Matrix Mutlipication with Map Reduce

Alejandro Jesús González Santana

January 2024

Contents

1 Introduction **2**

2 Problem Statement **3**

3 Methodology **4**
3.1 Project Structure 4
3.2 Compressed Matrix Approach 5
3.3 Generate Random Matrices 5
3.4 Dense Matrix Multiplication 6
3.5 Serialize and Deserialize Matrix 6
3.6 Map Reduce 8
3.6.1 Reducer Product 9
3.6.2 Mapper Product 10
3.6.3 Reducer Sum 10
3.7 Tests for Map Reduce 10
3.8 Testing the content 11

4 Experiment **13**
4.1 Results pytest 13
4.2 Results without pytest 16

5 Conclusion **17**

6 Future Work **18**

Github Repository: https://github.com/Alejeglez/FifthBenchmarkingTaskMatrixMultiplication

---

Abstract

Initiating a software project poses a considerable challenge, with resource management standing out as one of the most intricate aspects. While our primary focus typically revolves around achieving desired outcomes, this time, our aim is to maximize the utilization of available processing resources and making easy to understand the whole algorithm.

Our previous efforts involved comparing matrix multiplication across various programming languages and experimenting with diverse formats, and parallel implementations with threads and streams. This paper shifts its focus to the Hadoop framework, providing a more streamlined approach to distributed programming—an alternative to traditional methods involving threads and streams, with a more intuitive paradigm.

The results of this experiment highlight a significant reduction in execution time achievable with this specialized framework, characterized by a simpler and more concise codebase. Nevertheless, we could conclude some times hadoop and Map Reduce could not benefit us, as we are using a powerful tool for a simple problem.

---

1 Introduction

Software engineering aims to simplify tasks and maximize productivity by searching efficient solutions. To assess the performance of a program resources usage, benchmarking serves as a crucial tool for gauging resource utilization.

Matrix multiplication proves to be a practical standard for evaluating how well a programming language manages resources during execution. Yet, from our prior experience, we know that these operations can become computationally intensive, resulting in a steep rise in complexity. This increased complexity often translates into longer processing times that we could face using different matrix representations and additional resources from our computer.

However, our previous attempts at matrix multiplications were carried out in parallel, although requiring a deep understanding of resources and manual implementation nuances. While this approach may suffice for matrix multiplication, managing resources becomes more intricate in other scenarios. Hadoop, on the other hand, empowers developers to engage in distributed programming without the need for in-depth background knowledge, handling interfaces and file operations seamlessly on its own.

Hadoop efficiently utilizes CPU cores to execute tasks locally, even without the presence of a computer cluster. However, it also oversees the storage of intermediate and final results through its dedicated file system known as Hadoop Distributed File System (HDFS). While Hadoop is inherently designed for par-

allel programming in a cluster environment, it is versatile enough to be executed locally. We can conclude that map reduce(hadoop model) is widely used with approximately 1,392,223 papers, according to the scientific paper search engine Core[1].

The primary objective of this research is to assess the applicability of the MapReduce strategy in Python for matrix multiplication when compared to the sequential method. Our aim is to verify whether equivalent results can be achieved through a reduction in processing time and the adoption of a more straightforward code with the utilization of Hadoop

2 Problem Statement

Matrix multiplication, as demonstrated in our prior research, can leverage parallelism through the utilization of threads and streams to enhance execution time. This approach involves splitting matrices into submatrices and assigning them to threads for individual computation, with subsequent rotation to consolidate the complete result. However, this method necessitates a comprehensive understanding of the problem and entails a multistep process, introducing complexity.

Streams presented themselves as a viable alternative for java. Nevertheless, Hadoop emerges as a simpler option, providing a standardized interface for distributed programming without specific requirements tied to the problem at hand.

A notable advantage of Hadoop is the abstraction it affords to the programmer. In the thread-centric approach, manual allocation of threads to operations is required. In contrast, the MapReduce paradigm in Hadoop eliminates the need for explicit resource assignment, employing a divide-and-conquer strategy autonomously. Additionally, Hadoop incorporates fault tolerance, automatically retrying subtasks until a successful result is achieved in the event of failure.

Another compelling aspect of Hadoop is its streamlined parallel programming. Utilizing the MapReduce paradigm, Hadoop simplifies the process into two phases: mapping, which transforms input data into key-value pairs, applying the divide-and-conquer strategy, and reducing, where key-value pairs are grouped and a reducer function generates the global result. MapReduce operations in Hadoop require input files as arguments.

