Benchmarking of matrix multiplication in different programming languages

Carlos Mendoza Eggers

October 1, 2023

Abstract

Nowadays, the choice of programming language can significantly impact performance, making it crucial to measure and compare their capabilities. This paper focuses on comparing the performance of Java, Python, C, and C++ using the matrix multiplication benchmarking algorithm.

For the benchmarking method, we use symmetric matrices with equal rows and columns, varying the matrix size (n) for testing. For each language, we measure the execution time of matrix multiplication operations, using different libraries or methods.

Performance comparisons reveal Python's slower execution times, likely due to its interpreted nature. C++ lags slightly behind the other languages, while Java demonstrates strong performance, particularly for larger n values.This research provides valuable insights for selecting the most suitable programming language based on performance, with Java emerging as a top choice.

Introduction

In the era of big data, the need to find the most efficient way to process data becomes increasingly important every day. To ensure efficiency, the processes need to be user-friendly and fast. For this reason, the choice of programming language for implementing such tasks becomes a matter of utmost significance, as it can significantly impact performance depending on the programming language used. Hence, it is necessary to measure each of them to determine the best choice.

This paper main focus is to compare the performance of diverse programming languages. These will be Java, Python, C and C++. The algorithm employed to assess the performance of these programming languages is that of matrix multiplication.

In this paper, we detail our benchmarking methodology, present results, and discuss their different perfomances. Our goal is to contribute valuable insights to ultimately determine which programming language appears to have the best perfomance.

Methodology

The methodology used for benchmarking the different programming languages was consistently the same.

Main idea

The primary idea of this research, as previously mentioned, focuses on measuring performance using matrix multiplication. In our case, the matrices used for the experiment were symmetric matrices.These have an equal number of rows and columns.So, the first task was to decide on the $n \times n$ matrices that would be part of the test.

The n values on which we decided to conduct the tests where concretely these ones:

$$1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 768, 1024, 1536, 2048.$$

As we can observe, there are 14 objects of study.

Having stated this, we proceed to explain the nature of each experiment. In each programming language, with its respective library or function, the execution time of the instruction performing the matrix multiplication was measured.In different ways, owing to the libraries used, but all with the same objective. For each n, the code was executed plenty of times.In the end, what we consistently obtained were the mean, minimum, and maximum execution times for each multiplication of two matrices with n columns and n rows.

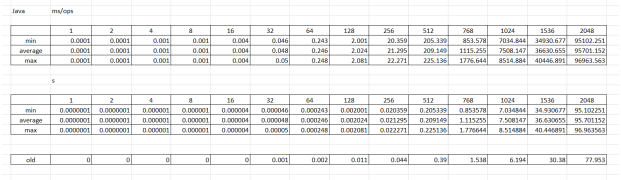
The libraries used for the benchmark in each language were these ones:

- Java: *Java Microbenchmark Harness (JMH).*[1]
- Python: *pytest.* [2]

[1] Github
[2] Pypi

- C and C++: None. Instead i coded a function.

All the information from the different measurements was collected manually in an Excel spreadsheet. This was done to facilitate its manipulation and control to subsequently create the charts.



Java	ms/ops													
	1	2	4	8	16	32	64	128	256	512	768	1024	1536	2048
min	0.0001	0.0001	0.001	0.001	0.004	0.046	0.243	2.001	20.359	205.339	853.578	7034.844	34930.677	95102.251
average	0.0001	0.0001	0.001	0.001	0.004	0.048	0.246	2.024	21.295	209.149	1115.255	7508.147	36630.655	95701.152
max	0.0001	0.0001	0.001	0.001	0.004	0.05	0.248	2.081	22.271	225.136	1776.644	8514.884	40446.891	96963.563
	s													
	1	2	4	8	16	32	64	128	256	512	768	1024	1536	2048
min	0.0000001	0.0000001	0.000001	0.000001	0.000004	0.000046	0.000243	0.002001	0.020359	0.205339	0.853578	7.034844	34.930677	95.102251
average	0.0000001	0.0000001	0.000001	0.000001	0.000004	0.000048	0.000246	0.002024	0.021295	0.209149	1.115255	7.508147	36.630655	95.701152
max	0.0000001	0.0000001	0.000001	0.000001	0.000004	0.00005	0.000248	0.002081	0.022271	0.225136	1.776644	8.514884	40.446891	96.963563
old	0	0	0	0	0	0.001	0.002	0.011	0.044	0.39	1.538	6.194	30.38	77.953

