Matrix Multiplication Benchmark

Ángela López Dorta
Grado en Ciencia e Ingeniería de Datos
Universidad de Las Palmas de Gran Canaria

October 2025

https://github.com/angelalpzdor/Basic_Matrix_Multiplication.git

Project Description

This individual project was developed as part of the Big Data course. The main topic is matrix multiplication, which is a fundamental operation in many computational tasks, especially in the field of Big Data and high-performance computing. The objective was to perform a comparative study between different implementations of the same algorithm, using three programming languages: Python, Java, and C++.

The project focuses on the process known as benchmarking, which is the evaluation of the performance of a system or algorithm by running several tests under different conditions. In this case, the goal was to measure efficiency, speed, and memory usage when performing the same matrix multiplication algorithm in each language.

Benchmarking is important in computing because it allows to understand how algorithms behave with different workloads and data sizes. It involves collecting performance data and analyzing it to identify strengths, weaknesses, and possible optimizations. In Big Data, this process is essential because systems often need to process large datasets in real time, where performance and scalability are critical.

In this project, the benchmarking process was applied to compare several matrix sizes and to observe how performance changes when the size of the matrix increases. The analysis included both execution time and memory usage for each implementation.

The experiments were performed with matrices of different sizes, starting from small examples (64×64) to larger matrices such as 1024×1024. This made it possible to observe how execution time grows as the number of elements increases, and to identify possible performance limitations with large data.

Through this project, I gained a better understanding of how different languages handle numerical computations and how implementation details influence performance and scalability. The experience also helped to appreciate the importance of efficient algorithms and optimized code in Big Data environments, where the amount of data is very large and execution speed is crucial.

1 Introduction

This project studies the performance of matrix multiplication in three different programming languages: Python, Java, and C++. The main goal is to understand how execution time and memory usage change when the matrix size increases. Each program creates two random matrices and multiplies them using the classical algorithm with a time complexity of $O(n^3)$.

The results are recorded in a CSV file to compare the average time, standard deviation, and memory consumption of each language. By analyzing these results, it is possible to observe the differences in efficiency and optimization between interpreted and compiled languages. This study also helps to understand how algorithmic complexity and implementation details can affect program performance.

2 Why We Use the Transposed Matrix

In this project, the matrix B is transposed before the multiplication process. This means that the rows of B become columns in a new matrix B^T. The reason for this change is to improve the speed of the program by making better use of the computer's memory.

When a matrix is not transposed, the program accesses its elements in a way that is not efficient for the CPU cache. By using the transposed version, the program reads continuous memory positions, which reduces cache misses and makes the multiplication faster.

Although the mathematical result is the same, the transposed version allows the algorithm to work in a more efficient way without changing its complexity. This optimization is simple but important to increase performance, especially when working with large matrices.

3 Methodology

The experiment was made to compare how Python, Java, and C++ perform the same matrix multiplication task. For each language, a program was created to generate two random matrices of size $n \times n$ and multiply them using the classical algorithm with time complexity $O(n^3)$. Before the multiplication, the second matrix was transposed to improve the performance of memory access.

Each program was executed several times with different matrix sizes. The user entered the matrix size and the number of repetitions. During each run, the execution time and the memory used were measured. Both the real memory used and the theoretical (estimated) memory were calculated and recorded.

The theoretical memory was estimated with the formula $3 \times n^2 \times 8$ bytes, because the algorithm uses three matrices (A, B, and C) of double precision numbers, each taking 8 bytes per element. The real memory was measured with system tools: `psutil` in Python, `MemoryMXBean` in Java, and `getrusage()` in C++. This allowed the programs to show the difference between the expected memory based on the algorithm and the real memory consumed by the system.

In Python, the `time.perf_counter()` function was used to measure time, and in Java, the method `System.nanoTime()` was used. In C++, the `chrono` library was used for precise timing. After each execution, the results were saved in a common CSV file

called `benchmark_results.csv`, containing the average time, standard deviation, best and worst time, and both types of memory.

All tests were performed under the same computer conditions to make the comparison fair. By comparing the three implementations, it was possible to analyze not only the execution time differences but also how close the real memory usage was to the estimated value in each language.

4 Results from the CSV File

Language	Size	Mean (s)	Stdev	Best	Worst	Real Mem (MB)	Theo. Mem (MB)
python	64	0.0149	0.0008	0.0139	0.0158	0.10	0.09
python	128	0.1090	0.0014	0.1072	0.1106	0.34	0.38
python	256	0.8739	0.0078	0.8636	0.8827	0.94	1.50
python	512	7.2560	0.0372	7.2042	7.3045	1.78	6.00
python	1024	59.2112	0.8854	58.4106	60.5246	2.11	24.00
java	64	0.0014	0.0013	0.0008	0.0037	0.40	0.09
java	128	0.0041	0.0030	0.0025	0.0094	0.40	0.38
java	256	0.0167	0.0073	0.0127	0.0298	2.00	1.50
java	512	0.1012	0.0065	0.0956	0.1120	6.40	6.00
java	1024	0.9226	0.0815	0.8346	1.0391	28.36	24.00
cpp	64	0.0006	0.00003	0.0005	0.0007	0.09	0.09
cpp	128	0.0041	0.0018	0.0022	0.0066	0.38	0.38
cpp	256	0.0213	0.0033	0.0186	0.0261	1.50	1.50
cpp	512	0.1686	0.0145	0.1596	0.1944	6.00	6.00
cpp	1024	1.5235	0.0422	1.4894	1.5933	24.00	24.00

Table 1: Results comparing Python, Java, and C++ in terms of execution time and memory usage. All tests were repeated five times for each matrix size.

The data shows a clear growth in execution time and memory usage as the matrix size increases, which is consistent with the theoretical complexity $O(n^3)$. The smallest matrices (64×64 and 128×128) take only a few milliseconds to compute, but for larger matrices such as 512×512 and 1024×1024, the difference between languages becomes very visible.

C++ achieved the best performance in all tests, with the lowest mean times. For example, at size 1024, C++ completed the multiplication in about 1.5 seconds, while Java took almost 0.9 seconds and Python required around 59 seconds. This difference reflects the efficiency of compiled code and direct memory management in C++, compared to Java's virtual machine and Python's interpreted nature.

The standard deviation values in all languages are small, showing that each program gives stable results over repeated executions. The difference between the best and worst times also remains low, which indicates that there were no major fluctuations caused by external system processes.

In terms of memory, theoretical values grow smoothly following the formula $3 \times n^2 \times 8$, because the algorithm uses three matrices of doubles. Real memory values are close to the theoretical ones, with small variations depending on how each language manages memory internally. C++ and Java show a closer relation between real and theoretical memory, while Python has slightly higher real usage due to dynamic object handling and interpreter overhead.

For all three languages, the relation between size and time is exponential in practice, confirming that computational cost increases very quickly with matrix dimension. However, the transposed version of matrix B helped to reduce the impact of memory access delays, especially for the largest cases, improving cache efficiency and making the process faster.

5 Conclusion

This project made it possible to compare the performance of matrix multiplication in three languages. C++ was the fastest and most efficient, showing strong performance and stable memory usage. Java performed well but slower, mainly because of the virtual machine overhead. Python was the slowest but the easiest to implement and test.

The results show that compiled languages like C++ are the best choice for intensive numerical computations, while interpreted languages are more suitable for prototyping or educational purposes. Overall, this experiment demonstrates how algorithmic design, language implementation, and memory management directly influence performance in Big Data applications.