To validate the effectiveness of Hadoop, comprehensive testing is imperative. Leveraging pytest, we measure average time (ms/operation) and throughput (operations/s) through iterations and forks. This methodology aims to demonstrate accelerated execution times. Addtional test are concurrently employed to verify result consistency with sequential operation procedures.

[1] Core website: https://core.ac.uk/search?q=parallelism

3 Methodology

Machine specs:

One of the key aspects in bechmarking is the hardware used and the operating system. In this field, the specification of the machine which ran the multiplications is:

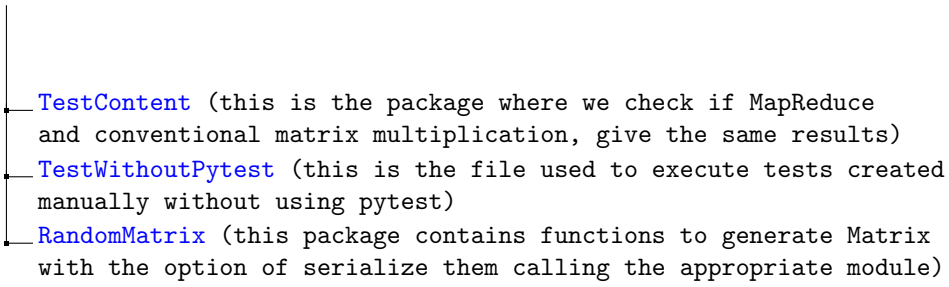
- Model: Lenovo Ideapad 330.
- CPU : Intel Core i7-8750H (6 cores, 2,2 GHz - 4,1 GHz, 9 MB).
- GPU : NVIDIA GeForce GTX 1050 4 GB.
- RAM : 16 GB 2400 MHz DDR4.
- HDD: 1 TB.
- SSD: 256 GB.
- OS: Microsoft Windows 10 Pro 64 bits.
- Logical processors: 12.

The programs were executed on an HDD disk, so the results may differ compared to running them on an SSD disk.

3.1 Project Structure

For this project we have reduced the number of necessary classes.

```
Project
├── Resources (this is the directory where we are going to store the
│   random generated matrices, in a coordinate format)
├── src (this is the folder where the packages used for the map reduce
│   and conventional matrix multiplicacion are located)
│   ├── MapReduce (this is the package where the mappers and reducers
│   │   for distributted programming, according to hadoop, are stored)
│   ├── MatrixSerializer (this package is used to store the random
│   │   generated matrices as a txt file that will be saved in Resources,
│   │   also contains a deserialize function)
│   └── MatrixOperations (this package contains the method for matrix
│       multiplication in dense matrices)
└── test (this is the folder where the packages for Map Reduce tests
    are located)
    ├── TestMapReduce (this is the package where the test of MapReduce
    │   with pytest are going to be performed)
    └── TestDenseMultiplication (this is the package where the test
        of Dense Matrix multiplication with pytest are going to be
        performed)
```



As many functions used in this project have been previously explained with code in earlier works, we will provide comments without embedding code to avoid an excessively long document with repeated content. Only the new Hadoop-focused functionality will be explained with accompanying code. The code can be reviewed in the github repository attached.

3.2 Compressed Matrix Approach

Initially, this project aimed to include additional modules. The plan was to implement the Coordinate Matrix format along with its corresponding builder and class, as well as a coordinate class. However, during testing, accessing object attributes proved to be time-consuming, especially with a matrix size of 32 rows and columns. The multiplication for this matrix size, when combined with dense matrices, took nearly 81 seconds. Furthermore, here is a list of recorded times for matrix sizes ranging from 2x2 to 128x128 using the map reduce implementation:

- 2: 0.8109130859375 seconds
- 8: 1.7113358974456787 seconds
- 16: 2.316984176635742 seconds
- 32: 3.2507314682006836 seconds
- 64: 8.126513957977295 seconds
- 128: 43.08181643486023 seconds

The recent timing data suggests a potential issue linked to the object-oriented programming approach. To investigate this hypothesis, a simple dense matrix multiplication test was performed with a matrix size of 128x128, yielding a time of 0.29346251487731934 seconds. In response to this observation, the decision was made to move away from the use of objects that were employed in the Java implementations for the Coordinate Matrix. As MapReduce necessitates an input file, the matrix will be serialized from the dense format.

