Benchmark of Matrix Multiplication
for Different Programming Languages

Ricardo Juan Cárdenes Pérez
Faculty of Computer Science
University of Las Palmas de Gran Canaria

Abstract

This paper is aimed at the challenging task it suppose Matrix multiplication, often involved in many of fields in Data Science. The standard matrix multiplication algorithm, usually used for small matrices, has a time complexity of $O(n^3)$. Thus, it takes a large amount of time to multiply matrices when their orders take several orders of magnitude. In this work, we focus on this problem and try to find the best way to implement such an algorithm in order to be able to apply matrix multiplication in Big Data problems. For this, we measured and compared the performance of this algorithm for different programming languages, giving several results to the reader to make it easier the decision making of which language should be used for a problem of this kind.

1 Introduction

Many existing problems can be expressed in terms of matrix multiplication. As an example, the relationships in a social network can be given as a matrix, often called the adjacency matrix, where each row and column refers to a person. This way, we can tell, for example, if there exists any relation between two people, with indexes i and j, and denoted by P_i and P_j, just by looking at the position (i, j) of the matrix. Thus, we say P_i is friends with P_j if position (i, j) of the matrix stores something different than zero.

By this point, we have perfectly described our network by using just one matrix, which has an order equal to the number of people on it. Now, a common task could be to find out communities, which could help us understand the network structure or even detect anomalies by the study of unusual behavior or events within the group. Mathematically, this is usually expressed as an optimization problem, and as the reader might guess, this implies matrix multiplication.

Social networks aside, this matrix operation is pretty used in many fields of computer science. Machine and deep learning, natural language processing, image processing, and bioinformatics are just a few examples of all the uses it has. It is this fact that made us go deep into this topic, and thus the necessity of finding the optimal way of multiplying large-scale matrices together in the shortest amount of time. As we will be using the standard multiplication algorithm only, we will make a comparison between different programming languages and machines, keeping the same code structure.

2 Matrix Multiplication Algorithm

Before starting the comparison, it is important to define the algorithm we are using. The matrix multiplication operation is mathematically defined as follows: Let $A \in \mathcal{M}_{n,p}(\mathbb{R})$ and $B \in \mathcal{M}_{p,m}(\mathbb{R})$ be two given matrices. The matrix $C = A \cdot B \in \mathcal{M}_{n,m}(\mathbb{R})$ is defined, for every entry c_{ij}, as

$$c_{ij} = \sum_{k=1}^{p} a_{ik} \cdot b_{kj} \tag{1}$$

Now, to efficiently compute this, we require a three-loop algorithm. The first loop computes A's rows, the second one handles B's columns. Thus, the innermost loop performs the element-wise multiplication and accumulation of products for each entry in the resulting matrix. This nested loop structure is essential for calculating the matrix multiplication of A and B while considering the impact n.

2.1 Standard algorithm

In our problem, we are only interested in multiplying square matrices. Thus, we will suppose $A, B \in \mathcal{M}_n(\mathbb{R})$ and so the resulting matrix $C \in \mathcal{M}_n(\mathbb{R})$ is defined for each entry as

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj} \tag{2}$$

It is now clear that, thinking of matrices as two-dimensional arrays, the first two loops described should go from position one to n, since the number of rows and columns take this value. The inner one will allow us to move k, and hence compute the result for the position (i, j) given by the other two loops. Once we have iterated through all possible values for i and j, we would have completed the algorithm.

In terms of pseudocode, the wanted algorithm looks as follows:

Algorithm 1: Matrix Multiplication

Data: Matrices $A, B \in \mathcal{M}_n(\mathbb{R})$
Result: Matrix $C \in \mathcal{M}_n(\mathbb{R})$

```
1 for i from 1 to n do
2     for j from 1 to n do
3         C[i][j] ← 0;
4         for l from 1 to k do
5             C[i][j] ← C[i][j] + A[i][l] · B[l][j];
```

This algorithm has a time complexity of $O(n^3)$ and will work for any pair of matrices that are compatible with multiplication, as long as you adjust the loop ranges accordingly.

2.2 Algorithm Implementation

The implementation of the algorithm is not hard at all. Indeed, given the pseudocode in section 2.1, the task is almost done. In Python, given A and B two 2-dimensional arrays of size nxn, and C another array of the same size previously set to all zeros, the code will look like the next:

```
for i in range(n):
  for j in range(m):
    for k in range(p):
      C[i][j] += A[i][k] *
                 B[k][k]
```

The rest of the languages we are comparing in this article, which are Rust, Node.js, Java, and C, have similar implementations. There are other ways to implement matrix multiplication as well, as could be using other techniques or algorithms such as Strassen's, which works better with high-order matrices, or some invented by Alpha Tensor. Anyway, in this paper, we will focus on the standard one, as our purpose is not to find the best algorithm, but the best programming language for solving this kind of problem.

