Basic Matrix Multiplication in Different Languages

Alejandro Alemán Alemán

Contents

Abstract

This study presents a comparative analysis of matrix multiplication algorithms implemented in three programming languages: Python, Java, and C. The purpose of this research is to evaluate the performance of each implementation based on execution time, memory usage, and scalability across different matrix sizes. Matrix multiplication is a fundamental operation in Big Data applications, where performance and resource efficiency are critical. By benchmarking the algorithms with increasing matrix sizes, we analyse the strengths and weaknesses of each language and identify potential areas for optimization. The results provide valuable insights into the efficiency of each approach and their suitability for large-scale data processing tasks in the field of Big Data.

1 Introduction

1.1 Background and Importance of Matrix Multiplication in Big Data Applications

Matrix multiplication is a fundamental operation in computational mathematics, supporting various applications like machine learning, graph algorithms, and scientific simulations. In the context of Big Data, where datasets are increasingly large and complex, efficient matrix operations are crucial for processing and analysing data in a timely manner. As data grows exponentially, optimising matrix multiplication is essential to ensure performance and scalability, especially for large-scale applications such as recommendation systems and machine learning models.

This study aims to explore the performance of matrix multiplication algorithms implemented in Python, Java, and C, with a focus on execution time, memory usage, and scalability as matrix sizes increase.

2 Problem Statement

2.1 Challenges of Matrix Multiplication in Big Data Applications

While matrix multiplication is essential for many Big Data applications, its high computational cost poses significant challenges, especially as matrix sizes grow. The standard $O(n^3)$ algorithm becomes inefficient with large matrices, leading to long execution times and high memory usage. In real-world Big Data scenarios, matrices can reach millions of elements, making the basic algorithm impractical for large-scale datasets.

2.2 Objectives of the Study

The primary objective of this study is to conduct a comprehensive comparison of matrix multiplication implementations in three widely used programming languages: Python, Java, and C. The study aims to assess the performance of these implementations based on key metrics such as execution time and memory usage, while also considering the scalability of each solution as matrix sizes increase.

This comparison will help identify which language offers the most efficient solution for matrix multiplication in terms of computational resources and scalability. Specifically, the study seeks to:

1. Evaluate Execution Time: Measure and compare the time taken by each implementation to multiply matrices of varying sizes, from small (10x10) to large (1000x1000), to analyse how each language handles increasing computational loads.

2. Assess Memory Usage: Analyse how much memory each language's implementation consumes during matrix multiplication, particularly for larger matrices, where memory management becomes a critical factor in performance.

By addressing these objectives, this study will contribute to a clearer understanding of the balance between ease of implementation, execution efficiency, and resource consumption in Python, Java, and C.

3 Methodology

3.1 Explanation of the Benchmarking Approach

The benchmarking process begins by implementing the same matrix multiplication algorithm in each language, ensuring that the logic and structure remain consistent across the implementations. The tests are conducted on matrices of varying sizes, starting with smaller matrices (e.g., 10x10) and scaling up to much larger matrices (e.g., 1000x1000), to assess how each language handles increasing computational loads.

Each matrix size is run 5 times with 2 warm up iterations, to account for variability in execution time due to factors like system background processes. To ensure accurate benchmarking, tools specific to each language are used. For instance, perf is employed in C to gather detailed performance metrics, while equivalent benchmarking tools are used for Python and Java. The benchmarking tests are also executed under similar system conditions to maintain consistency and fairness in the comparison. The results are then averaged, and the performance of each language is compared based on the metrics collected.

3.2 Description of the Matrix Multiplication Algorithms ($O(n^3)$) Complexity)

The matrix multiplication algorithm used in this study follows the traditional and most widely known method for multiplying two matrices, often referred to as the classical algorithm. This method operates with a time complexity of $O(n^3)$, where n represents the size of the matrix (for an n x n matrix). The algorithm involves performing three nested loops to calculate the resulting matrix.