3.3 Generate Random Matrices

We essentially have a function for generating new random matrices. In this context, we need to specify not only the size of the desired matrix but also an additional parameter, 'id' which is typically set to "A" or "B." Additionally, there is a parameter to indicate whether we want to serialize the matrix. If the

serialize parameter is set to True, the function will invoke the serialize method of a MatrixSerializer object. The dense matrix will be populated with random values ranging from 0 to 10. When the serialize parameter is set to True, we invoke the serialize method with the dense matrix, the specified id, and the size provided to generate the random matrix.

If we run this file directly, matrices ranging from size 2 up to size 512 will be generated using the generate_random_matrices function with powers of two. This function essentially creates two matrices of the same size but with different ids, and the serialize parameter is set to True. Consequently, we can store these matrices in the Resources folder, located in the root folder of the project. The main goal is to have matrices to test with pytest.

3.4 Dense Matrix Multiplication

The package MatrixOperations contains a class which the same name, and only one method, called multiply_matrices. We will basically create a matrix result as a list of list where each sublist will respresent a row in the result. We will employ the standard matrix multiplication algorithm. In this approach, the value at position (0, 1) in the result matrix is determined by multiplying each element of the first row with the corresponding element in the second column of the other matrix and sum all the products. This can be generalized for every position

We will return the result matrix. The main goal of this function is to use it to have also a reference for the classic dense matrix multiplication to compare them with the hadoop focus in the experiments.

3.5 Serialize and Deserialize Matrix

This is a crucial component that has been adapted for use with our MapReduce focus. The MatrixSerializer class encompasses the serialize and deserialize methods, which differ slightly from previous implementations due to the absence of the coordinate matrix format. The serialize method accepts a Python list of lists, while the deserialize method generates two lists of lists. The reason for deserialize returning two matrices will become clear in the following subsection.

The serialize method first checks if the "Resources" folder exists in the root directory of the project. If the folder doesn't exist, it creates one. If the provided ID in the method is "A," it signifies the creation of a new file. Consequently, it removes the current matrix of the specified size to allow testing and easy iteration through files by changing the size. The "A" denotes the first matrix to be stored in the text output file.

The major deviation from the previous serialized text is that the file will now store two matrices. The idea is to pass only one argument to the MapReduce function and incorporate the matrix's ID ("A" or "B") as the first element in each line. If the ID is "B," there is no need to remove the current file; instead,

the new matrix is appended to the end of the file. The file's line will look like this:

id (points to the matrix) row column value

The iteration through a dense matrix during this serialization will result in code that looks like this:

```
class MatrixSerializer():
    def serialize(self, matrix, id, size):
        resources_path = os.path.join(os.path.dirname(__file__), '..', 'Resources')
        if not os.path.exists(resources_path):
            os.mkdir(resources_path)


        output_file_path = os.path.join(resources_path, f"matrix_{size}.txt")

        if os.path.exists(output_file_path) and id == "A":
            os.remove(output_file_path)


        with open(output_file_path, 'a') as writer:
            for i in range(len(matrix)):
                for j in range(len(matrix)):
                    if matrix[i][j] == 0:
                        continue
                    else:
                        line = f"{id} {i} {j} {matrix[i][j]}"
                        writer.write(line + '\n')
```

Listing 1: Serialize method

On the flip side, during deserialization, two lists of lists are created, initially filled with zeros for each list. The number of sublists matches the size in each list, and the same number of lists is present.

After initializing the result matrices, the method reads the lines of the files passed as an argument. Upon splitting the current line, it checks whether the first element is equal to "A" or "B," indicating the matrix it belongs to. Once the matrix is identified, the method fills the matrix's position correctly using the second element (row) and the third element (column), assigning the last element (value).