3 Experiment

It is important before we start measuring to define the experiment we are going to run properly. Every implementation aspect should be clear before introducing any measurements so that the reader can have an idea of what is happening underneath the numbers.

3.1 Implementation

For this algorithm to be efficient in terms of memory, we will define just two matrices, $A, B \in \mathcal{M}_{2^n}(\mathbb{R})$, being n the number of iterations or times we want to test the algorithm. Now, to study time complexity, we need these matrices to have different orders on each iteration. The key here is that, by choosing the ranges of the different loops conveniently, we can simulate this operation for the orders we want, concretely, for the different powers of 2, even if the implied matrices have a much bigger one.

This way, for the given A and B, if we were to multiply two matrices of order $2^m < 2^n$, it would be enough to put 2^m wherever it says n in the algorithm described in Section 2. As a result, we obtain a program that can be easily tested by executing it into an outer loop that chooses the value for m sequentially.

Algorithm 2: Matrix Multiplication

Data: Matrices $A, B \in \mathcal{M}_n(\mathbb{R}), n > m$
Result: Matrix $C \in \mathcal{M}_m(\mathbb{R})$

```
1 for i from 1 to 2^m do
2     for j from 1 to 2^m do
3         C[i][j] ← 0;
4         for l from 1 to 2^m do
5             C[i][j] ← C[i][j] + A[i][l] · B[l][j];
```

3.2 Hardware

For this task, we used a PC of brand MSI, a Katana model, with 16GB of RAM memory, solid state disk, and processor Intel Core i7 (16 cores). All windows will be closed while executing, except those needed to run the matrix multiplication code. The PC will be connected to power every time, as it is known that charging the PC turns to make the computer faster.

PC Model Katana GF66 11UE-486XES (MSI)
GPU NVIDIA® GeForce RTX™3050 Ti

4 Languages Performance

This section will be aimed at measuring the time execution of the matrix multiplication algorithm in Python, Rust, Java, Node.js and C, just like we did for the previous task. These measurements will be useful in the following sections, allowing us to make benchmarks of them and will help us decide which language is more appropriate for the task we are undertaking.

To have reliable insights, we made our measures for matrices with orders of the first 11 powers of 2, which means orders up to 2048, and is enough for noticing a distinction between the languages. This way, assuming the time complexity of the algorithm, we will be able to make good predictions for higher orders, something not hard to see in Big Data problems.

4.1 Execution Times

In Figure 1 we can see that Python is the worst programming language in terms of execution time for matrix multiplication, just as it was expected. What was not that clear was that difference between C, Java, and especially, node.js, would turn out to be that small.

Now, does this make any sense? Actually, Java can outperform C when Java's Just-In-Time (JIT) compiler optimizes the code effectively, allowing it to leverage hardware-specific optimizations. Additionally, Java's multithreading support can lead to better parallelization, particularly on multi-core processors. Furthermore, Java's extensive standard library provides a wide range of built-in functionalities that can streamline development and improve code efficiency.

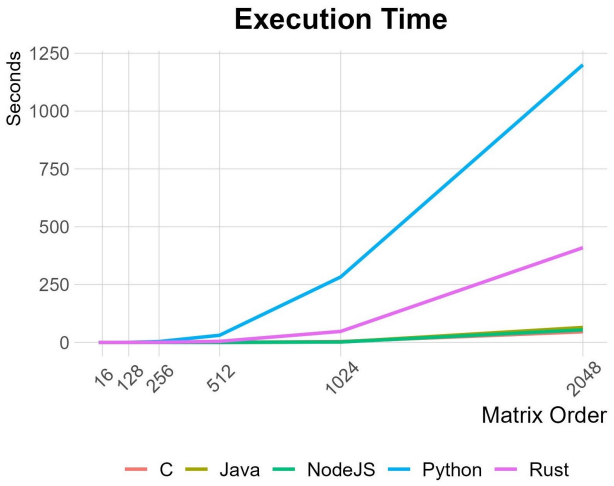

Figure 1: Execution time in all languages

Focusing on the difference between these three, in Figure 2, we can see it is not that large.

