Universidad de Las Palmas de Gran Canaria
Grado en Ciencia e Ingenier´ia de Datos

Big Data

Distributed Matrix Multiplication

Zuzanna Furtak

Github link

Contents

1 Abstract

In this report I have compared the parallel method of multiplying matrices with the new - distributed - approach. To achieve distributed computation, I used the blocking method of matrix multiplication and Hazelcast to send specific blocks to different machines in a cluster. The results were surprising as the time results were worse than normal parallel computations. This is probably due to the long data transfers between nodes in the cluster. Since large parts of matrices have to be sent and the benchmark does not distinguish between multiplication time and transfer time, the final time is really high. Finally, I also tried the distributed approach in Python to check its possibilities in this area, but the resulting execution times were so large that I could not even compare the same sizes with Java code. As a continuation of this experiment, I am thinking about finding an alternative way to send data between nodes, hopefully more optimized, as well as different optimizations that can be adjusted for specific matrices.

2 Problem statement

So far, I have managed to benchmark multiplication matrices against several programming languages: Python, C and Java. Since Java seemed to be the best, the following benchmark was done in Java and I compared different ways to optimize these calculations. For example, blocking or column blocking methods. I also analyzed sparse matrices using a special implementation and multiplication method. The next step was to take advantage of the computer's cores and run the matrix multiplication on many threads. The natural next step was to distribute the multiplication, not across threads, but across different machines. In order not to be limited by the hardware, I introduced matrix multiplication distributed over many nodes to speed up the whole process. What I wanted to know was whether an extra machine could make a huge improvement, and whether the improvement in computation would make up for the time taken to send the data around the cluster.

3 Methodology

To measure the performance of each multiplication method, I used a Macbook Pro with an Apple M3 Pro chip and 18GB of RAM. As a second computer, I used my friend's Lenovo with an AMD Ryzen 4800U and 16GB of RAM. The matrix sizes used to evaluate the performance of the algorithms range from 500x500 to 2000x2000. As a measurement tool, I use the Java Microbenchmark Harness (JHM), which allows me to obtain accurate results by using 5 warm-up iterations followed by 10 rounds of 10 iterations each. This allows me to obtain appropriately averaged final results. I also implemented the distributed approach in Python to compare the distributional impact in these 2 languages. Here I didn't use the benchmarking framework, I just counted the time between the start and end of the executed program. All execution times are in milliseconds.

4 System architecture

The program is designed to benchmark a distributed matrix multiplication algorithm using Hazelcast, a framework for distributed computing across nodes. At its core is the *MultiplicationBenchmark* class, which uses JMH to set benchmark parameters and configures Hazelcast to efficiently distribute tasks. Supporting classes include *MatrixMultiplicationTask*, which represents individual unit of work, and *MultiplicationTaskDistributor*, which handles the division of matrix multiplication into smaller tasks and their distribution across nodes in the cluster. Shared data, such as matrices and intermediate results, is managed by the singleton *MultiplicationMatrixDataManager*, while the *MultiplicationTaskReceiver* handles tasks - multiplication calculating - on each node. The Hazelcast XML configuration file defines the cluster setup and serialization settings.

5 Experiments

6 Java

6.1 Basic Matrix Multiplication

6.1.1 Introduction

I recalled the basic matrix multiplication algorithm to obtain better comparison for optimized - parallel methods.

6.1.2 Execution time

As is showed below in the *Table 1* the algorithm is getting significantly slower for bigger matrices. Therefore it would be hard to use this method for realistic problems as the matrices are much bigger than 2000x2000.

Size	Min time	Max time	Mean
500	60.293	84.148	61.828
1000	934.281	1000.342	949.800
1500	3544.187	3636.462	3572.708
2000	8908.702	9714.008	9261.023

Table 1: Benchmarking results for basic matrix multiplication

6.1.3 Chart

As can be noticed on the chart, execution time is growing rapidly with bigger matrix sizes.