After processing all the lines, the method returns the compiled matrix. The code is as follows:

```
class MatrixSerializer():
    def deserialize(self, path, size):
        matrix_a = []
        matrix_b = []
        for i in range(size):
            matrix_a.append([])
            matrix_b.append([])
            for j in range(size):
```

```
                    matrix_a[i].append(0)
                    matrix_b[i].append(0)
        with open(path, 'r') as reader:
            lines = reader.readlines()

            for line in lines:
                line = line.split()
                if line[0] == "A":
                    matrix_a[int(line[1])][int(line[2])] = int(line
    [3])
                elif line[0] == "B":
                    matrix_b[int(line[1])][int(line[2])] = int(line
    [3])
        return matrix_a, matrix_b
```

Listing 2: Deserialize method

3.6 Map Reduce

As we have seen previously, Hadoop essentially revolves around two operations: Map and Reduce. This framework can be implemented using the MRJob package. The class MatrixMultiply must inherit from MRJob. However, we can employ as many mappers and reducers as needed; in our case, we will utilize two mappers and two reducers. These processes are referred to as "steps" in the Hadoop environment. Each step can be defined using MRStep. In our scenario, we will use two steps, where each step is associated with a mapper and a reducer. We will define the specifics of each step later, but first, let's explore how they are structured:

```
def steps(self):
    return [
        MRStep(mapper=self.mapperRead,
               reducer=self.reducerProduct),
        MRStep(mapper=self.mapperProduct,
               reducer=self.reducerSum)]
```

Listing 3: Defining the steps

Mapper Read

The command to execute a MapReduce task is as follows: python src/MapReduce.py input_file.txt.

In the initial mapper, which processes the input file provided as an argument to invoke the Python MapReduce program, each line is split into a list of elements. The first element in this list serves as the matrix key, determining the arrangement of key-value pairs. In matrix multiplication, the rows of the first matrix are multiplied by the columns of the second matrix.

If the matrix is A (the first matrix), the key will represent the column, excluding it from the tuple of values. This tuple includes the matrix key, the row, and the value. Conversely, if the matrix is B, the key will represent the row, and

the value will be a tuple comprising the key, column, and value. This organization ensures that the results of matrix multiplication are grouped appropriately in the reducer.

The code will be the following one:

```
    def mapperRead(self, _, line):
        elements = line.split(' ')
        if len(elements) == 4:
            key, row, column, value = line.split(' ')
            key = str(key)
            row = int(row)
            column = int(column)
            value = int(value)

            if key == "A":
                yield column,(key,row,value)
            elif key == "B":
                 yield row,(key,column,value)
```

Listing 4: Mapper Read

3.6.1 Reducer Product

The reducer receives a key and a tuple of values from the mapper. The tuple of values is converted to a list, and two empty lists are initialized to store the values for each matrix. Depending on the first element of the incoming value (the matrix key), the transformed tuple list is stored in the list for Matrix A or Matrix B, excluding the key of the matrix in this list. Then, an empty result list is created. We iterate first through the list of the first matrix, and within it, iterate through the list of values for the second matrix. Each element of the result list is a tuple composed of a coordinate tuple and the product of the corresponding elements. This list contains the partial result for each cell.

Once we have gathered all the partial products in the result list, we emit them to the next mapper. The key becomes the coordinate, and the value becomes the value of the partial product. The code is as follows:

```
    def reducerProduct(self, key, values):
        values_list = list(values)

        values_matrix_a = []
        values_matrix_b = []

        for value in values_list:
            if value[0] == "A":
                values_matrix_a.append(value[1:])
            elif value[0] == "B":
                values_matrix_b.append(value[1:])

        result = []
        for value_a in values_matrix_a:
            for value_b in values_matrix_b:
                coordinate = (value_a[0], value_b[0])
```

```
                product = value_a[1] * value_b[1]
                result.append((coordinate, product))
        for coordinate, value in result:
            yield coordinate, value
```

Listing 5: Reducer product

3.6.2 Mapper Product

This mapper essentially organizes the results by coordinates, grouping together the outcomes of each partial multiplication associated with a specific coordinate. The purpose is to forward these grouped values to the subsequent reducer, initiating the second step where all the values for a cell or coordinate are consolidated. The code is quite simple:

```
    def mapperProduct(self, coordinates, products):
        yield coordinates, products
```

Listing 6: Mapper Product

3.6.3 Reducer Sum

In this final step, the reducer receives the grouped values associated with a specific coordinate. It calculates the sum of the partial products for that coordinate, effectively assembling the final result. This pivotal stage marks the conclusion of the MapReduce process, yielding the ultimate outcome and completing the map reduce.

The code is the following one:

```
    def reducerSum(self, coordinates, products):
        yield coordinates, sum(products)
```

Listing 7: Reducer Sum

To be able to execute this Map Reducer we need to execute the method run of MatrixMultiply so we need to add the following lines at the end of the file:

```
    if __name__ == '__main__':
        MatrixMultiply.run()
```

Listing 8: Main

The result of MapReduce will be stored in a file storage by the hadoop file system.