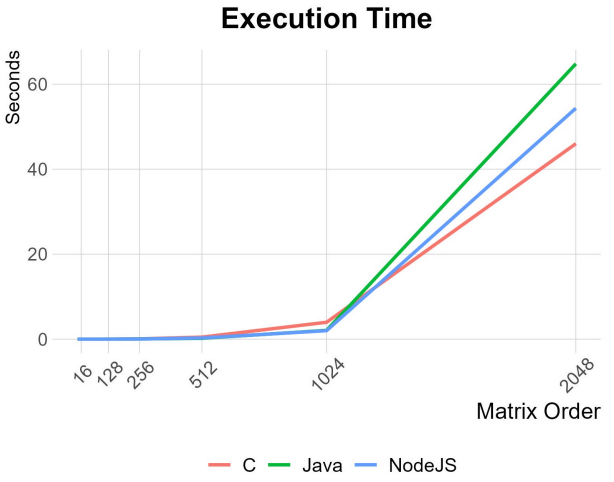

Figure 2: Execution time for fastest languages

Understanding why these outperforms other languages requires a closer examination of how different programming languages interact with the computer's architecture. This investigation has motivated us to delve deeper into the details of program execution and conduct a thorough analysis of various hardware factors that influence program performance. This exploration involves a comprehensive assessment of the hardware environment in which the program runs, illuminating the complex interplay between software and the underlying hardware components.

4.2 Hardware Aspects

Turning our attention to time efficiency, it is essential to focus on CPU management. The way a program utilizes the various CPU cores plays a pivotal role in determining the speed at which an algorithm completes. This factor can perfectly explain why Python's performance is notably inferior to Java's, for instance.

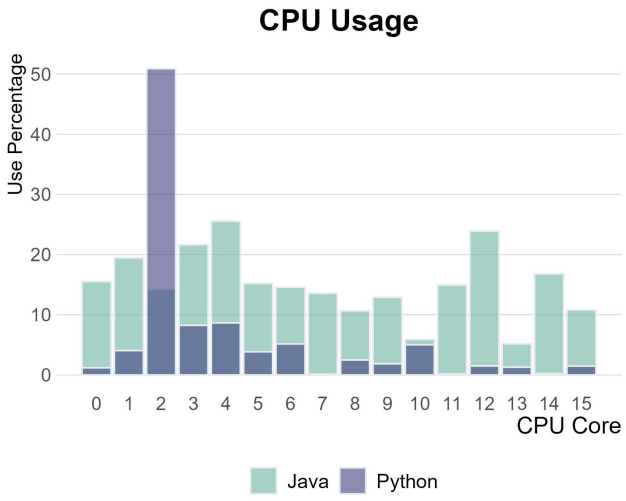

Figure 3: Replication Based Technique

As we can see in Figure 3, the program execution is distributed in a pretty bad way on the different cores for Python. Here, cores 2, 3 and 4 takes the most part of the job, meanwhile the rest of the cores are idle much of the time.

On the other part, Java's execution is distributed in a quite uniform way, where there are no idle cores at all, and cores tend to have the same work as one another. Here's why Java is better in terms of time. Java is well-suited for parallel processing and multithreading, allowing it to harness the full power of modern multi-core processors. This efficient core utilization makes Java an excellent choice for CPU-intensive tasks and applications where parallelism is essential, and that is the case of matrix multiplication.

5 Benchmark

By this time of the article, we have seen different properties and measures that made us know the power that Java language can offer when matrix multiplication is the field of discussion. In this scenario, we are ready to discuss the results obtained and, even more, relate them to other metrics such as OPS/ms.

5.1 Source Code

To achieve the goal of this work, it became necessary to split the matrix multiplication code into two parts: production and test. Production code contains the algorithm implementation of matrix multiplication itself. It is, let's say, the part of the project that will be saved for future versions and which gives the real functionality. On the other hand, the test part contains all benchmarks, the ones we have used to obtain the measures of previous sections, and the ones given in this. To do so, we have used several open-source libraries, such as 'jmh' for Java, 'Criterion' for Rust, or 'Pytest-benchmark' for Python. In the bibliography, you can find these different sources as a GitHub repository, Pérez [2023], where you have the link to a different repository for each language.

5.2 Operations per time unit

In the field of benchmarking, one crucial metric that holds immense importance in assessing the performance of software is 'Operations per Time Unit.' This metric, often denoted as ops/ms (operations per millisecond), quantifies the rate at which a system or application can execute a specific set of operations within a given time frame. Essentially, ops/ms provides a measure of efficiency, revealing how swiftly a piece of code or a computational task can be accomplished, matrix multiplication for our case, making it an indispensable yardstick for evaluating and comparing the effectiveness of different programming languages or hardware configurations.