Given two square matrices, A and B, of size n x n, the algorithm aims to compute the result matrix, C, also of size n x n. For each element in the result matrix C, the algorithm computes the dot product of the corresponding row from matrix A and the corresponding column from matrix B. The process can be described as follows:

1. Outer Loop: Iterates over each row i of matrix A.
2. Middle Loop: Iterates over each column j of matrix B.
3. Inner Loop: Performs the dot product calculation by iterating through the elements of the row from A and the column from B. For each index k, the algorithm multiplies the elements $A[i][k]$ and $B[k][j]$, then adds the result to the current value of $C[i][j]$.

The resulting matrix element $C[i][j]$ is computed using the formula:

$$C[i][j] = \sum_{k=0}^{n-1} A[i][k] \cdot B[k][j]$$

This requires n multiplications and n additions for each element in the result matrix. Since there are n^2 elements in the result matrix, the overall complexity of the algorithm is $O(n^3)$.

```
SQUARE-MATRIX-MULTIPLY(A, B)
1  n = A.rows
2  let C be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          c_ij = 0
6          for k = 1 to n
7              c_ij = c_ij + a_ik · b_kj
8  return C
```

Figure 1: Pseudo code of matrix multiplication algorithm implemented.

3.3 Overview of the Programming Languages and Tools Used

Each language has its own characteristics in terms of performance and optimization capabilities, which are explored through their respective benchmarking tools.

- **Python**: The Python implementation of matrix multiplication uses the ‘pytest-benchmark‘ library to measure performance. This tool allows for easy integration into Python’s testing framework and provides detailed results on execution time and memory usage.
- **Java**: In Java, the benchmarking is conducted using the Java Microbenchmark Harness (JMH). JMH is a well-established benchmarking tool for Java that provides accurate and detailed performance measurements, including execution time and optional memory usage tracking.
- **C**: For the C implementation, ‘perf‘ is used as the primary tool to measure performance. ‘perf‘ is a powerful Linux-based tool that allows for precise tracking of various performance metrics, including execution time, memory usage. It provides detailed insights into how the program interacts with the system, offering a low-level view of resource consumption.

These benchmarking tools were selected to ensure a fair and accurate comparison of performance metrics across the different programming environments, providing insight into how each language handles matrix multiplication under increasing computational loads.

3.4 Metrics Used to Measure Performance

The performance of the matrix multiplication algorithms is evaluated using three primary metrics:

Execution Time: This is the most critical metric, representing the time taken by the algorithm to complete the matrix multiplication. It is measured in milliseconds and is recorded for each implementation. The execution time is measured for varying matrix sizes, allowing for a comparison of how the performance of each language scales with increasing matrix dimensions.

Memory Usage: Memory consumption is another key factor in the performance of matrix multiplication algorithms. This metric tracks the amount of memory required by each implementation during the execution of the matrix multiplication. The memory usage is measured at different matrix sizes to evaluate how efficiently each language manages system memory.

3.5 Test Environment

The benchmarks were conducted on a system with the following specifications:
- Processor: Apple M1 (8-core CPU)
- RAM: 8GB Unified Memory
- Operating System: macOS Sequoia 15.0
- Storage: 512GB SSD

Each language’s implementation was executed under similar conditions, with specific tools and environments as follows:

Java: The Java implementation was executed using IntelliJ IDEA on the host system.
Python: The Python implementation was run using PyCharm on the host system.
C: The C implementation was executed within a virtual machine using UTM, running Fedora Linux. The reason for using a virtual machine was due to the lack of native support for performance analysis tools like perf on the macOS environment. However, it is important to note that the use of a virtual machine can introduce overhead and may affect the accuracy of the results. This is particularly relevant for the execution time and memory usage metrics, as the virtualization layer can introduce additional latency and resource management complexities. Despite this limitation, using Fedora Linux within the virtual machine allowed the use of perf to gather more detailed performance data for C.

4 Experiments and Results

4.1 Description of test cases

The matrix multiplication algorithms were tested with square matrices of varying sizes to assess how each implementation performs as the data size increases. The matrix sizes used for testing were:

Small matrix (10x10): A small-scale matrix used to measure the baseline performance of each implementation.

Medium matrix (100x100): A moderately sized matrix to examine how the algorithms handle increasing data complexity.

Large matrix (500x500): A larger test case to identify how well the implementations scale with more computational demands.

