ULPGC

BD

Hadoop Matrix Multiplication

Victoria Torres Rodríguez[1] [1]*victoria.torres101@alu.ulpgc.es*

Abstract

Matrix multiplication is a fundamental computational problem in areas such as artificial intelligence, simulation, and data analysis. However, when dealing with large arrays, processing this task can be very costly in terms of time and resources. Hadoop is an open source distributed software system that enables the processing of large volumes of data on a computer network. The combination of Hadoop with the MapReduce algorithm allows the distribution of the matrix multiplication task in several nodes of the network, which allows a faster and more efficient execution in terms of resources.

The importance of matrix multiplication in Hadoop lies in its ability to process large data sets in a scalable and efficient manner. This is especially important in applications that require the processing of large amounts of data, such as machine learning, simulation, and data analysis. In addition, the use of Hadoop for matrix multiplication also enables the creation of distributed data processing solutions, which can significantly improve performance compared to single node based solutions. This method can be scalable in cloud applications such as GoogleCloud, which will be specified in this procedure.

In short, matrix multiplication in Hadoop is an essential technique for processing large volumes of data in a resource-efficient and scalable manner. This is especially important in applications that require large amounts of data to be processed and is a valuable tool for improving performance in distributed data processing solutions.

- *Keywords*: matrix multiplication, hadoop, MapReduce, data processing, distributed, scalability, efficiency, simulation, analysis of data, distributed data processing solutions, performance.

1 Exploring the limits of distributed programming

Topics like matrix multiplication are just a first step into the field of distributed programming. These types of techniques have attractive potential in BigData projects due to their favorable scalability, reduced dependencies between tasks, performance improvements, and the creation of flexible data processing solutions.

As an example, we can put the matrix multiplication problem in mosaic, it is a technique used to divide a matrix into smaller sub-matrices, with the aim of being able to perform parallel operations in a distributed environment. This technique is based on the idea of splitting a large matrix into several smaller sub-matrixes, and then multiplying these sub-matrixes together in parallel at different nodes in a distributed network. Later, we will explain this approach in depth. Currently, there are several authors who have researched this topic, including M. K. Qureshi, who presented an approach to matrix multiplication using Hadoop MapReduce. Other authors such as J. Dean and S. Ghemawat have presented an approach for matrix multiplication using the MapReduce algorithm in Google's Bigtable.

This paper focuses on the development of a solution for matrix multiplication in distributed environments using Hadoop. The design and implementation of a distributed matrix multiplication algorithm using the Hadoop MapReduce framework is discussed. In addition, the performance of the proposed solution is analyzed in comparison with other existing approaches.

In summary, this article focuses on the development of a solution for matrix multiplication in distributed environments using Hadoop and MapReduce, proving the incredible potential that these programming techniques can bring to future big data projects.

2 Problem Statement

The theme discussed in this research focuses on very specific topics that may come out of the reader's general knowledge. To do this, we will briefly explain the concepts of tiled matrix multiplication, distributed programming and the general structure of the Hadoop MapReduce framework.

2.1 Tiled Matrix Multiplication

Mosaic matrix multiplication is an algorithm that allows dividing a large matrix into several smaller submatrices, with the aim of being able to perform parallel operations in a distributed environment.

This algorithm makes it possible to take full advantage of the processing capacity of the different nodes of the network, thus increasing efficiency and reducing execution time. However, it is important to note that splitting arrays into a large number of sub-arrays may cause overhead and not significantly improve execution time. Also, it is important to mention that this general algorithm can be implemented in different distributed programming technologies like Hadoop, Spark, etc.

For example, suppose we want to multiply two matrices A and B, both of size 4x4. If we decide to split each matrix into 4 2x2 submatrices, the tiling process would look like this:

1. Matrix A is divided into 4 submatrices A1, A2, A3 and A4.
2. Matrix B is divided into 4 submatrices B1, B2, B3 and B4.
3. Each submatrix of A is multiplied with the corresponding submatrix of B at a different node in the distributed network.
4. For example, A1 is multiplied by B1 at one node, A2 is multiplied by B2 at another node, and so on.
5. The results of each submatrix multiplication are sent to the main or master node to be combined and form the resulting matrix C.
6. For example, the multiplication submatrix A1 x B1 is combined with the multiplication submatrix A2 x B2 to form the first row of matrix C.

Figure 1 shows how each matrix is divided into smaller submatrices and how they are multiplied in parallel to obtain the final result. However, it is important to note that the effectiveness of this algorithm will depend on the structure of the arrays and the number of sub-arrays into which they are partitioned.