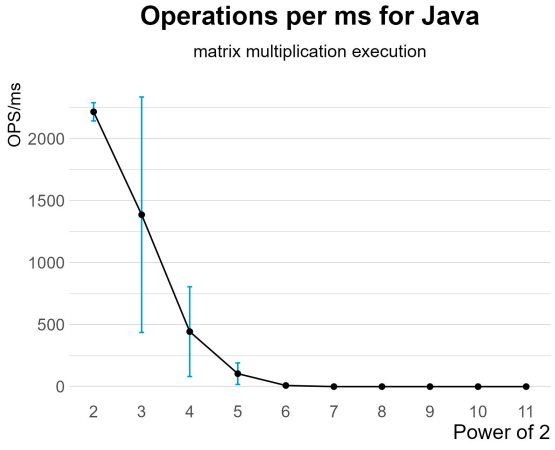

Figure 4: Time execution in all languages

The chart above shows Java's number of operations per millisecond for the matrix multiplication algorithm. As we can see, part of Java's outperformance can be clearly explained due to these values. The upcoming comparison figure, which will showcase ops/ms measures for Java, Python, and C, is a powerful tool for illustrating the differences between languages. If we observe the graph, we'll likely notice that Java and C are capable of doing many more operations than Python for first orders, indicating that these languages can complete a higher number of operations within the same time frame. This suggests both are more time-efficient for the specific tasks being benchmarked.

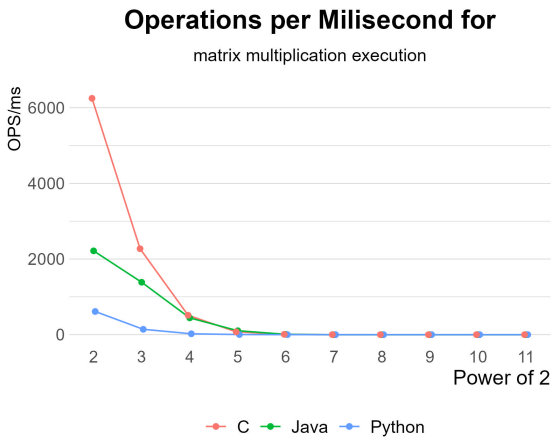

Figure 5: Time execution in all languages

The inherent nature of the system's force that, beyond a specific threshold, all values of OPS/ms become exactly zero, so it seems to be no difference between languages. This behavior stems from the practical constraint that prevents the entire operation from being executed within a single millisecond and the impossibility of getting the real value of OPS/ms due to working with a computer with finite-precision arithmetic.

However, the true significance lies in scenarios with lower-order matrices, where we can execute the function or process a significantly higher number of times. Here, and especially in Figure 5, we can notice that as the order of the multiplied matrices tends to 1, the operations per millisecond for C becomes much higher than in other languages. This shows how fast C can be for executing the simplest operations and, therefore, it would be the best option for taking this task.

5.3 Speed-Up

In the next table, you can see C's Speed-Up in time when comparing it to the rest of languages and taking matrices of different magnitudes.

	Python	Rust	Java	Node.js
$2^7 = 128$	x55.05	x10.59	x1.98	x0.25
$2^8 = 256$	x56.25	x8.49	x0.69	x0.41
$2^9 = 512$	x65.78	x9.82	x0.52	x0.57
$2^{10} = 1024$	x71.33	x12.02	x0.5	x0.5
$2^{11} = 2048$	x26.08	x8.90	x1.41	x1.18
Avarage Speed Up	54.89	9.96	0.99	0.58

Although the speed-up table is specifically designed for C, it provides valuable insights from which we can derive the varying speed-up factors for different programming languages. By analyzing the performance improvements achieved in C under specific conditions, we can extrapolate and understand how other languages might experience similar enhancements when subjected to similar optimizations or configurations, reason why we can not appreciate much difference between Java, C and Node.Js.

In essence, this table serves as a reference point for gauging the potential performance gains across the presented languages, offering valuable comparative data.

6 CONCLUSION

Based on the results of this experiment, we can confidently assert that the Python programming language is a sub-optimal choice for implementing a matrix multiplication algorithm, particularly when handling matrices of larger orders. Consequently, Python is not the most suitable option when confronted with Big Data problems of this nature. In stark contrast, Rust manages to significantly outperform Python in terms of execution time by a substantial factor.

However, the most commendable performers for this specific computational task undoubtedly include Java, Node.js, and C. While each of these languages boasts commendable attributes, for tackling such challenges, the preference leans toward Java or Node.js due to their robust execution capabilities and their use and support in Big Data problems. C, on the other hand, while highly efficient at a low level, is not typically employed for the complete development of projects of this magnitude.

References

Ricardo Cárdenes Pérez. Benchmark of matrix multiplication. https://github.com/ricardocardn/matrix_multiplication, 2023.