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

Big Data

Optimized Matrix Multiplication

Zuzanna Furtak

Github link

Contents

1 Abstract

In this report I have compared 2 parallel methods of multiplying matrices with the basic algorithm as well as with vectorized method. To get parallel multiplication I used executors and parallel method from *Arrays* library. Of all the methods measured, the best results were obtained using the Parallel Blocking algorithm, especially for larger matrices (1000 - 2000) with a block size of 1024. The execution time was about 13 times faster than the basic algorithm. They're also much better compared to other optimized methods. The biggest disappointment is the vectorized way of multiplying matrices as it gave worse results than the basic method. The plan for the next project is to introduce distributed methods for holding calculations.

2 Problem statement

So far, I have managed to benchmark multiplication matrices with regard to different 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 with a special implementation and multiplication method. As a next step, I made use of the cores in the computer and ran the matrix multiplication on many threads. Since the optimized methods limited us in a certain point, the best option is to make some of these methods parallel. This time I implemented 2 parallel methods: a blocking multiplication algorithm and a parallel matrix multiplication using the method provided by the *Arrays* library. What's more, I benchmarked these methods, comparing them to the basic matrix multiplication algorithm and also to vectorized matrix multiplication. What I wanted to get was a clear comparison of all methods with calculated speedup to see the improvement.

3 Methodology

To measure the performance of each multiplication method, I use a Macbook Pro with an Apple M3 Pro chip and 18GB of RAM. The matrix sizes used to evaluate the performance of the algorithms range from 10x10 to 2000x2000 for all algorithms to get fair results. As a measurement tool I use 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. All the execution times are given in milliseconds.

4 Experiments

4.1 Basic Multiplication Algorithm

4.1.1 Introduction

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

4.1.2 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**
10	10^3	0.026	10^3
20	0.003	0.120	0.003
50	0.040	0.079	0.051
100	0.373	1.974	0.405
200	3.265	10.830	3.398
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

4.1.3 Chart

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

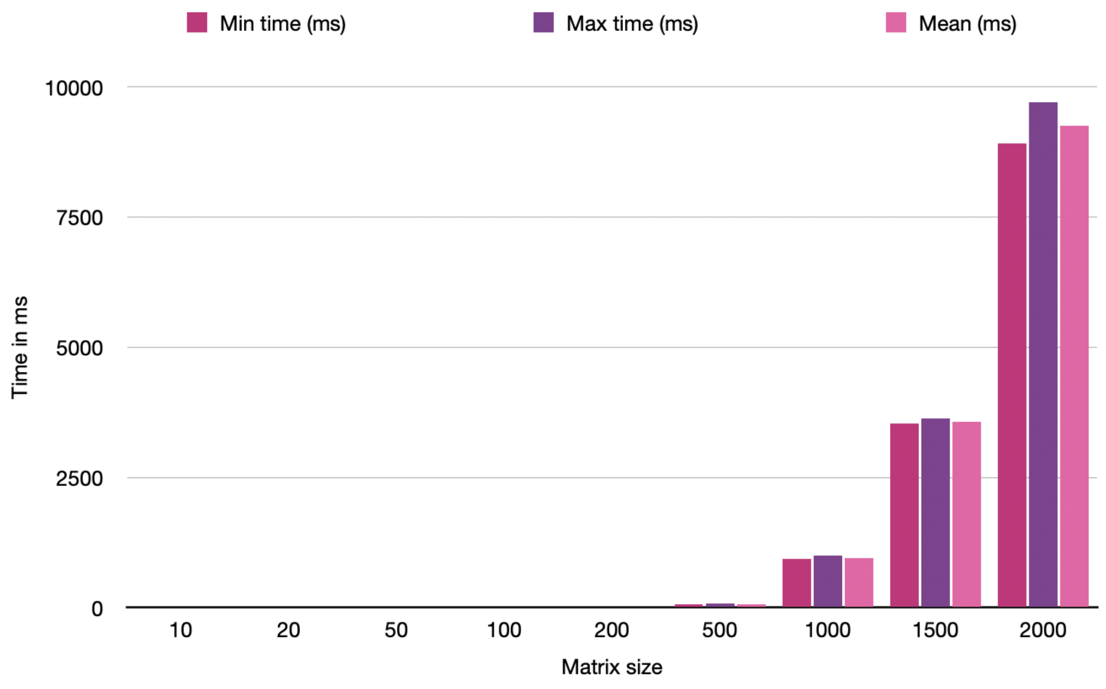

Figure 1: Visualized results for basic matrix multiplication

4.2 Parallel Blocking

4.2.1 Introduction

Blocking is a technique used to optimize matrix multiplication by dividing the matrices into smaller sub-matrices or blocks, which I elaborated on in the previous post. Now, let's take it a step further by accelerating the process through parallelization. This approach is ideal for parallel computing, as it allows us to process multiple blocks simultaneously, significantly speeding up the computation.