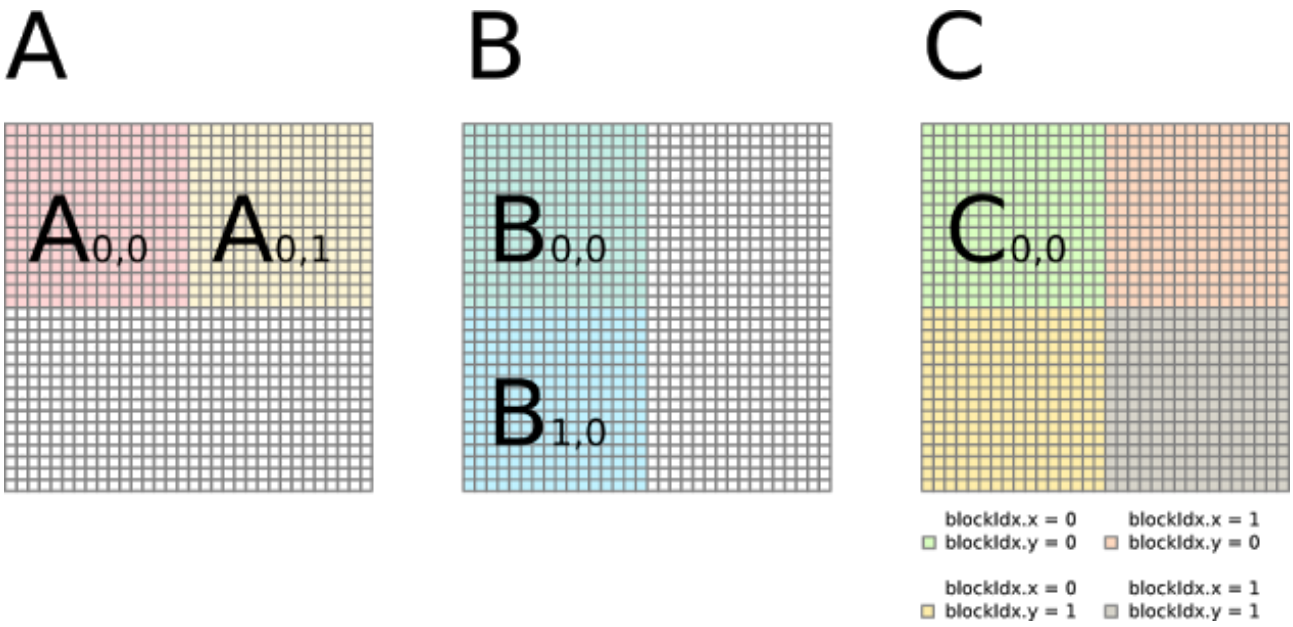

Figure 1: Tiled Matrix Multiplication Algorithm example

2.2 Distributed Programming

Distributed programming is a programming approach in which programs run on multiple networked devices or systems, with the goal of making the most of the processing and storage capacity of these devices. In a distributed system, each device or node works independently, but collaborates with other devices to complete a specific task.

Distributed programming is used in a variety of applications, including distributed database systems, distributed filing systems, parallel processing systems, distributed artificial intelligence systems, and cloud computing systems.

In general it can be said that distributed programming is a technique to increase the scalability, availability, performance and security of a system by allowing the different components of an application to run on different devices and/or networks. It is also used to break complex tasks into more manageable subtasks and distribute these subtasks across multiple devices for parallel processing.

2.3 Hadoop MapReduce framework

Hadoop MapReduce is a distributed programming framework developed by Apache that allows processing large amounts of data in parallel in a distributed environment. It is used to analyze and process large data sets in a cluster of computers.

MapReduce consists of two phases: the map phase and the reduce phase. The map phase processes each input and breaks it down into key-value pairs, while the reduce phase groups and processes these key-value pairs to produce a set of outputs. It is important to note that Hadoop MapReduce is a framework used to process large amounts of data in parallel, but it is not designed to process data in real time.

3 Methodology

In order to create an implementation that solves this research case, the creation of a java program is proposed. In order to explain this implementation in detail, we have chosen to divide it into three packages with different utilities:

- **matrix**: Package that contains all the classes in charge of the creation and management of subpartitions that will be later used in MapReduce.
- **hadoop**: Package that contains the three base classes of the framework, a Driver that will handle "jobs", a mapper and a reducer.
- **utils**: Package that contains a set of auxiliary classes for managing the persistence of the generated data and for the multiplications, facilitating the work of reducer.

