Comparative performance analysis between sequential and MapReduce matrix multiplication

SUSANA SUÁREZ MENDOZA[1]

[1] *University of Las Palmas de Gran Canaria, Data Science and Engineering Degree*
[1] *Las Palmas, Canarias, 35001, Spain*

Compiled January 13, 2024

This study focuses on evaluating and comparing two approaches to matrix multiplication: one sequential and one based on the distributed, parallel data processing technology known as "MapReduce". Matrix multiplication is essential in computer applications, from numerical simulations to machine learning, and its performance varies depending on the approach. We implement these approaches in Python with MRJob for MapReduce and Java with Hadoop for the Java implementation. In Python, we define and run MapReduce jobs, while in Java we set up a Hadoop project with mappers and reducers. The results reveal that MapReduce is not optimal on a single computer, suggesting distributed testing. Furthermore, the parallelisation approach stood out as the most efficient among sequential, optimised sequential, sparse arrays and MapReduce, taking full advantage of the available threads.
In summary, this study contributes to computing and data processing by identifying limitations and advantages of the evaluated approaches. It provides useful information for data scientists and practitioners to choose methods according to their needs, promoting experiment replication and transparency in the scientific community, driving the advancement of knowledge in a constantly evolving field.

Keywords: Matrix multiplication, parallelisation, java, python, performance, speed, MapReduce

1. INTRODUCTION

Matrix multiplication represents a fundamental pillar in the field of scientific computing, exerting a determining influence in numerous fields, from the resolution of linear systems to the development of artificial intelligence and machine learning algorithms. Efficiency in this operation stands as a primary goal in software research and development, given its direct impact on the performance of applications and algorithms that intrinsically depend on this operation.

This article focuses on the comparison of matrix multiplication in two main modes: sequential and using the MapReduce algorithm, and in two languages: Python and Java. These variants have significant implications for the performance of matrix operations, especially when dealing with large datasets, with a large difference in execution time. Specifically, this study provides a detailed and comparative analysis of the performance between matrix multiplication in distributed, parallel data processing and the sequential method. We seek to establish the relative effectiveness of these techniques in terms of speed and efficiency in handling dense matrices.

Our main contribution is to provide a rigorous quantitative assessment backed by comprehensive experiments. These results guide researchers and practitioners in making informed decisions about which approach to take based on the particularities of their applications and algorithms. In addition, the results can have a direct impact on optimising algorithms and improving performance in various areas of computer and data science.

2. PROBLEM STATEMENT

MapReduce is a programming model and data processing system designed to process large data sets in a distributed and parallel manner. It was developed by Google and is widely used in industry to perform large-scale data analysis. The MapReduce programming model is based on two main operations: "map" and "reduce". Here is a brief description of each of them to perform matrix multiplication:

1. **Mapping:** In this phase, the input data is divided into smaller chunks and applied to a mapping function. This function takes each data element and generates a list of key-value pairs as output. The mapping is performed in parallel on several nodes or servers in a computer cluster.

2. **Combiner:** Optional phase in the MapReduce model that occurs after the mapping phase and before the reduction phase. Its main purpose is to reduce the amount of data that is transmitted between cluster nodes and to improve the efficiency of the reduction process. The merge phase is not present in all MapReduce jobs, and its use depends on the specific application and implementation.

3. **Reduce:** In this phase, data is grouped by key and processed by a custom reduction function. Each group of data with the same key is passed to the reduce function, which may perform additional calculations or summarise the data in some way.

In summary, MapReduce has become an essential tool for the efficient processing of large volumes of data in distributed environments. The combination of map, combine (if used) and reduce phases allows complex tasks, such as matrix multiplication, to be tackled in a scalable and parallel manner. This programming architecture provides businesses and data scientists with a powerful tool to perform large-scale data analysis and processing, facilitating informed decision making and the discovery of valuable information from massive datasets. MapReduce's flexibility and scalability continue to be valuable in the field of distributed computing and big data processing today.

3. METHODOLOGY

To address the problem posed in this study, a methodology section is developed detailing the multiplication process using the MapReduce framework in two different programming languages: Python with the MRJob library and Java with the Hadoop dependency. First, in the Python implementation, we have used MRJob to define and execute the MapReduce job, specifying the necessary mapping and reduction functions. As for the Java implementation, we have configured and executed a Hadoop project, defining the corresponding mappers and reducers in Java. For both cases, we provide a set of detailed instructions and code examples that allow the reader to reproduce the multiplication process in a MapReduce environment in Python and Java on the Github platform. In this way, we ensure the replicability of our approach and facilitate the understanding and application of our methodology.