Figure 1: *Example of the Java spreadsheet.*

We can see that there is a row that contains the name old. It has been decided for a better tracking of the experiment to maintain the benchmark of the first task, which was done in a somewhat more manual manner.

Now, we will proceed to explain in each of the different cases the composition of each of the projects. Additionally, we will describe how we separate production code from test code and address any other relevant considerations to take into account.

There will be a link at the end of this section where you could acces the source code of the different programming projects.Nevertheless, a brief visual description will be provided.

Java

The code was divided as we have seen in class. The experiment tried to follow the same disposition.

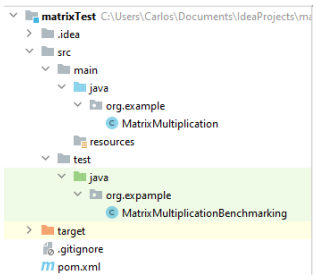


Figure 2: *Project Structure in Java.*

As we can see in 2, the code is mainly divided in two Java classes. The first one, ***MatrixMultiplication***, is where the execution of the multiplication belongs.

In the second class, **MatrixMultiplicationBenchmarking**, we can find the details regarding the benchmarking process. This class is implemented to facilitate the execution of the previously mentioned library, ***JMH (Java Microbenchmarking Harness)***. It includes a class named **Operands**, which serves as the state for ***JMH***, and within it, the ***Setup*** method is defined.

Towards the end of this class, you will find the *multiplication* method, which is where we measure the performance. This method executes the *execute* function defined in the ***MatrixMultiplication*** class.

Through the use of the ***JMH library***, we conduct benchmarking of the matrix multiplication. For this language, parameters such as *fork*, *measurement*, and *warmup* remained always the same due to the expected its expected time.

- **Forks:** 4.
- **Warmups:** 3.
- **Iterations:** 5.

To conclude this part, i would like to say that the execution time with this settings and for $n = 2048$ was 51 minutes.

Python

For the design of this project, I attempted to adhere to the structure used in the Java experiment. It was somewhat challenging to become familiar with the ***pytest*** library, but with some effort, we eventually achieved the desired results.

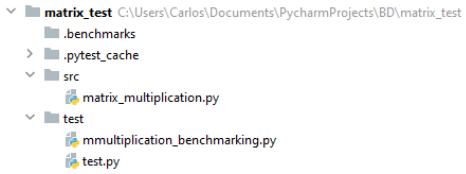


Figure 3: *Project Structure in Python.*

Actually, the structure of 3 follows the same design as the Java experiment. However, this time, the measurement is conducted in ***test.py***. The advantage of ***pytest*** is that it allows for measurement with similar efficiency as ***JMH***, or at least it appears to do so. In this case, the parameters changed as n increased because we assumed, based on the previous task, that the execution time in Python was significantly greater. The parameters used for *rounds*, *warmups*, and *iterations* for the different n's were as follows:

- $n = 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 768$.
  - **Forks: 4.**
  - **Warmups: 3.**
  - **Iterations: 5.**
- $n = 1024$.
  - **Forks: 3.**
  - **Warmups: 2.**
  - **Iterations: 4.**
- $n = 1536, 2048$.
  - **Forks: 2.**

- **Warmups: 1.**
- **Iterations: 3.**

I will also conclude this section by adding that the execution time for $n = 1536$ was 1 hour and 51 minutes, and for $n = 2048$ was 4 hours and a half.