3.1 Matrix package

As we mentioned before, this package contains all classes that are in charge of handling matrices during the multiplication process. Like previous treatments of matrices, we must create an object that allows us to efficiently manage the attributes that a submatrix can have, such as the values by indexes, or its size. For this reason, the *MatrixPartition* class has been created, which is an implementation of the Partition interface, which allows working with submatrices of a matrix. This class has attributes to store the row, column, and size of a subarray, as well as a two-dimensional array called *"elements"* to store the values of the elements of the subarray. It also has methods for setting and getting values of the subarray elements and a static method called *fromString* that allows you to create a new instance of the class from a String array containing the values of the subarray, that is, a from the .txt or text type input files that will contain the matrices to be multiplied.

In turn, when taking files as inputs, we will need a reader to facilitate this process. The *MatrixReader* class is a Java class whose function is to read a set of files from a specific path and store their content in a list of Strings. It uses the Java Files class to read the contents of each file and store it in the list arrays. The class also has a static method called *checkSize*, which takes as parameters the size of the array and the number of subarrays into which you want to split the array, and returns a boolean value indicating whether the size of the array is divisible by the number of submatrix. The class also uses Logger to log any errors that occur during the file reading process.

The *Partitioner* class is used to partition an array into smaller subarrays. It takes as input a path to a file containing two matrices and an integer "nSubMatrixes" representing the number of submatrices each matrix

should be divided into. It uses the *MatrixReader* class to read the matrices from the file and checks if the size of the matrices is divisible by "nSubMatrixes" using the *checkSize* method. Then, use the *SubMatrixCreator* class to split each matrix into submatrices and add them to a "subMatrixes" list. Finally, it uses the *MultiplicationManager* class to check if the submatrices can be multiplied and writes the results to disk using the *FileProccesor*.

Finally, the *SubMatrixCreator* class is used to split a matrix into smaller submatrices. It takes as input the size of the original array, a list of subarrays, and a list of rows of an original array. It uses this information to create submatrices from the rows of the original matrix.

The class has two static methods: *saveSubMatrixes* and *createSubMatrixContainer*. The first is responsible for saving the subarrays created in the subarray list and clearing the temporary subarray list. The second is responsible for creating a list of empty lists, where each inner list represents a subarray.

The *createSubMatrixes* method is in charge of creating the submatrixes. It uses a nested loop to loop through the rows of the original array and split it into subarrays. Each subarray is saved in the temporary list of subarrays. If the temporary list contains the correct number of subarrays (determined by the size of the original array divided by the number of desired subarrays), they are saved to the subarraylist and the temporary list is cleared.

3.2 Hadoop package

On the other hand, we have the hadoop mapReduce package, which contains the three key classes to be able to develop this type of algorithm, a driver, a mapper and a reducer. Next we will explain the usefulness of each one.

- ***Mapper***: The *Mapper* class is used in the Hadoop MapReduce process to process input data in a specific format. In the *map()* method, the input data in Text format is converted to a string array using the split(";") method. Next, two objects of the *MatrixPartition* class, matrix1 and matrix2, are created using the fromString() method and the data from the previously created string array.

  Finally, the private method *multiply()* is called, which receives the objects matrix1 and matrix2 as arguments, and also receives a context object. Inside the *multiply()* method, a nested loop is performed to multiply the values of the arrays and write the result to the context using the write() method. The arguments to write() are two Text objects, the first is a combination of indices i and j, while the second is the result of multiplying the values of the arrays at those indices.

- ***Reducer***: The Reducer class is a Hadoop class that is used to group and sum the values of a resulting array at a specified position. The reduce() method is overridden which uses a stream of values from the Iterable, then the mapToLong() method is used to convert each value to a long type and finally the sum() method is used to add all the stream values and obtain the total amount. The result is written to the context with the first argument being null and the second argument being a Text object, which contains the position of the array concatenated with the total sum.

- ***Driver***: In short, the Driver class is the main class that uses the *Partitioner* class to split the input array into subarrays and uses the Hadoop MapReduce framework to multiply the arrays to get the final result. In the main() method, several tasks are performed: The *manageDatamartDirectory()* method is called to remove the "submatrixes" directory if it exists. The *partitionate()* method of the *Partitioner* class is used to partition a given file array into a specified number of subarrays. A Hadoop Job object is created to configure and run the MapReduce process.Then, the Mapper and Reducer classes are set to work. Next, the key and value output classes are set. The input path is set to "submatrixes" and the output path is given by a command line argument. Finally, the job is executed and waits for its completion.

3.3 Utils package

Finally, the utils package contains the set of auxiliary classes that allow the continuous flow of the application, both regarding the management and control of submatrixes and their persistence.