4. EXPERIMENTS

A. Hardware and IDE

The experiments were carried out on a single computer system equipped with an Intel Core i7 processor, 15.7 GB of available RAM and a 64-bit Windows operating system. This system has a total of 4 virtual cores with 8-thread capacity, which the operating system can assign to processing tasks. The selection of this hardware provided an ideal platform for the performance evaluation of various matrix multiplication operations. In addition, the algorithms were executed within the IntelliJ integrated development environment for the Java programming language, using the Hadoop dependency to perform the multiplication and in Visual Studio Code for the Python programming language using the MrJob library.

B. Source code

Despite the particularities inherent to each language, it is possible to structure the code in a general way, as will be detailed below. It is important to note that the use of interfaces has been implemented exclusively in the Java language, although Python also includes the same class.

- **Matrix module:** contains all the matrix models that we will be working with in the project: dense matrix and coordinate matrix. Figure 1 shows the attributes and dependencies of that module.

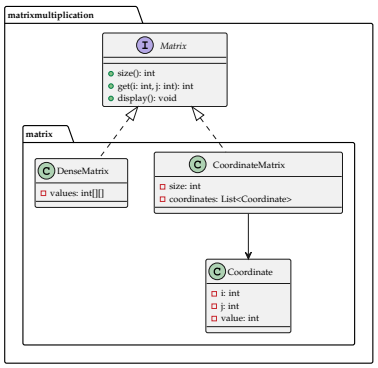


Fig. 1. Matrix Module Class Diagram

- **MatrixBuilder module**: Contains the class that generates a coordinate square matrix according to its dimension and density and fills it with random integers. In addition, this module contains the class in charge of transforming from coordinate matrix to dense matrix.

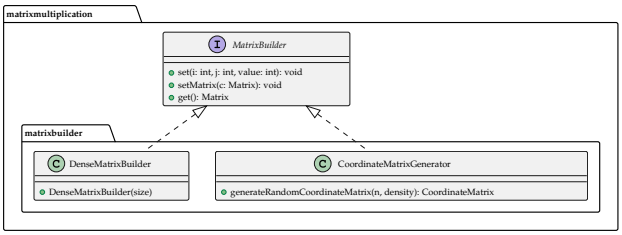


Fig. 2. Matrix Builder Module Class Diagram

- **Writer module:** This module hosts the class responsible for the generation of text files (in the case of Python) or two text files (in the case of Java) containing the matrices for further processing using the multiplication algorithm in MapReduce format.

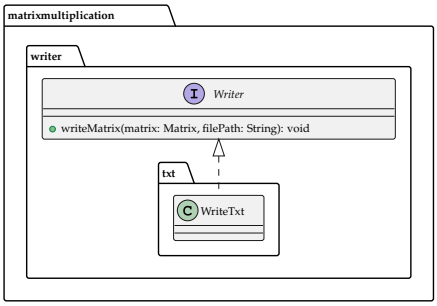


Fig. 3. Writer Module Class Diagram

- **Operators:** This module is composed of two different submodules. One of them is in charge of carrying out the matrix multiplication in a sequential way, while the other one is in charge of executing the multiplication using the

"MapReduce" algorithm according to the specific framework of each programming language. It has been decided to represent only the Java language as it is the most difficult to understand.

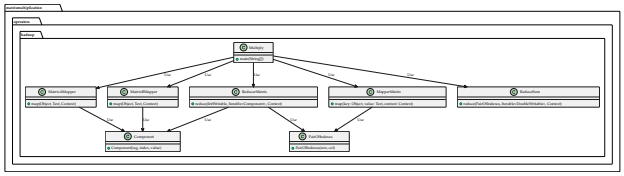


Fig. 4. Hadoop Java Module Class Diagram

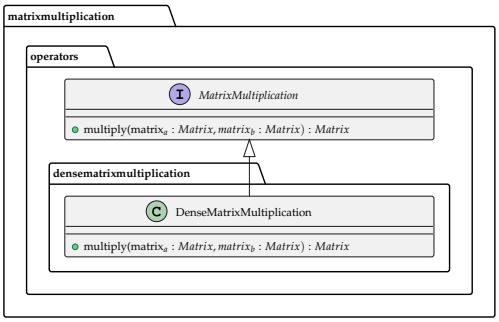


Fig. 5. Dense Matrix Multiplication Module Class Diagram

- **Checker module**: an extra module whose function is to check that the multiplication methods work correctly using the following associative property of arrays:

$$(A \times B) \times C = A \times (B \times C)$$

C. Tests and results

This section tries to implement and implement the different tests on the various situations. A total of four tests have been carried out, of which three have been measured.