To obtain parallelism I used *ExecutorService* from Java with all available cores which is 11.

4.2.2 Execution time

As we can see, the results, especially for larger matrices, are much better than the basic algorithm. Furthermore, the use of a larger block size - 1024 for the largest matrices (1000, 1500, 2000) has resulted in a significant improvement, over 2 times faster than the 256 block size.

Size	Block size	Min time	Max time	Mean time
10	**256**	0.156	1.276	0.274
20	256	0.203	6.922	0.320
50	256	0.215	2.740	0.333
100	256	0.306	3.138	0.419
200	256	0.855	6.316	0.992
500	256	13.222	21.496	13.806
1000	256	187.957	202.375	192.554
1000	**1024**	46.268	62.849	48.432
1500	256	862.978	878.707	867.854
1500	**1024**	349.700	362.807	354.541
2000	256	2281.701	2399.142	2334.550
2000	**1024**	940.573	1157.628	1036.218

Table 2: Benchmarking results for parallel blocking matrix multiplication

4.2.3 Chart

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

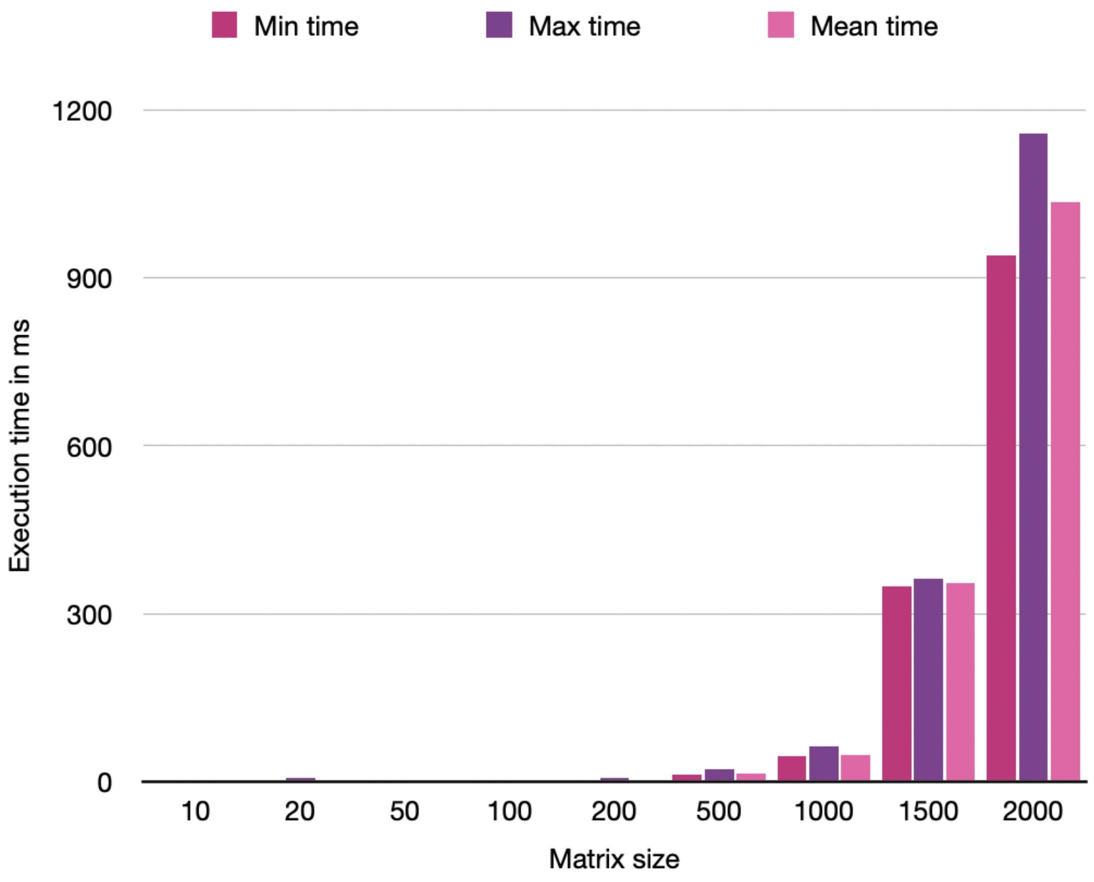

Figure 2: Visualized results for parallel blocking matrix multiplication

4.3 Parallel Arrays

4.3.1 Introduction

As a second parallel method, I implemented a parallelization provided by the *Arrays* library: the method *Arrays.parallelSetAll*, which is a parallel version of *Arrays.setAll* and allows parallel processing of each row in the result matrix. This method divides the task of calculating each row of the result matrix into separate threads and executes them in parallel for faster computation. The number of threads used for parallel execution depends on the available processors and the underlying parallelization framework used by Java.

4.3.2 Execution time

The execution times for this approach are very good, although slightly slower than the Parallel Blocking algorithm. However, the performance is still very good, with significant improvements over the basic method. While the blocking approach achieves slightly better results, this parallel method still offers efficient computation, especially for larger matrices.

Size	**Min time**	**Max time**	**Mean time**
10	0.001	1.571	0.010
20	0.003	1.176	0.018
50	0.014	19.726	0.036
100	0.086	1.167	0.118
200	0.537	3.613	0.603
500	9.486	11.944	9.875
1000	143.131	245.367	162.040
1500	571.474	637.534	578.762
2000	1505.755	1681.916	1557.974

Table 3: Benchmarking results for parallel arrays matrix multiplication

4.3.3 Chart

There is no significant difference between this chart and the previous one. What we can see is a slightly longer execution time.