Very large matrix (1000x1000): A challenging test case to push the performance limits of each language, especially in terms of execution time and memory usage.

4.2 Presentation of performance data

The performance data for each programming language (Java, Python, and C) was collected in terms of execution time (ms) and memory usage (MB) for each matrix size. The results are shown in the tables below:

Table 1: Execution Time Comparison (ms)

Size	Java (ms)	Python (ms)	C (ms)
10	0.002	0.204925	0.191592800
100	0.811	56.9348332	1.890886733
500	117.102	8711.772900	170.532166267
1000	1332.356	74102.7506584	1380.003926400

Table 2: Memory Usage Comparison (MB)

Size	Java (MB)	Python (MB)	C (MB)
10	1	0.01	1.06
100	2	0.91	1.32
500	8	27.86	6.78
1000	15	36.75	24.444

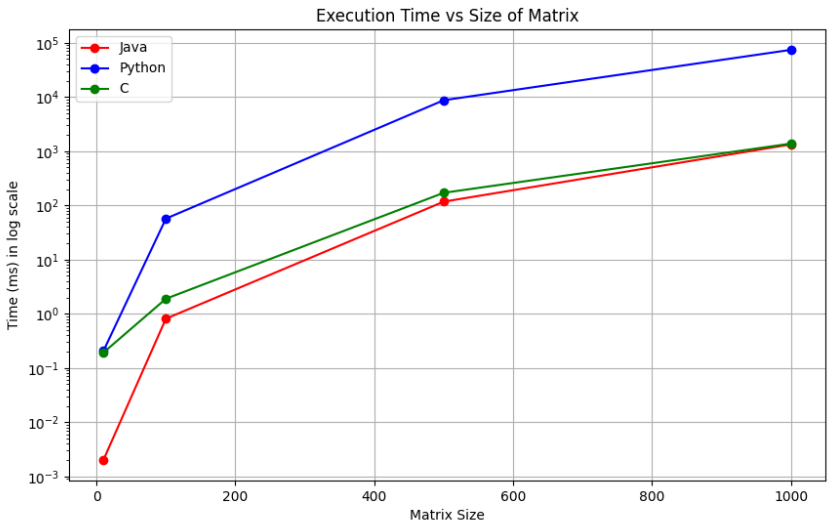

Figure 2: Execution time variation across different matrix sizes for each programming language

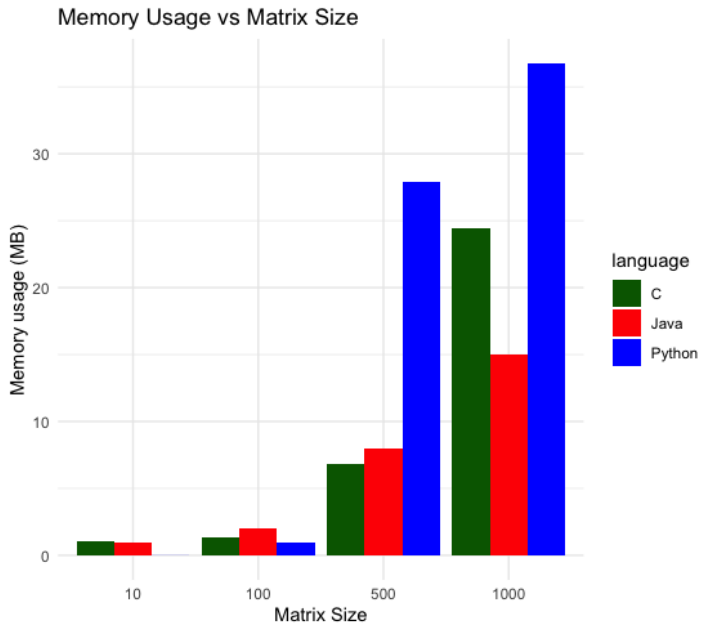

Figure 3: Memory usage during execution, varying by matrix size and programming language.

4.3 Discussion of Results

The performance comparison of matrix multiplication across Python, Java, and C, as presented in the tables and graphs, highlights significant differences in both execution time and memory usage as matrix sizes increase. The trends observed provide insights into the strengths and weaknesses of each language in terms of performance and scalability.