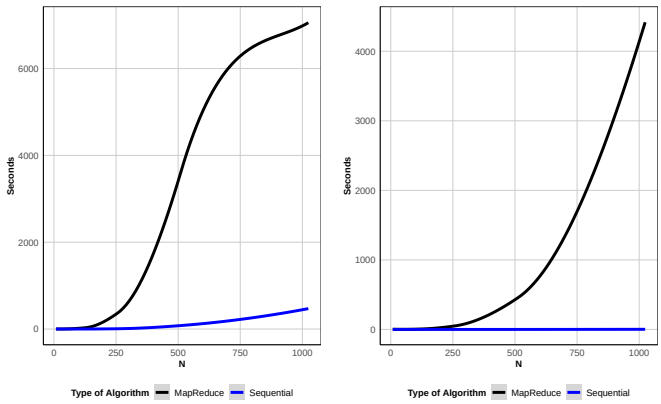
C.1. Multiplication done right

When having different multiplications of such large matrices, it must be taken into account that the method implemented to perform the multiplication is correct. This is done by means of the Checker class which contains the implementation of the associative property of three matrices and its comparison between both results. Finally, in the various executions, a true result has been obtained, which means that the method works correctly.

C.2. Dense Multiplication vs MapReduce on a single computer

In this study, a comparison between dense multiplications and those performed using the MapReduce framework on a single computer has been carried out, as detailed in the Hardware section of the report. It is important to take into consideration that the speed of execution of these multiplications may vary depending on the programming language used. In addition, it should be noted that the operations carried out using the MapReduce algorithm tend to be more expensive in a monolithic execution environment, given that the algorithm is designed to run on distributed systems, which is not available in the test environment used in this study.

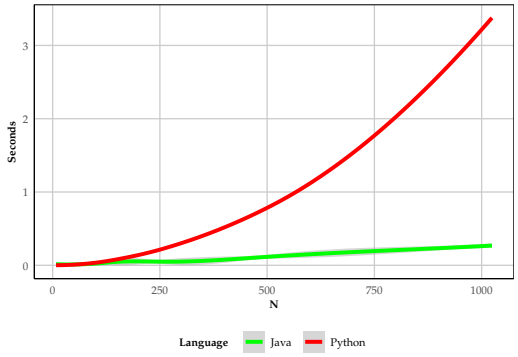
It is relevant to note that, in general terms, it has been observed that the Python programming language exhibits slower performance compared to other languages, and that non-traditional multiplication requires longer execution time due to the previously mentioned reasons.



(a) Python Multiplication **(b)** Java Multiplication

C.3. Writing time

Additionally, it is important to take into account a time factor that is not reflected in the previous graphs, and is related to the MapReduce algorithm. This algorithm requires additional time due to the write operation required in the process. This write is necessary because the program is designed to read a file containing the coordinates and values of the matrices to be multiplied. Initially, one might have considered that this write time would have a significant impact on performance. However, observations show that, while there is an impact, it is not as substantial as anticipated. Furthermore, it is important to note that the Python programming language is still considerably slower than Java in this context.

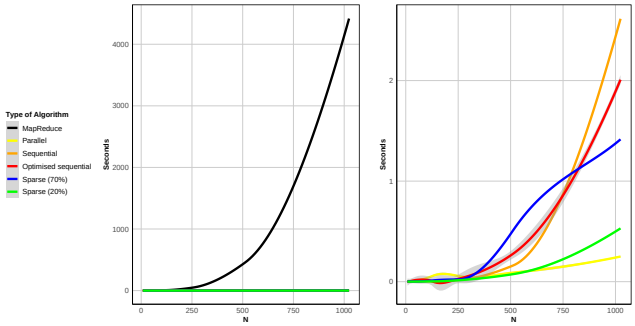


C.4. Compilation of all kinds of multiplication formats in Java

Finally, as a summary of all the previous articles, it is critical to determine which algorithm or matrix format is most efficient in the context of a single system. It is important to note that, as mentioned above, the performance of MapReduce tends to increase significantly as the scale of the distributed system on which it operates increases. However, in the context of a single system, it is essential to evaluate and compare the different alternatives available to determine which one best fits the specific needs and constraints of that environment.

