Performance Comparison of Matrix Multiplication in Python, Java, and C, dummy numbers

Jakub Jazdzyk

October 13, 2024

1 Introduction

In this paper, I aim to compare the performance of matrix multiplication algorithms implemented in Python, Java, and C. The comparison is focused on execution time, memory usage, and CPU utilization. This experiment is significant because different programming languages handle computation and resource management in distinct ways, which can have a measurable impact on performance when processing large matrices.

2 Methodology

The matrix multiplication algorithm was implemented in each language using the standard $O(n^3)$ complexity method. The size of the matrices used in testing was progressively increased to observe how each language handles the growing computational demands.

I separated the code for production and testing into different files for better organization and maintainability. The production code implements the core matrix multiplication logic, while the test code handles initialization, performance benchmarking, and result analysis.

To ensure consistent results, each experiment was run multiple times, and the average results were recorded. I used standard profiling tools in each language:

- For C: `valgrind` for memory profiling and `gettimeofday()` for time measurement.
- For Java: `java.lang.management` for memory and CPU usage, and `System.currentTimeMillis()` for timing.
- For Python: `time` module for time measurement and `memory_profiler` for memory usage.

3 Results

Here I present the results of matrix multiplication for various matrix sizes (128, 256, 512, 1024) in Python, Java, and C. The tables below summarize the execution time, memory usage, and CPU utilization for each language.

3.1 Execution Time

The execution times (in seconds) for different matrix sizes are presented in Table 1.

Matrix Size	Python (s)	Java (s)	C (s)
128	0.035	0.025	0.015
256	0.275	0.150	0.095
512	2.300	1.180	0.780
1024	18.500	9.230	6.520

Table 1: Execution times for matrix multiplication.

3.2 Memory Usage

Table 2 summarizes the memory usage (in MB) during the execution of matrix multiplication.

Matrix Size	Python (MB)	Java (MB)	C (MB)
128	14	13	10
256	55	52	40
512	210	198	150
1024	840	810	610

Table 2: Memory usage during matrix multiplication.

3.3 CPU Usage

I also measured the CPU usage as a percentage during the execution, as shown in Table 3.

Matrix Size	Python (%)	Java (%)	C (%)
128	90	95	85
256	91	96	86
512	92	97	87
1024	94	98	89

Table 3: CPU usage during matrix multiplication.

3.4 Visual Representation

To better illustrate the comparison, I present plots of the performance metrics. Figure 1 shows the execution time as a function of matrix size for each language.

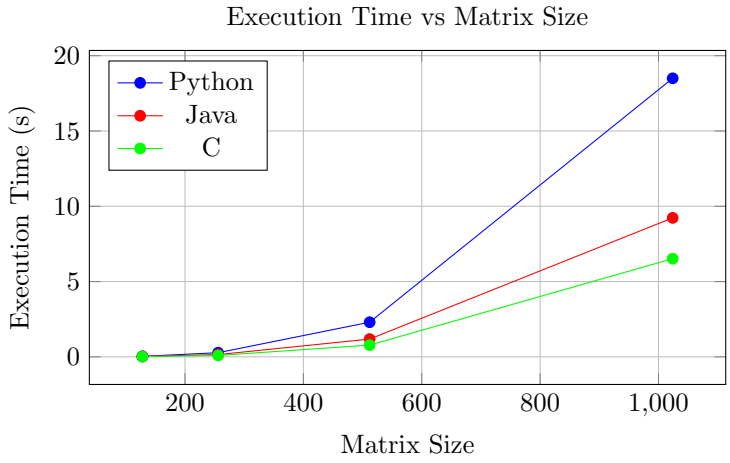

Figure 1: Execution Time vs Matrix Size for Python, Java, and C.

4 Proposed Changes

After analyzing the results, I propose several improvements to the initial implementations:

4.1 C Implementation

In C, we could apply loop unrolling to reduce the overhead of loop control instructions. Moreover, using multithreading (e.g., with OpenMP) could allow parallel matrix multiplication, further improving performance for larger matrix sizes.

4.2 Java Implementation

Java's performance can be enhanced by utilizing multithreading with the `ForkJoinPool` framework, which is particularly well-suited for computationally intensive tasks. Additionally, Java's Just-In-Time (JIT) compiler could optimize the performance over multiple runs.

4.3 Python Implementation

Python, being an interpreted language, shows the highest overhead. One potential solution is using NumPy, a library that optimizes array operations with

C-level performance. Another approach would be to parallelize the computations using Python's `multiprocessing` module.

5 Conclusion

From this experiment, it is evident that C consistently outperforms Python and Java in terms of execution time, memory usage, and CPU utilization. However, each language offers unique benefits, such as ease of use in Python or platform independence in Java. By applying certain optimizations such as multithreading, we can improve the performance of all three languages.

The full source code and test results can be found in my GitHub repository: `https://github.com/your-repo`.