C++

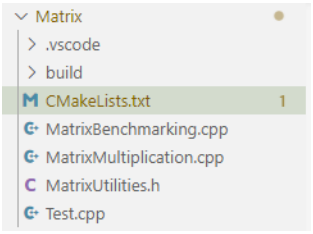


Figure 4: *Project Structure in C++.*

Once again,If we look at 4 I attempted to follow the same design as before. This time, we have three classes and a header file (**MatrixUtilities.h**), where the two main classes are defined. In the **MatrixMultiplication** class, you will find the **Operands** class, which initializes the three matrices involved in the multiplication, as well as the *execute* method, which performs the multiplication.

Since I wasn't able to use a specific library for benchmarking, I implemented a *test* function in the **MatrixBenchmarking** class. This function performs the multiplication multiple times and calculates the maximum, minimum, and mean of these values based on the rounds and iterations you specify. It essentially performs rounds x iterations executions, although it doesn't achieve the same level of effective benchmarking as we had with other libraries.

Finally, in the **Test** class, there is a *main*method that calls the*test* function.

C

This time it was not compulsory to measure the multiplication time in this language. Nevertheless, due to an issue that will be discussed later, the following approach was deemed suitable. The code from the previous implementation was modified in a certain way, although not entirely separated as in the previous languages. This was because this language, rather than being a distinct experiment in itself, was more focused on verifying the results obtained in C++.

The code is distributed in one file. It contains 3 functions: *multiplication*, *benchmark* and *main*. *multiplication* and *benchmark* are both void methods.*multiplication* takes the 3 matrix defined previously and execute the operation.

In the *main* function, we fill the matrices defined earlier and call the *benchmark* function. The *benchmark* function takes as arguments the number of iterations to be measured, and for each iteration, it measures the execution time using the method of subtracting the **start time** from the **end time**. The results are stored in a vector, from which the maximum, minimum, and mean values will be derived.

A very simple but efficient function for the result we desired.

Links

You can review the code of each project at the following link: `https://github.com/carlillous/matrixBenchmarking`.

Experiments

There won't be extensive commentary on the individual performance of each language separately because the foundation of this study is the comparison among them.

Java

For the Java Benchmarking test, we got the following results.

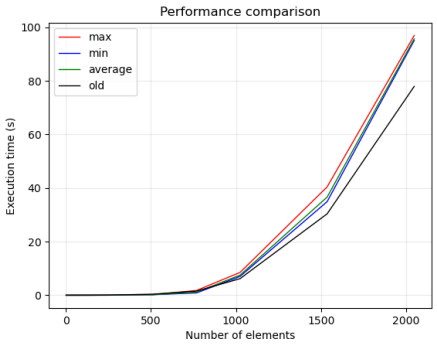


Figure 5: *Performance comparison overall.*

As we can se in 5 and 6, Java has a pretty decent execution performance if we look for it isolated. It has low execution time i would say until we get near 768 number of elements.

Python

If we haven't done the previous task, we would have been for sure surprised with the results of this experiment. Here we can see them.

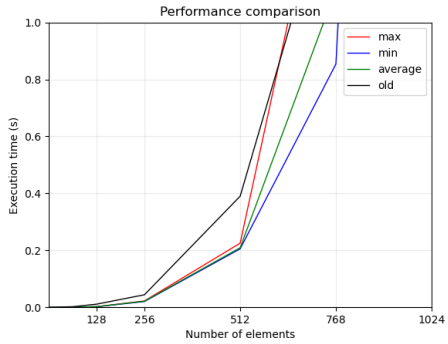


Figure 6: *Performance from 0 to 1 seconds of execution and to 1024 number of elements.*

