions:

- **Optimizations:** Explore optimization techniques to further enhance the algorithm's efficiency and reduce computation time.
- **Framework Integration:** Adapt the algorithm for compatibility with other distributed computing frameworks, such as Apache Hadoop or Apache Spark.
- **Real-world Applications:** Investigate the applicability of the algorithm in real-world scenarios, such as large-scale data analytics and scientific simulations.
- **Algorithmic Enhancements:** Explore alternative algorithms or improvements to address specific challenges associated with matrix multiplication in distributed environments.

6 Matrix Format

The matrices used in the implementation follow the coordinate pattern format. Each line represents an entry in the matrix with the format:

$$MatrixID, RowIndex, ColumnIndex, Value$$

For example:

$$A, 0, 0, 2$$

$$A, 1, 0, 0$$

$$A, 2, 0, 9$$

$$A, 3, 0, 3$$

$$A, 0, 1, 1$$

$$B, 0, 0, 6$$

$$B, 1, 0, 4$$

$$B, 2, 0, 1$$

$$B, 3, 0, 4$$

$$B, 0, 1, 1$$

$$\ldots$$