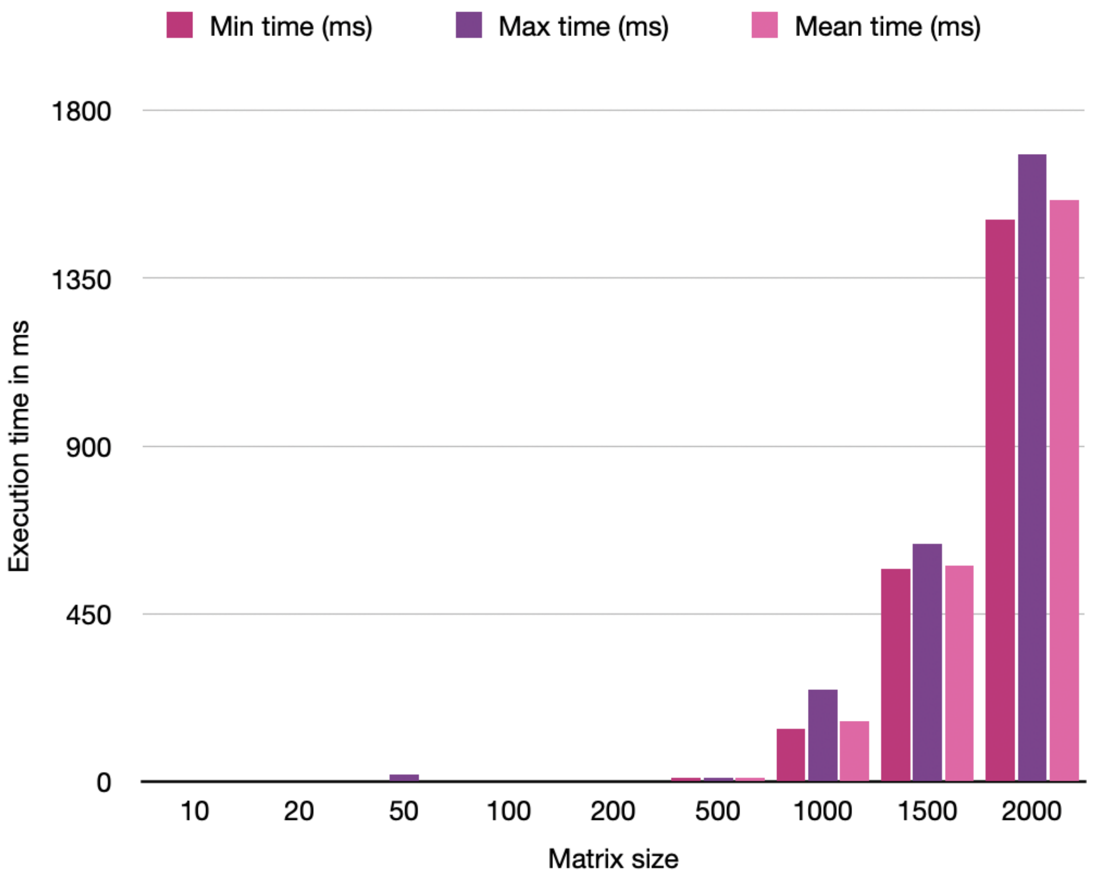

Figure 3: Visualized results for parallel arrays matrix multiplication

4.4 Vectorized Matrices

4.4.1 Introduction

This method performs matrix multiplication using a vectorized approach. It computes the dot product of each row from the first matrix with each column from the second matrix to produce the result. Although the code implements matrix multiplication in a vectorized way, it is not necessarily optimized for performance, which will become apparent later. I used the code provided by the professor.

4.4.2 Execution time

As I said, the results are not satisfactory. They are even worse than for the basic algorithm. This is because the method extracts every column of the second matrix in every iteration, which can be inefficient for large matrices. A more optimized version could avoid extracting columns repeatedly and access the matrix elements directly in a more efficient way.

After the disappointing results, I decided to try using the parallel method with *ForkJoinPool* and *RecursiveTask*. This improved the execution time significantly, with results more than five times faster—1900 ms for a matrix size of 2000 and 750 ms for a matrix size of 1500. While these results are still slower than the previous parallel methods, it was an interesting experience to combine these two approaches and see the effects.

Size	**Min time**	**Max time**	**Mean time**
10	0.001	0.442	0.001
20	0.005	0.508	0.006
50	0.065	12.370	0.076
100	0.486	2.433	0.532
200	3.822	4.833	3.958
500	77.332	107.086	78.693
1000	858.784	875.561	864.918
1500	3699.376	3854.565	3732.511
2000	9512.681	10636.755	10096.529

Table 4: Benchmarking results for vectorized matrix multiplication

4.4.3 Chart

Compared to previous results obtained from optimized methods, the values on this graph are very large, reaching 10000 for the largest matrix size.