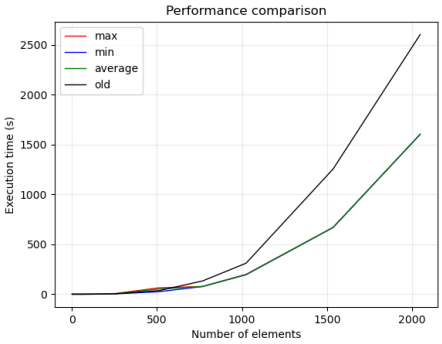


Figure 7: *Performance comparison overall.*

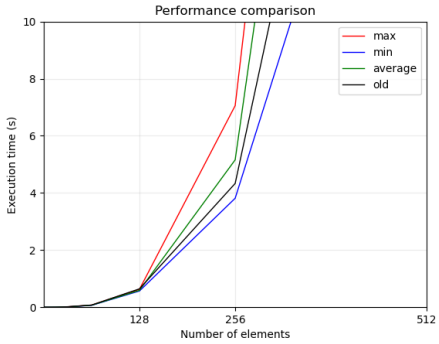


Figure 8: *Performance from 0 to 10 seconds of execution and between 0 and 512 number of elements.*

In 7, we can see that the measure using ***pytest*** was significantly better than the measurement we obtained manually.

It also illustrates that there is a point where the max, min, and average line are nearly the same. So in 8 and 9 we will search for the point when they diverge.

In 8 we can admire that the line representing the old measure, is still between the rest at 256 number of elements.

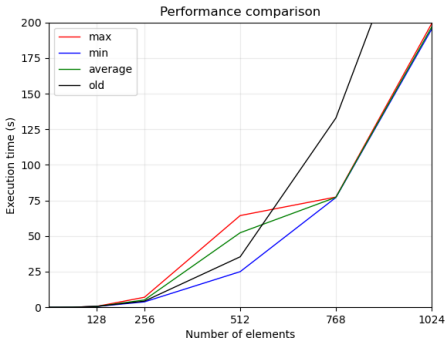


Figure 9: *Change at 512 number of elements*

And in 9, we can observe the point that the three lines diverge.

C++

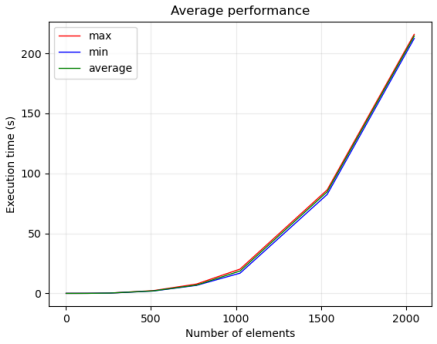


Figure 10: *Overall Performance in C++.*

There's nothing important to say about this graph. Only that we can see at 10 that the three lines make almost a same line. It is better to talk about this experiment in the final comparison.

C

The chart 11 shows that with a bigger number of elements, the old benchmark seems faster. Overall we can see that C is also a quite good language i would say with a low execution response.

In chart 12 we can see that there is a point, likely around 1200 elements, that the old measuring is slower than the new one.

Final comparison

We are almost at the end of the research. We have arrived at the most important part. The real comparison between all the programming languages.

At first in 13, we can see that there is a high difference between **Python** and the other 3 programming