It should be noted that, according to the observations made, the MapReduce algorithm is the slowest algorithm in the context of dense matrix multiplication. This is due to the previously mentioned characteristics that influence its performance in this

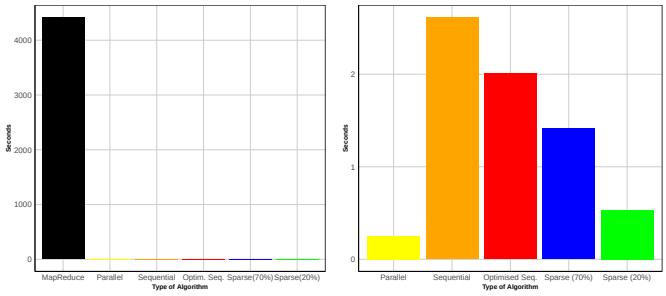
type of matrices. However, when analysing the other algorithms together with their respective formats (as shown in the figure 8b), the efficiency of parallel multiplication stands out, especially when applied to matrices with a density of 70%. Furthermore, it is observed that multiplication with matrices in Sparse format is highly efficient, especially when the density of the matrix is low, as in the case of a density of 20%. In summary, in a single-system execution environment, the preference is to parallelise matrix operations to maximise thread usage and obtain optimal performance.



(a) All of Algorithms **(b)** Without MapReduce

Fig. 7. Matrix multiplication algorithms

In the context of the multiplication of 1024x1024 matrices, the previous observation is repeated, where MapReduce demonstrates a remarkable slowness. Likewise, when considering other types of matrices, the conclusions remain consistent, highlighting the effectiveness of the parallelism approach.



(a) All of Algorithms **(b)** Without MapReduce

Fig. 8. 1024x1024

5. CONCLUSIONS

In the course of this research, we have been challenged to evaluate and compare the performance of two different approaches to matrix multiplication: one sequential and the other based on the distributed, parallel data processing technology known as "MapReduce". Matrix multiplication, a fundamental operation in various fields of computer science from numerical simulation to machine learning, is a critical component in many applications. However, its performance can vary significantly depending on how it is approached, either sequentially or in parallel.

On the one hand, test results indicate that the algorithm format known as "MapReduce" is not optimised to run on a single computer, suggesting the need for testing on a distributed system to realise its full potential. In addition, it is important to note that this programming requires text files describing the matrices, although the time taken to write such files does not seem to be a relevant factor.

On the other hand, when compiling and analysing all the algorithms evaluated, the parallelisation approach stands out in particular, taking full advantage of the number of threads available. Among all the tested approaches, which include sequential, optimised sequential, sparse arrays and MapReduce, parallelism stands out as the most prominent option, without a doubt.

The importance of this experimentation lies in its contribution to the field of computing and data processing. By identifying the limitations and advantages of these approaches, we provide data scientists and practitioners with valuable information for making informed decisions on when and how to apply each method according to the needs of their specific applications. Furthermore, by facilitating the replication of our experiments and promoting transparency in the scientific community, we contribute to the advancement of knowledge in this evolving field.

6. FUTURE WORK

In the context of future research and work, two crucial aspects stand out. First, it is imperative to expand this study by implementing the MapReduce algorithm in a distributed environment to accurately assess its performance in real-world conditions where larger and more complex datasets are handled. This extension will allow a more complete understanding of the effectiveness of MapReduce in practical applications and the need for its execution in distributed systems.

Secondly, there is potential for improvement and optimisation in the codes used in this study. The constant evolution in algorithm optimisation and the search for efficiencies suggests the opportunity to review and refine the codes, which could result in improved performance in both the sequential approach and the MapReduce approach. These improvements may include implementing advanced parallelisation techniques, simplifying algorithms and identifying areas for reducing execution times.

In addition to the improvements mentioned above, other areas for future research include exploring new approaches to matrix multiplication and evaluation on various hardware architectures. Studying other specific algorithms and strategies for this operation, as well as testing the approaches on different hardware configurations, such as multiprocessor systems and systems with hardware accelerators, can provide a more complete and versatile understanding of the available options and their advantages in varied applications.