- Execution Time

  The graph "Execution Time vs Size of Matrix" shows a clear difference in how the three languages handle the increasing computational load as matrix sizes grow. For small matrices, such as 10x10, all three languages demonstrate minimal execution time. Java is the fastest, completing the operation in 0.002 ms, followed closely by C at 0.191 ms. Python, while still fast for smaller sizes, takes longer with 0.204 ms.

  As matrix sizes increase to 100x100, the differences between the languages become more pronounced. Java continues to perform well, with an execution time of 0.811 ms, while C remains competitive at 1.890 ms. However, Python begins to lag significantly, requiring 56.93 ms for the same matrix size.

  For larger matrices, such as 500x500, Python's execution time increases dramatically to 8711.77 ms, compared to 117.10 ms for Java and 170.53 ms for C. This trend continues for the largest matrix (1000x1000), where Python reaches 74102.75 ms, in stark contrast to Java's 1332.36 ms and C's 1380.00 ms. These results suggest that Python is not well-suited for large-scale matrix multiplication, while Java and C maintain efficient performance even as the data size grows.

  Overall, Java consistently outperforms both Python and C, with a more stable increase in execution time across matrix sizes. C, while slightly slower than Java for the larger matrices, still demonstrates strong performance compared to Python, making both Java and C better options for performance-critical applications.

- Memory Usage

  The graph "Memory Usage vs Matrix Size" highlights the differences in how each language manages memory as matrix sizes increase. For smaller matrices (10x10), memory usage is minimal across all three languages, with Python using 0.01 MB, Java using 1 MB, and C using 1.06 MB. For matrices of size 100x100, Python uses 0.91 MB, which is still relatively low but higher than Java (2 MB) and C (1.32 MB).

  As matrix sizes grow, Python's memory usage increases significantly, reaching 27.86 MB for a 500x500 matrix and 36.75 MB for a 1000x1000 matrix. In contrast, Java and C maintain much lower memory consumption, with Java using 15 MB for the 1000x1000 matrix and C using 24.44 MB.

As matrix sizes increase, Python's memory usage grows exponentially, making it less efficient for large-scale operations where memory management is critical. Java, on the other hand, maintains a consistent and low memory footprint, making it the most efficient option in terms of memory usage. C, while using slightly more memory than Java for the largest matrix, still manages memory much more efficiently than Python.

These results indicate that Python's high memory usage, particularly for larger matrices, makes it less suitable for applications where memory is a constraint. In contrast, Java demonstrates both low memory usage and fast execution times, making it the most balanced language in this study. C, while using slightly more memory than Java, still offers a good trade-off between memory consumption and execution speed.

5 Conclusion

5.1 Summary of Key Findings

This study compared the performance of matrix multiplication algorithms implemented in Python, Java, and C across various matrix sizes. The results demonstrate significant differences in both execution time and memory usage, with Java emerging as the most efficient language overall. Java consistently exhibited the fastest execution times and the lowest memory usage, making it the ideal choice for matrix multiplication in Big Data applications where performance and scalability are crucial.

C also performed well, offering competitive execution times, especially for larger matrices, and reasonable memory usage. However, it was run in a virtual machine, which might have introduced some overhead, suggesting that C's performance could be even more competitive in a native environment.

Python, while popular for its simplicity and flexibility, showed clear limitations in this study. Its execution time and memory usage grew exponentially as matrix sizes increased, making it unsuitable for large-scale matrix operations, particularly in resource-constrained environments.

In conclusion, Java is the best-performing language for matrix multiplication in terms of both speed and memory efficiency, followed closely by C. Python, though useful for smaller tasks or development purposes, is not recommended for performance-critical operations involving large datasets.

5.2 Future work

Future work could focus on optimizing matrix multiplication by exploring more efficient algorithms which can reduce the number of computations required. Additionally, implementing parallel computing approaches, such as using multi-core processors or GPUs, could significantly improve performance for larger datasets. Finally, distributed computing methods, like using cloud platforms, could be investigated to handle even larger matrices in Big Data environments.

A Link to GitHub repository

https://github.com/alejandroalemanaleman/MatrixMultiplication.git