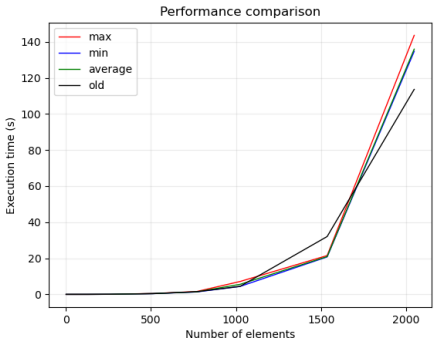


Figure 11: *Overall performance.*

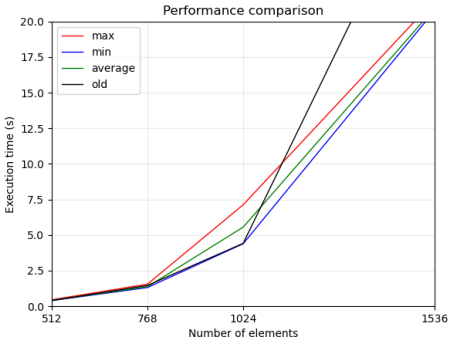


Figure 12: *Change of lines.*

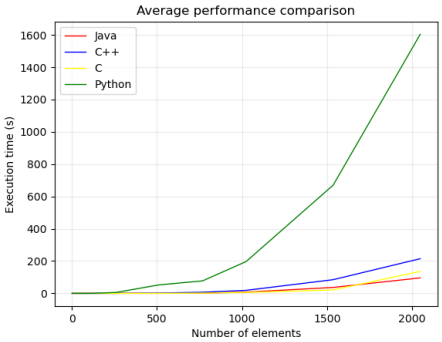


Figure 13: *Performance comparison overall.*

languages. So it seems we are discarding Python as a possible answer for our experiment.

The chart number 14 reflects that we will have to discard C++ in comparison to the rest. However, as we can observe in 13, it is not as poor in comparison to Python, for example.

Now here it comes. As we can see in Figure 15, Java seems to be slower than C for low n,despite it it still fast. However, at around 1700 as the number of elements in the rows and columns, it appears that Java becomes faster than C.

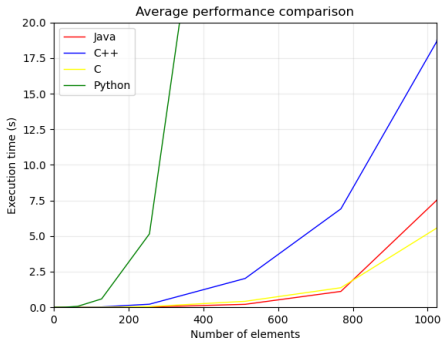


Figure 14: *Performance from 0 to 20 seconds of execution and between 0 and 1024 number of elements.*

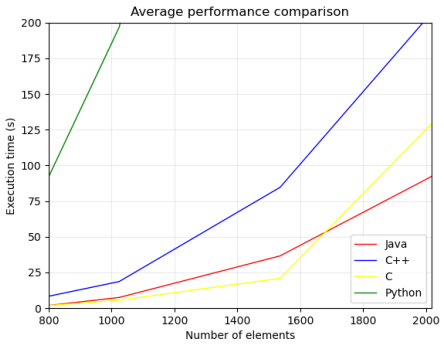


Figure 15: *High number of elements comparison*

Conclusion

After conducting a performance comparison of all programming languages, it appears that we have reached a conclusion.

If we solely consider performance, it is evident that we must discard **Python**. It seems to be a rather slow language when dealing with a large number of operations, possibly due to its interpreted nature. However, it's worth noting that **Python** is undeniably the most user-friendly among these languages.

Next, I would like to discuss the case of C++. When measuring its performance, I was surprised, as I initially thought that **C** and **C++** would perform equally. I still have some suspicions that this might not be the case, and there could be something amiss in the code implementation. Setting that aside and based on our results, we can see that **C++** stays slightly behind the other two remaining languages. Additionally, and personally, I am not really a **C++** fan.

Lastly, we have **C** and **Java**.These languages owe their speed to the fact that both are compiled, unlike Python, which is interpreted. As we observed earlier, **Java** appears to be faster than **C** when it comes to a higher number of operations. Furthermore, it is much

more intuitive than **C**. Thus, we arrive at the final conclusion: **Java** seems to be the best-performing language among all those we have analyzed.

Future work

It's evident that a comprehensive review of the C++ implementation is necessary to determine whether its performance accurately represents its potential or if there might have been issues related to code optimization or other aspects that influenced the results.

I also don't know how to use Latex properly and it made a mess with the charts. Next time I will work on it, and I hope to be able to improve it.