3.7 Tests for Map Reduce

To ensure consistent metrics for matrix multiplication using MapReduce, we have employed pytest, a framework previously utilized in our projects. pytest facilitates the collection of metrics, including average time and throughput mode.

A crucial aspect is that, as observed, MapReduce can only be invoked from the command line. Consequently, we utilize the subprocess package and the run method with the command python src/MapReduce.py input_file. For benchmarking, we create a benchmark group where we specify warm-up iterations and the maximum time per iteration, set to one second in our case. All files need to be passed as parameters. We conduct 5 iterations and 4 rounds, with each round comprising 5 iterations, resulting in a total of 20 iterations.

There will be another similar set of tests to store benchmarks for conventional dense matrix multiplication. However, it is important to note that this test shares the same attributes as the MapReduce benchmark, and we have already reviewed it in previous works.

The code for the MapReduce test is as follows:

```
import subprocess
import pytest
import sys
import os
from glob import glob

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(
    __file__), '..')))

def run_map_reduce(input_file):
    command = ["python", "src/MapReduce.py", input_file, "--output"
    , "output/{input_file}"]
    process = subprocess.run(command, shell=True)
    assert process.returncode == 0

resource_files = glob(os.path.abspath(os.path.join(os.path.dirname(
    __file__), '..', 'Resources', 'matrix_*.txt')))


@pytest.mark.benchmark(group="throughput", warmup=True,
    warmup_iterations=5, max_time=1)
@pytest.mark.parametrize("output_file", [(resource_file)  for
    resource_file in resource_files])
def test_matrix_multiplication_with_setup(benchmark, output_file):
    benchmark.pedantic(run_map_reduce, args=(output_file,),
    iterations=5, rounds=4)
```

Listing 9: Test for Map Reduce

3.8 Testing the content

MapReduce saves its output in a file stored within a directory utilized by Python. This introduces challenges when attempting to leverage frameworks like unittest to verify result consistency. To address this issue, we have devised a custom testing file—TestContent.py. Running this file allows us to test the MapReduce output effortlessly.

The initial step involves outlining the MapReduce procedure and redirecting

the output to a file located within our project. We also nullify the additional information generated by MrJob to streamline the process.

For each matrix in our resources folder, we execute both dense matrix multiplication (using deserialization beforehand) and MapReduce multiplication. The output of the latter is stored in a folder, divided into different parts/files corresponding to the elements in the matrix. Consequently, we traverse through all the files in the folder to construct a matrix that we can iterate over, containing the resulting values. This matrix is crucial for comparison with the matrix obtained from dense multiplication.

Upon obtaining both matrices, we systematically check each element's equality by iterating through the size. If discrepancies are found, a message is displayed indicating the problematic position. Successful completion of the test triggers a confirmation message. The corresponding code is as follows:

```
for resource_file in resource_files:

    size = int(resource_file.split('_')[1].split('.')[0])
    print("Running test for size {}".format(size))
    run_map_reduce(resource_file, size)
    matrix_a, matrix_b = MatrixSerializer.deserialize(resource_file
    , size)
    matrix_result = MatrixOperations.multiply_matrices(matrix_a,
    matrix_b)
    hadoop_output_dir = "output/{}".format(size)

    matrix_result_hadoop = []
    for i in range(size):
        matrix_result_hadoop.append([])
        for j in range(size):
            matrix_result_hadoop[i].append(0)

    for filename in os.listdir(hadoop_output_dir):
        hadoop_result_file = os.path.join(hadoop_output_dir,
    filename)

        if os.path.isfile(hadoop_result_file):
            with open(hadoop_result_file, 'r') as reader:
                for line in reader.readlines():
                    line = line.split()
                    matrix_result_hadoop[int(line[0][1:-1])][int(
    line[1][:-1])] = int(line[2])

    all_comparisons_pass = all(matrix_result[i][j] ==
    matrix_result_hadoop[i][j] for i in range(size) for j in range(
    size))

    if all_comparisons_pass:
        print(f"The comparison for size {size} passed successfully"
    )
    else:
        for i in range(size):
            for j in range(size):
                assert matrix_result[i][j] == matrix_result_hadoop[
```

```
    i][j], f"Error in {resource_file} in position {i} {j}"
```

Listing 10: Test for content consistency

The execution of this file has verified that both MapReduce and Dense Matrix Multiplication yield identical results.