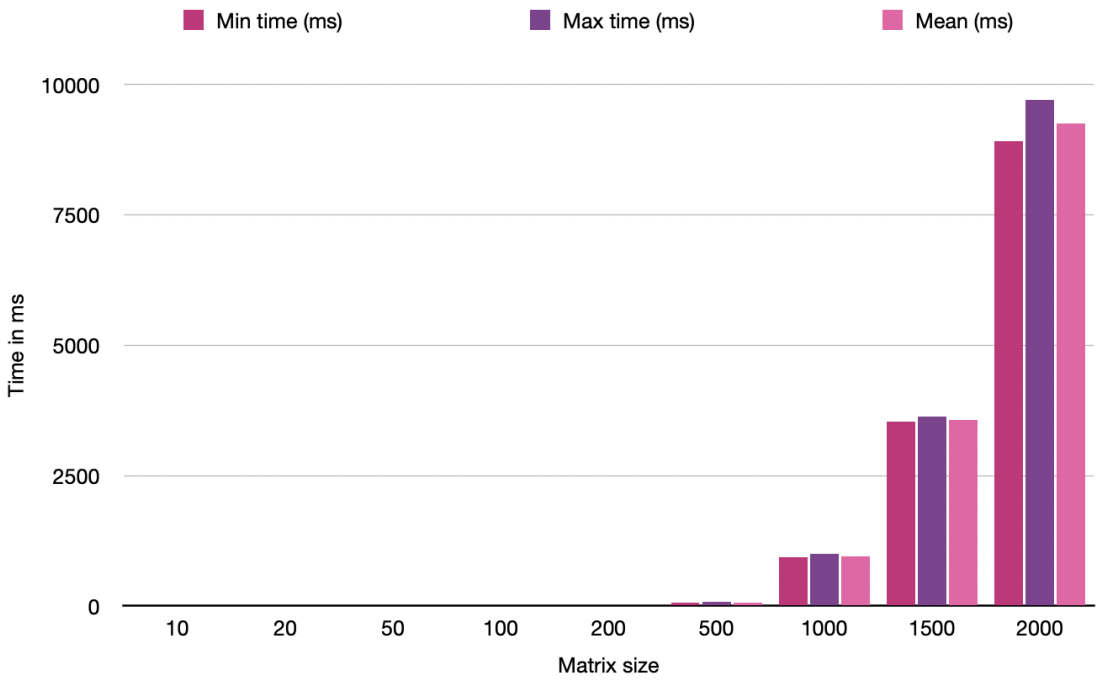

Figure 1: Visualized results for basic matrix multiplication

6.2 Parallel Blocking

6.2.1 Introduction

Blocking is a technique used to optimise matrix multiplication by dividing the matrices into smaller sub-matrices or blocks, which I discussed in the second report. This approach is ideal for

parallel and distributed computing because it allows us to process multiple blocks simultaneously, either by many threads or many machines, which speeds up the computation significantly. To achieve parallelism, I used *ExecutorService* from Java with all available cores, which is 11. Here, just to recall the results from the previous report, I am trying to beat diving computations for more machines.

6.2.2 Execution time

As we can see, the results, especially for larger matrices, are much better than the basic algorithm. Even for larger matrices (1000-2000) the results are very satisfactory.

Size	Block size	Min time	Max time	Mean time
500	256	13.222	21.496	13.806
1000	1024	46.268	62.849	48.432
1500	1024	349.700	362.807	354.541
2000	1024	940.573	1157.628	1036.218

Table 2: Benchmarking results for parallel blocking matrix multiplication

6.2.3 Chart

As we can see, the results increase significantly for larger matrices, but the time is much shorter than for a basic algorithm.