In summary, the future of this research focuses on the continued expansion, improvement and exploration of approaches and codes for matrix multiplication. These efforts will contribute to the advancement of computing and data processing, providing more efficient and effective solutions for various applications.

7. REFERENCES

- Github Repository. The repository code is divided into three branches: the Main master branch contains the python matrix multiplication code, the Java branch contains the java matrix multiplication code and the Itemset branch contains the java Frequent-item-set algorithm code.

OPTIONAL ADDITIONAL SECTION: ITEM-SET-ALGORITHM

In the previously mentioned article, the application of the MapReduce algorithm in matrix multiplication was discussed. However, it is important to note that this is not its only area of application. Among numerous other contexts, MapReduce algorithms can be used in recommender systems, such as the Frequent Itemset algorithm. The latter, widely used in the field of data mining and pattern analysis, has as its main objective the identification of sets of items that exhibit a significant presence in a transactional database. In essence, this algorithm is oriented towards the identification of groups of elements that frequently co-occur, allowing the discovery of interesting relationships between them. To accomplish this task, it makes use of metrics such as support, which quantifies the frequency with which a particular set of items appears. The application of the Frequent Itemset algorithm is essential in a number of areas, such as market analysis to uncover buying patterns, as well as in recommender systems to identify related products or items that may be of interest to a user.

The ItemSetMapReduce class plays a fundamental role in the implementation of the MapReduce process, and its key components are described as follows:

- **ItemSetMapReduce:** This class extends the Configured class and implements the Tool interface. This approach is typical in MapReduce programs to take advantage of the configuration capability provided by the Hadoop framework and use the ToolRunner utility to run the program efficiently.
- **Static variables:** The SET_SIZE and MIN_COUNT constants are used as essential parameters in the Frequent Itemset algorithm. SET_SIZE defines the minimum size of the item sets being analysed, while MIN_COUNT sets the minimum count required for a set to be considered frequent.
- **Main method:** This method serves as the entry point of the program and uses ToolRunner to launch the MapReduce job. It configures and executes the tasks necessary to process the data, taking advantage of the default configuration provided by Hadoop.
- **Run method:** Implements the Tool interface and is fundamental for the execution of the MapReduce job. In this method, an instance of the Job class is created to configure and manage the workflow. Several important properties are defined, such as the main class, the mapper and the reducer, as well as the output key and value types, and the data input and output paths.
- **ItemSetMapper class:** This inner class is responsible for implementing the custom mapper for the MapReduce job. In its main function, it takes an input line and splits it into individual items. It then generates all possible item sets of size SET_SIZE and outputs them with a value of 1, which is essential for frequent set analysis.
- **ItemSetReducer inner class:** This inner class implements the custom reducer, in charge of processing the data emitted by the mapper. Its task is to sum up the values associated to each set of elements and only emit those sets that exceed the minimum count specified in MIN_COUNT. This step is essential for the identification of frequent itemsets in the analysed dataset.

Tested examples

An experiment was carried out in which the products contained in each individual's shopping basket were recorded, as shown in the table on the left. The aim of the algorithm was to identify relationships between items by considering sets of two items (2 to 2) occurring with a frequency greater than 2, which was set as the minimum threshold required to be considered meaningful. The algorithm seems to have tackled the fruit-related task effectively, and has provided the following information, presented in the table on the right.

Products
apple banana orange strawberry
banana orange pineapple
lemon cherries
banana orange strawberry lemon

Table 1. Shopping basket

Relation	Freq.
banana - orange	3
banana - strawberry	2
orange - strawberry	2

Table 2. Results

An additional experiment was carried out using different products than those purchased, instead of fruits. As anticipated, the algorithm performed successfully, meeting the expectations set for this task.

Products
Milk Chicken Beer
Chicken Cheese
Cheese Boots
Cheese Chicken Beer
Chicken Beer Clothes Cheese Milk
Clothes Beer Milk
Beer Milk Clothes

Table 3. Second shopping basket

Relation	Freq.
Beer - Cheese	2
Beer - Chicken	3
Beer - Clothes	3
Beer - Milk	4
Cheese - Chicken	3
Chicken - Milk	2
Clothes - Milk	3

Table 4. Second results