4 Experiment

4.1 Results pytest

Through our testing, we observed that the Hadoop implementation consumes more time than the sequential one, even though it produces the same result. While all the previously mentioned advantages of Hadoop remain true, scalability becomes a significant challenge for our problem. The benefits of matrix multiplication lie in the ability to handle files of any size. However, our computers face limitations when dealing with large volumes of data, where Hadoop proves to be more efficient. When referring to high data volumes, we are talking about gigabytes or even terabytes. Consequently, we acknowledge that the tests may exhibit bias in favour of conventional matrix multiplication due to logical constraints. These tests might yield more positive results for Hadoop and MapReduce with larger matrix sizes. Unfortunately, such a comparison is not feasible with our current resources, given the considerable time it already takes in a sequential Python approach.

The results obtained in this experiment should not be viewed as fully representative of the intended use cases for Hadoop and MapReduce due to our constrained computing environment. Additionally, the experimentation was limited to a select range of matrix sizes, acknowledging the notable increase in execution time for Hadoop implementation. The matrices evaluated in this study span from 2x2 to 64x64.

The recorded experiment results adhere to the following table structure:

Size	Mean Execution Time(us)	Operations per second
2	0.055620	17979.144200
4	0.071760	13935.340000
8	0.184040	5433.601400
16	0.836970	1196.072100
32	6.123375	163.308600
64	54.280760	18.422700

We've gathered the mean execution time in milliseconds for each matrix size, along with the corresponding operations per second—calculated as the inverse of the mean time in seconds.

The pytest results, however, appear somewhat perplexing:

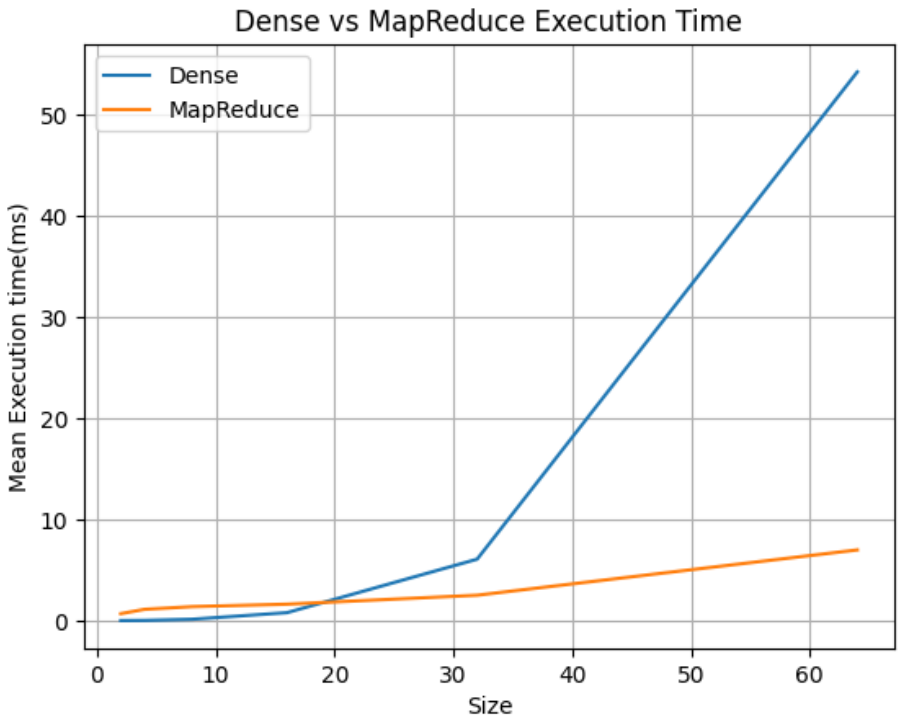


Figure 1: Pytest execution time until 64 matrix size using pytest

The graph 1 indicates that MapReduce outperforms sequential multiplication, suggesting a faster execution. However, it's important to note that we observed earlier issues with MapReduce when validating its functionality. There is a possibility that pytest may not be executing the matrix multiplication with MapReduce correctly, or it might be measuring the execution time of each subproblem. Another possibility is that pytest does not work well with subprocess. The exact cause is challenging to pinpoint, given the limited feedback provided by pytest. Nevertheless, considering the operations per second provides more meaningful insights:

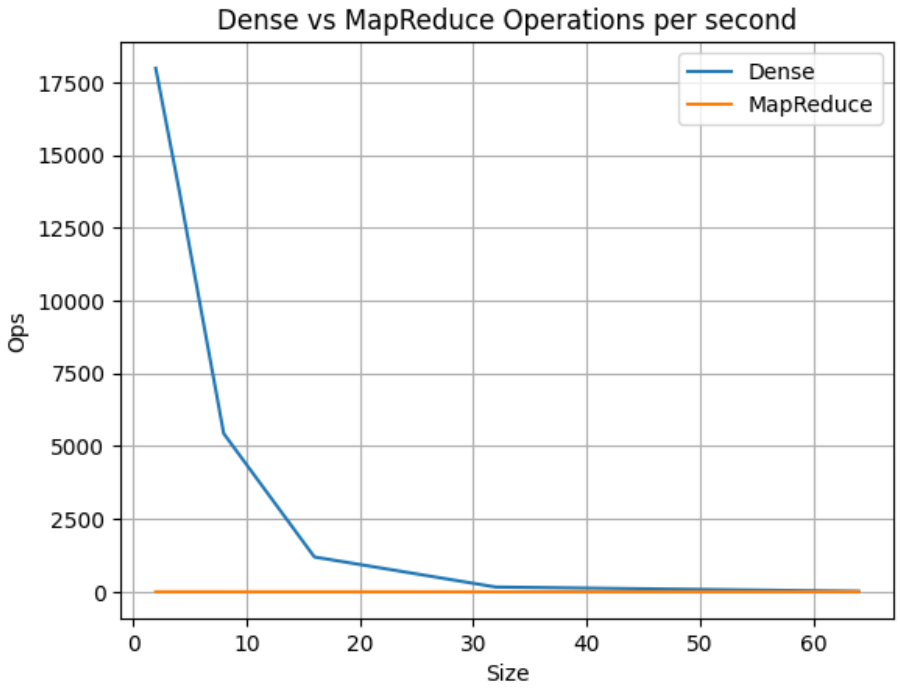


Figure 2: Pytest operations per second until 64 matrix size using pytest

The graph depicted in Figure 2, illustrating operations per second, reveals a notable contrast between MapReduce and sequential multiplication. The MapReduce implementation maintains a relatively stable value near 0, whereas the sequential approach exceeds 17500 operations per second. This disparity underscores the divergent strategies employed by each method. Sequential multiplication achieves a higher operation rate since it leverages a single thread. In contrast, MapReduce aims to reduce the number of operations by dividing the global task and distributing it across multiple processor units. However, it is crucial to note that, as previously mentioned, the results presented by pytest may not accurately reflect the actual performance.