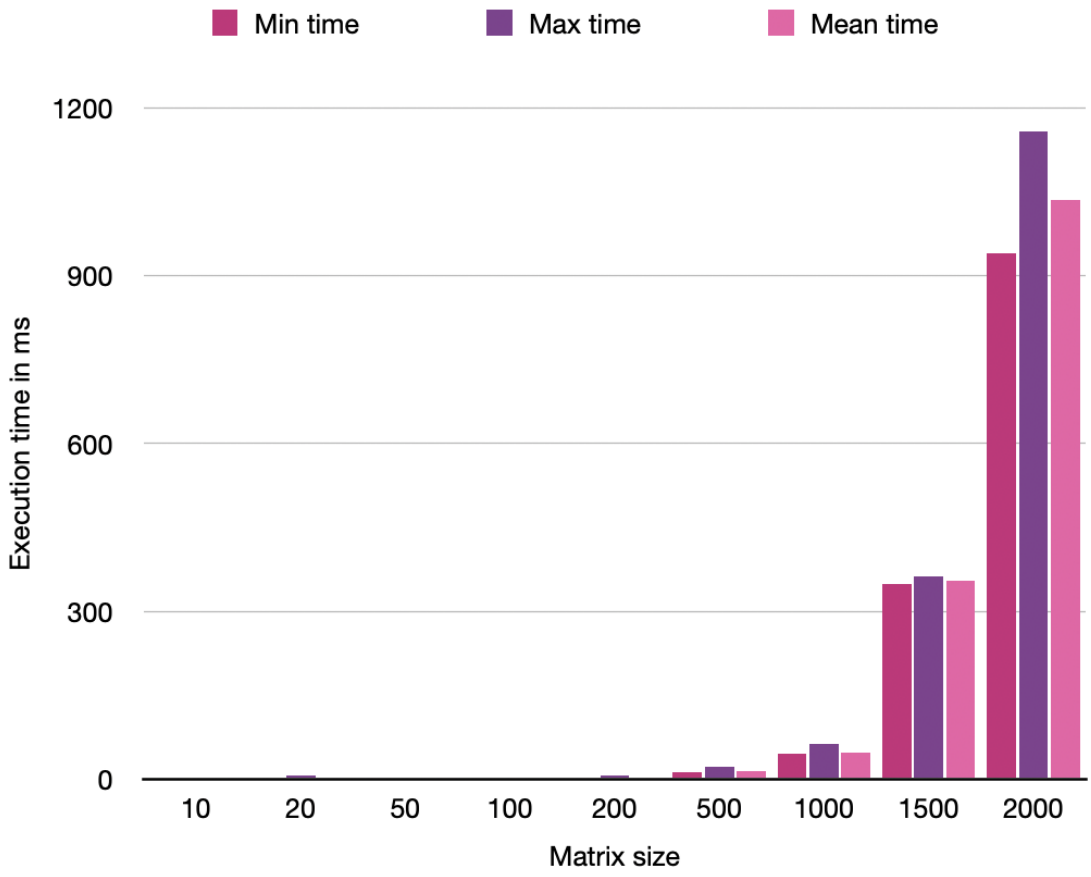

Figure 2: Visualized results for parallel blocking matrix multiplication

6.3 Distributed multiplication

6.3.1 Introduction

As I mentioned before, locking is a great way to distribute computations because you can split the blocks and send them to different nodes on a network. To see how much of a boost this gives, I tried the same algorithm I used for parallel computing before. The main difference this time was that I didn't just split the work between threads - I split it into tasks that were sent to different nodes on the network.

6.3.2 Execution time

As you can see in the table below, the results are worse than the parallel ones. The problem with the benchmark is that the time includes multiplication and sending data between nodes. As we have to send the whole matrices, the time increases significantly with the size of the matrix. However, the results are surprising and it was not obvious to me at first that an improvement

could slow down the performance as there are different factors to consider. I can imagine that the results might be better if we used an optimized way of sending the data or maybe divided the task differently.

Size	Block size	Min time	Max time	Mean time
500	256	840.958	1459.618	1021.152
1000	1024	4739.564	6442.451	5197.582
1500	1024	21005.074	22582.133	21864.068
2000	1024	50398.759	53955.527	52720.724

Table 3: Benchmarking results for distributed matrix multiplication

6.3.3 Chart

As you can see from the graph, the execution time increases dramatically as the size increases. In theory, it should scale better as the number of nodes in the cluster increases. However, as mentioned above, most of the time is spent transferring data between nodes.