The *FileProccesor* class has a static method called *writeSubMatrixesToDisk()* that is used to write a partitioned array to a file on disk. In the *writeSubMatrixesToDisk()* method, a string called createdSubmatrixes is received as a parameter, which represents a partitioned array in string format. The path is where the partitioned array will be written to disk. In this case the path is "./submatrixes/divisionTest". Then, the *createDirectories()* method of the Files class is used to create the "submatrixes" directory if it doesn't exist. Finally, the writeString() method of the Files class is used to write the partitioned array to the file at the specified path.

In short, this class has a method that is used to write a partitioned array on disk to a specific file.

Ending, the *MultiplicationManager* class has two static methods: *isMatrixComplete()* and *checkMultiplication()*. The *isMatrixComplete()* method takes as parameters the size of the submatrices and a list of lists of strings representing the submatrices. It is used to check if the array is complete, by comparing the number of elements in the first subarray with the size of the subarrays. The *checkMultiplication()* method receives as parameters the number of subarrays and a list of strings representing the subarrays. It is used to check if the submatrices can be multiplied, and if possible, returns a string containing the multiplied submatrices. A series of nested loops and stream operations are used to check whether the subarrays are the correct size to multiply and to multiply them.

4 Experimentation

To check the results, we perform the multiplication of two matrices in dense format written in a text file, and we will obtain a result in sparse form. The comparison of both results is as follows in Figure 2:

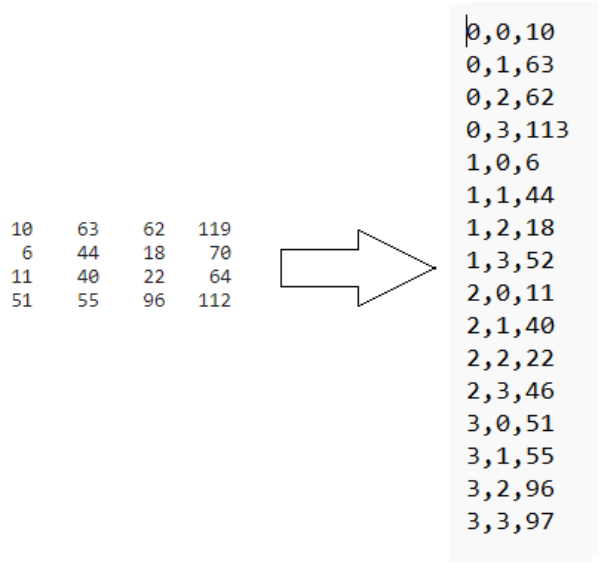

Figure 2: Expected output vs Calculated output

As we can verify, the result given in sparse format coincides with the real result of the multiplication verified from Matlab. In this way we have made sure that each result in the tests has been adequate. In the same way, the subdivisions of the matrices have been correct, and it can be checked from the "divisionTest" file.

5 Conclusion

In today's world, the treatment of large data sets has become a critical challenge for companies and organizations. Distributed programming has become an essential tool to efficiently process and analyze these large data sets. Hadoop is an open source framework that provides a distributed programming platform for storing and processing large data sets.

In this paper, a Java implementation of a tiled matrix multiplication technique in Hadoop has been presented. The implementation uses the Hadoop framework to split the input arrays into sub-arrays, process them in parallel, and get the final resulting array. The use of Hadoop allows processing large data sets quickly and efficiently, since it uses a distributed system that allows parallelization of the process.

In addition, Hadoop provides great scalability, since more hardware can be added to the cluster to process more data. It also provides high availability, as data is replicated across multiple nodes in the cluster, reducing the risk of data loss.

The implementation of this tiled matrix multiplication technique in Hadoop is important because it provides an efficient way to process large data sets. This is especially useful in applications such as image processing,

real-time data analysis, and machine learning. Distributed programming on Hadoop is a valuable tool for processing large data sets quickly and efficiently.

6 Future work

Possible future work for this Hadoop implementation would be to investigate the possibility of deploying it in a cloud environment, such as Google Cloud. This could allow the system to scale dynamically, according to processing needs, and reduce infrastructure costs by not having to maintain and manage your own cluster. In addition, Google Cloud offers advanced tools and services for distributed processing, such as Cloud Dataflow and Cloud Dataproc, which could be used to improve system efficiency and scalability.

Another possibility could be the implementation of a security system, this time in the cloud, to protect the data and guarantee their privacy and confidentiality during the process.

In summary, the implementation of this tiled matrix multiplication system in a cloud environment such as Google Cloud could offer additional advantages in terms of scalability, efficiency and security of the system.