4.2 Results without pytest

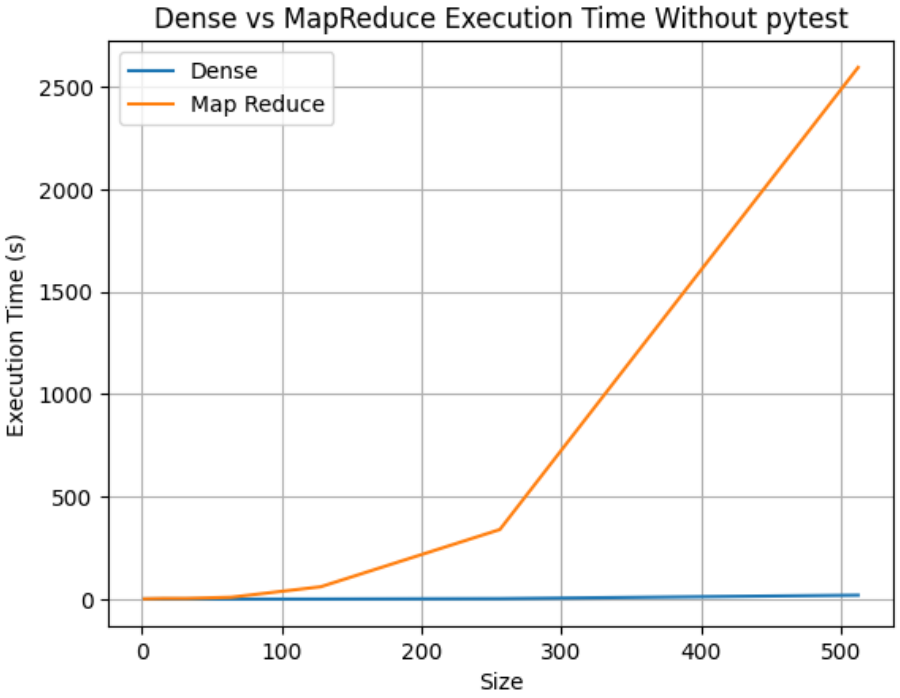


Figure 3: Execution time until 512 matrix size without using pytest

In this analysis, the marked contrast in execution times between Dense Matrix Multiplication and the Map Reduce implementation is evident in Figure 3. Notably, as the matrix size reaches 64, the disparity in performance becomes particularly pronounced. While differences exist for smaller matrix sizes, they are not as substantial.

The crux of the matter becomes apparent when examining the case of a matrix size of 512 rows and columns. The Map Reduce implementation takes approximately 2594 seconds (or nearly 43 minutes) to complete, whereas Dense Matrix Multiplication accomplishes the task in just 19 seconds. This stark contrast reinforces our observation that Map Reduce and Hadoop face challenges when dealing with matrix files that are not sufficiently large. Even in the case of the larger matrix (512 rows x 512 columns), which occupies a mere 5MB of storage, the performance of Map Reduce is notably suboptimal.

While it is important to consider the possibility of suboptimal Map Reduce configuration, it is evident that the advantages of using Map Reduce are more

pronounced with substantially larger file sizes.

To gauge the speedup, let's examine the indicator showcasing how much faster Dense Sequential Matrix Multiplication is compared to Map Reduce with one iteration across all sizes:

	Mean of execution time in seconds
Dense	2.40969
MapReduce	334.858474

The analysis reveals a substantial contrast, with Map Reduce exhibiting a significant mean execution time of 334 seconds (approximately 5 minutes), compared to the mere 2.4 seconds for the sequential and conventional mode. This results in a speedup indicator of 138.96331, signifying that Dense Sequential Matrix Multiplication is 138 times faster than Map Reduce. These findings corroborate our earlier observations regarding the comparatively lower performance of Map Reduce in the context of our study.

5 Conclusion

MapReduce is a powerful tool that enables programmers to introduce parallelism into their tasks without requiring an in-depth understanding of parallel processing or additional operations to divide the workload. It employs a straightforward interface comprising a mapper to group values by key and a reducer to operate on the values sent by the mapper.

While MapReduce can be highly effective for large file sizes, especially in the gigabyte range, its inherent tasks, such as working with its own file system and distributing workloads, add some overhead to the overall task execution time. This overhead becomes particularly noticeable when dealing with smaller file sizes, such as those used in this experiment (none exceeding a gigabyte). The file operations and task distribution implemented by MapReduce become more advantageous when working with much larger files, ranging from gigabytes to terabytes.

However, when testing with matrix multiplication tasks and comparing them to dense multiplication, challenges arise as Python's sequential mode already exhibits slower performance beyond matrix sizes of 1024 or 512. Consequently, the testing focused on files containing matrices up to size 64, which does not fully showcase the potential of MapReduce. Despite producing consistent results with sequential operations, individual executions of MapReduce matrix multiplications appear considerably slower than their sequential counterparts. This slowdown could be attributed to the additional operations implemented by Hadoop to achieve parallelism, which, at a lower level, may overly complicate the problem.

In summary, MapReduce serves as an alternative means to implement parallelism seamlessly, eliminating the need for manual thread management through

a simple interface and automated mechanisms. While it significantly streamlines the processing of massive file sizes, it may introduce unnecessary overhead for smaller files. In the context of matrix multiplication, the sequential mode proves to be more efficient for the range of sizes considered in this study.

6 Future Work

In the future, it would be advantageous to explore the capabilities of MapReduce with substantial datasets. However, executing such tests on our personal computer, as discussed in the document, is not a feasible option. Viable alternatives include leveraging web services like Amazon EMR (Elastic MapReduce), utilizing containerized environments, or orchestrating a cluster of computers. Additionally, exploring the implementation of this strategy in Java, where we observed faster execution times in previous studies, is worth considering. It's important to note that this approach necessitates a separate installation and download of Hadoop components.

Furthermore, there is room for enhancing the testing methodology. The current pytest-based tests did not yield fully representative results, suggesting opportunities for refining the testing framework, incorporating additional metrics, and exploring alternative testing environments and hadoop implementations to ensure a more comprehensive evaluation of MapReduce's potential.