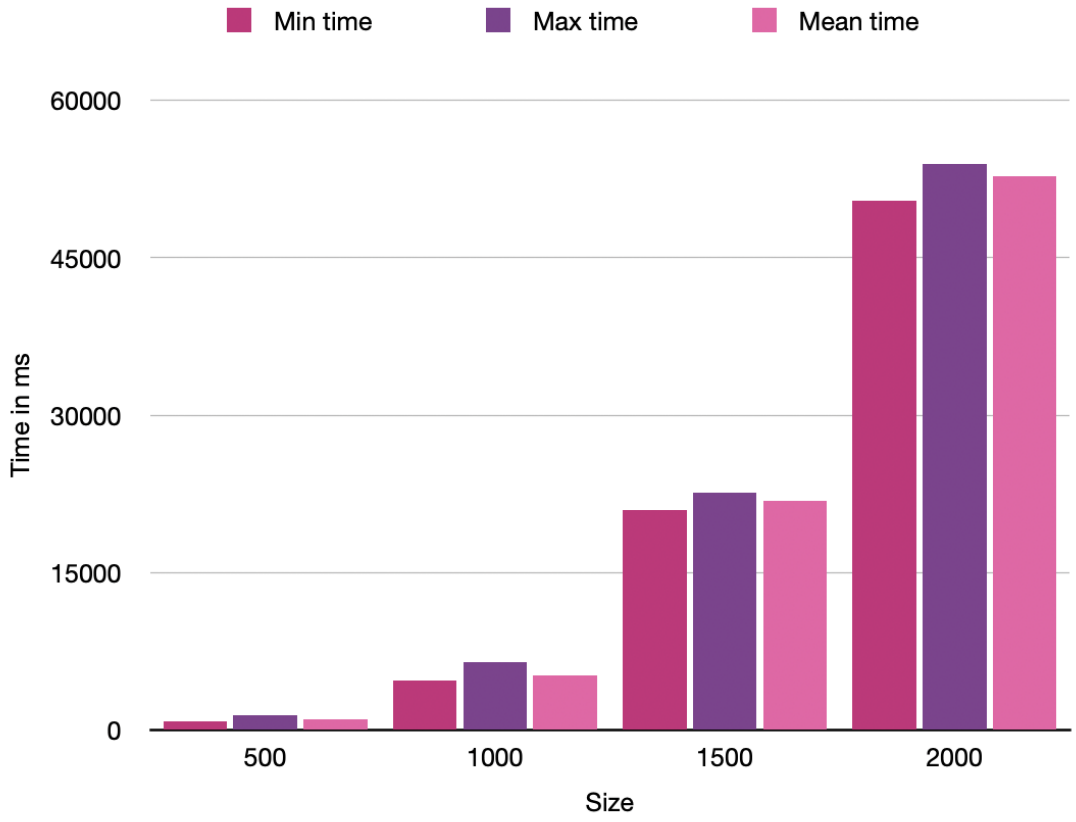

Figure 3: Visualized results for distributed matrix multiplication in Java

6.4 Python

6.4.1 Introduction

This Python implementation uses Hazelcast to distribute the work of multiplying large matrices by splitting them into smaller blocks, similar to Java code. It first divides the matrices into smaller chunks and stores them in Hazelcast's map. It then distributes the blocks and multiplies them by fetching the corresponding pieces from the distributed maps. After the calculation, it collects the results from Hazelcast and combines them into the final result matrix. The program uses a custom benchmark because it was the most convenient way to count time.

6.4.2 Execution time

As we can see from the table, the results are much worse than the Java results. There are several reasons for this. Firstly, Python is a much slower programming language than Java, and it doesn't truly support parallel computations, which is a significant bottleneck in the computations. Because of the much longer execution times, I was forced to test smaller matrices.

Size	Min time	Max time	Mean time
50	75.876	90.827	82.145
100	651.786	699.654	667.987
200	6018.984	6709.926	6378.273
300	18027.793	20229.762	19293.479

Table 4: Distributed Matrix Multiplication: Expected Results in Python (Single-Threaded Multiplication)

6.4.3 Chart

As we can see on the chart, even for little sizes of the matrices, the time increases sharply.