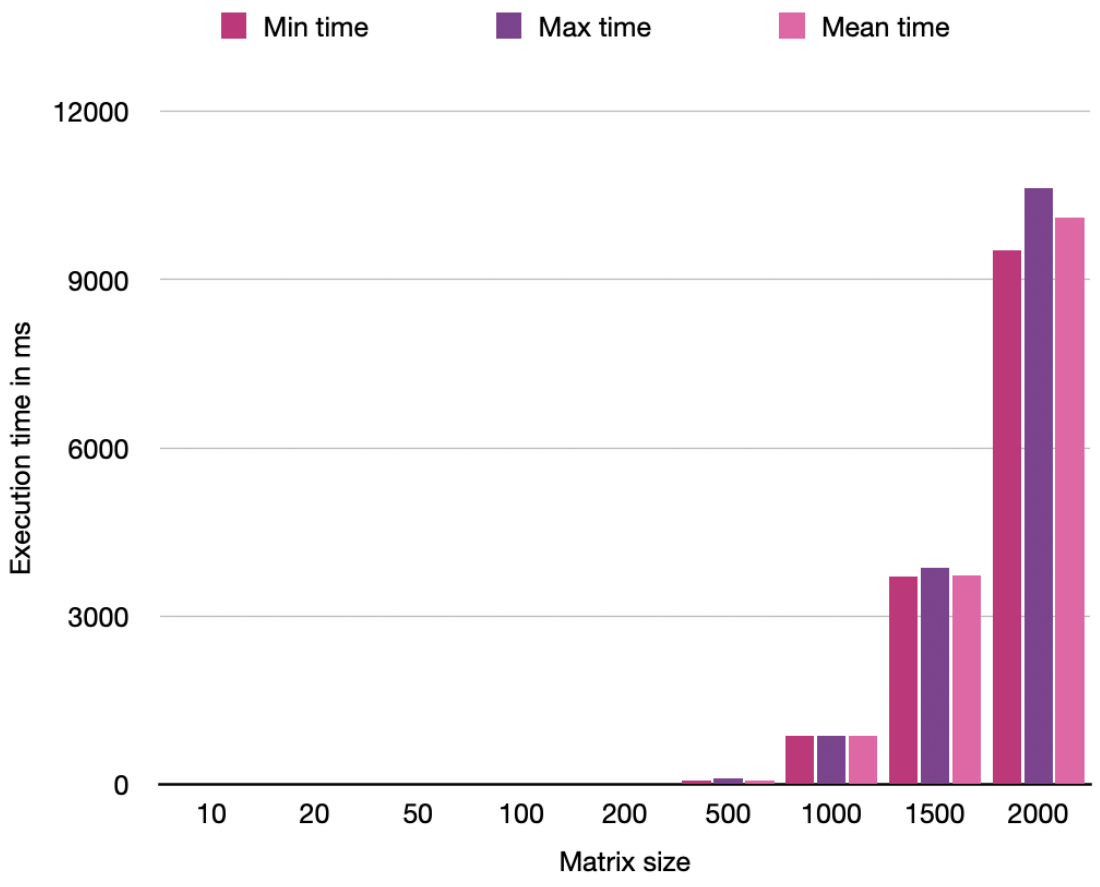

Figure 4: Visualized results for vectorized matrix multiplication

5 Conclusions

To sum up all the benchmarks, the Parallel Blocking algorithm is the unanimous winner. The results were the fastest and really stand out from the others. The biggest disappointment is the vectorized method, which gave unsatisfactory results similar to the basic algorithm.

5.1 Speedup

As shown in *Table 5*, the speedup for parallel methods compared to the basic algorithm is huge (3 and 5 times faster), but in reality it is even bigger because the basic approach copes better with smaller matrices. This is irrelevant in real computations, since nobody cares about matrices of size 10 or 20. So the speedup is large even if it is underestimated. This leads us to the conclusion that parallel methods give us a huge advantage over the basic one.

On the other hand, the speedup of the vectorized method is less than 1, which means that this approach performed worse.

Size	**Blocking Speedup**	**Arrays Speedup**	**Vectors Speedup**
10	0.00365	0.1	1
20	0.009375	0.166667	0.5
50	0.153153	1.416667	0.671053
100	0.966587	3.432203	0.761278
200	3.425403	5.635158	0.858514
500	4.478343	6.261063	0.785686
1000	19.611001	5.861516	1.098139
1500	10.076995	4.198646	0.957186
2000	8.937331	5.944273	0.917248
Average speedup	5.3	3.67	0.84

Table 5: Speedup results for different methods

5.2 Charts

The chart below illustrates the average execution times for all methods across various matrix sizes. As the size of the matrices increases, the differences in performance between the methods become more pronounced, with parallel methods showing a much larger improvement over others.

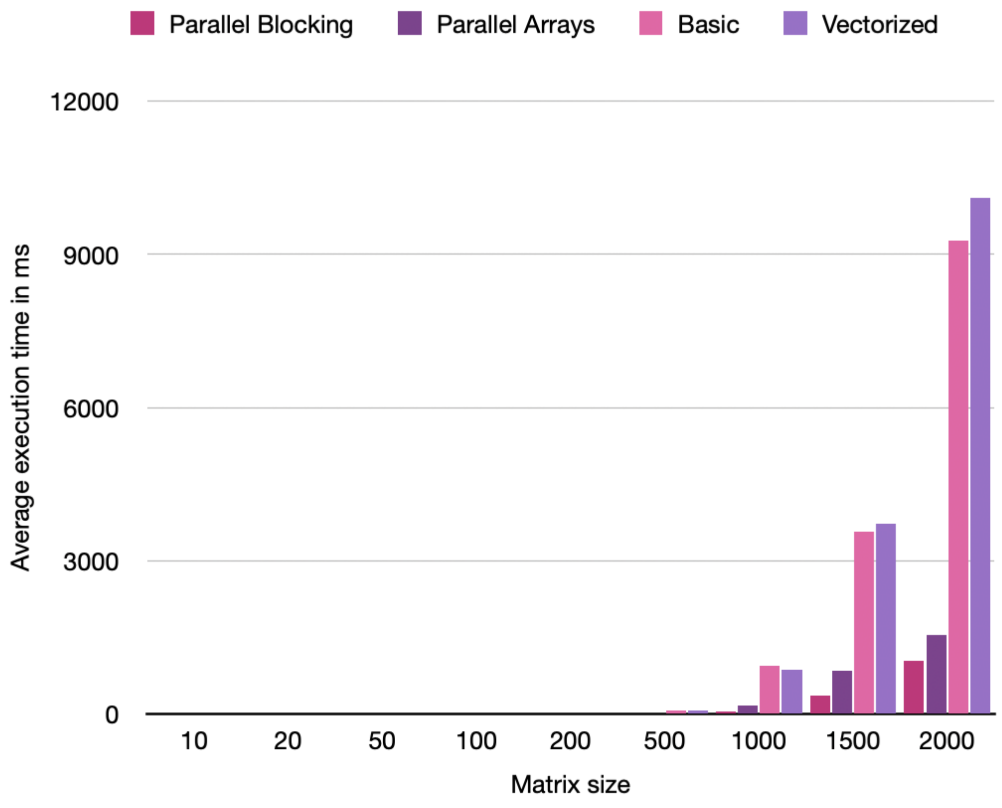

Figure 5: Visualized comparison between average execution time of all methods

The last chart compares the average execution times of the parallel multiplication methods. While the differences between the methods are not very large at smaller matrix sizes, the gap becomes more noticeable as the matrix size grows. Ultimately, the Parallel Blocking method consistently outperforms the other methods, showing the best results as the size increases.

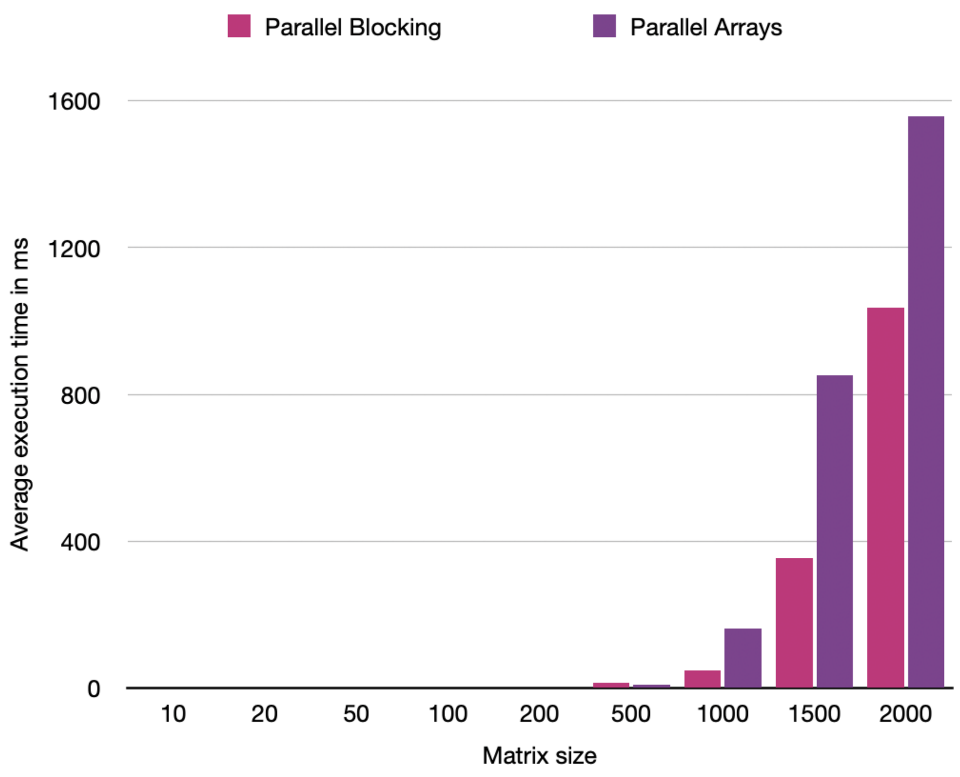

Figure 6: Visualized comparison between average execution time of parallel methods

6 Future work

Having experimented with different programming languages and optimization methods, including parallelization in this report, I think the next step is to distribute the computation across multiple machines. This will allow us to further improve performance and handle even larger matrices more efficiently. Distributing the workload would not only improve execution times, but also make the system more scalable, potentially leading to even greater improvements in speed and efficiency.