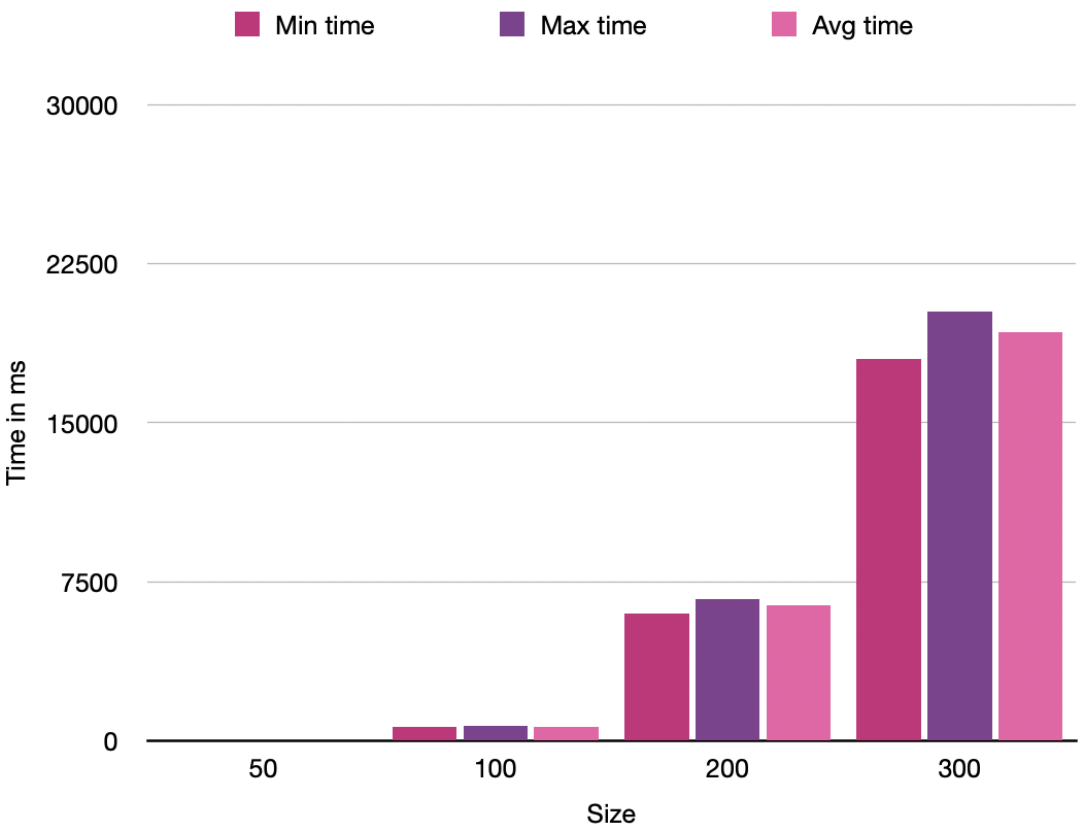

Figure 4: Visualized results for distributed matrix multiplication in python

7 Conclusions

As you can see from the last graph, the difference is significant. This clearly indicates that I should either introduce a more optimised method of transferring data between machines or differentiate this process from benchmarking.

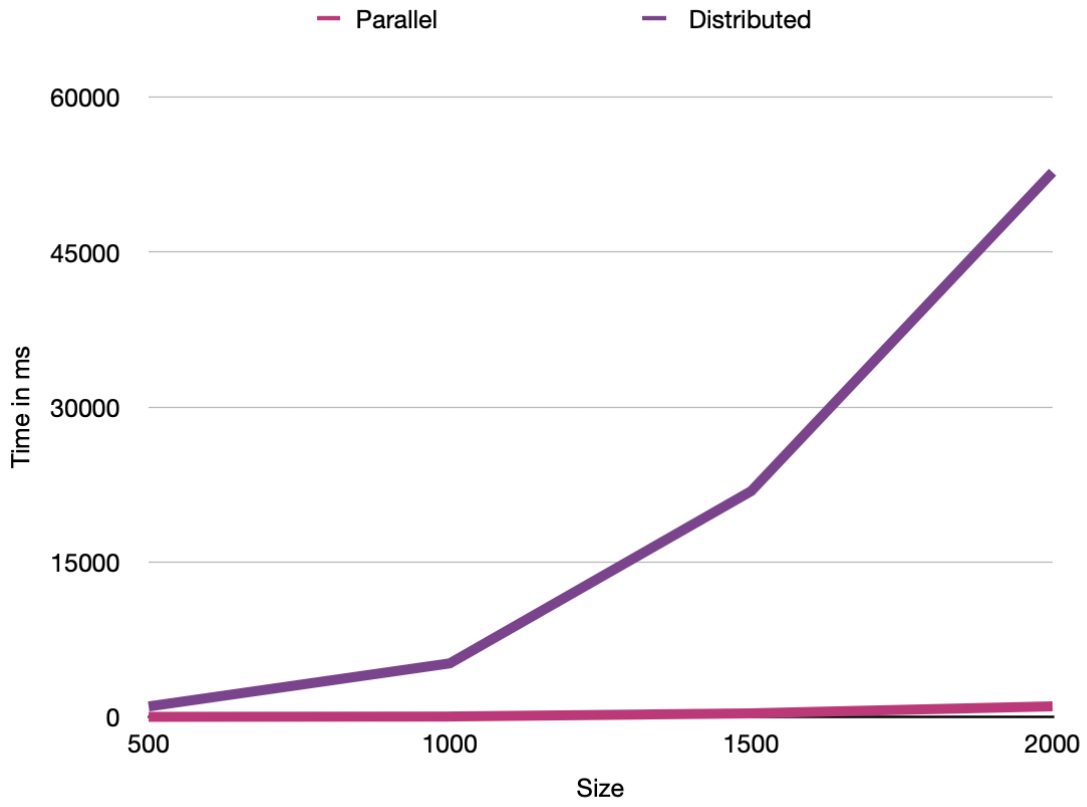

Figure 5: Visualized comparison of parallel and distributed matrix multiplication

8 Future work

Having experimented with different programming languages and optimization methods, as well as parallelization and distributed approaches in this report, I have the feeling that the road is coming to an end. What can still be done, however, is to experiment with different algorithms adapted to specific matrices. What can make a huge difference is increasing the computing power by adding new, better machines to get the most satisfactory results. Most importantly, an optimized data transfer method should be introduced, as it is currently impossible to compute very large matrices because the sending process takes too long. Since I'm looking at a fairly wide range of possibilities here, there's still a lot to explore.