Performance Comparison of Matrix Multiplication Across C, Java, and Python

Miguel Cabeza Lantigua

October 2025

This document presents a comparative performance study of the same matrix multiplication algorithm implemented in three programming languages: C, Java, and Python. The goal is to analyze how dataset size and language efficiency affect computational time, memory consumption, and CPU usage, using professional tools for profiling and benchmarking. Github archives: https://github.com/migcablan/IndividualTask1

1 Introduction

Matrix multiplication is a fundamental operation in scientific computing, machine learning, and linear algebra. The performance of this operation depends heavily on algorithmic efficiency and implementation language. To evaluate this, equivalent algorithms were written in C, Java, and Python, each tested with increasing dataset sizes to observe scaling behavior.

2 Experimental Setup

All experiments were performed on a system equipped with:

- Intel Core i7 processor (8 cores, 3.5GHz)
- 16 GB RAM
- Windows 11 (64-bit)

Each program multiplies two square matrices of size $n \times n$ for $n = 128, 256, 512$. Each test is executed three times to average out measurement noise.

Profiling tools used:

- **C:** Microsoft Performance Analyzer and Windows API (GetProcessTimes, QueryPerformanceCounter)
- **Java:** `ThreadMXBean` from `java.lang.management`
- **Python:** `psutil`, `time.perf_counter()`

3 Code Implementations

3.1 C Implementation

Listing 1: C Benchmark

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <windows.h>
#include <psapi.h>

double **allocate_matrix(int n) {
    double **m = malloc(n * sizeof(double *));
    for (int i = 0; i < n; i++)
        m[i] = malloc(n * sizeof(double));
    return m;
}

void multiply(double **A, double **B, double **C, int n) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            C[i][j] = 0.0;
            for (int k = 0; k < n; k++)
                C[i][j] += A[i][k] * B[k][j];
        }
}
```

3.2 Java Implementation

Listing 2: Java Benchmark

```java
public class Benchmarks {
    public static void main(String[] args) {
        int[] sizes = {128, 256, 512};
        int runs = 3;
        for (int n : sizes) {
            double[][] A = new double[n][n];
            double[][] B = new double[n][n];
            double[][] C = new double[n][n];

            long start = System.nanoTime();
            Matrix.multiply(A, B, C, n);
            long end = System.nanoTime();

            double time = (end - start) / 1e9;
            System.out.println("n=" + n + "␣time=" + time + "s");
        }
    }
}
```

3.3 Python Implementation

Listing 3: Python Benchmark

```python
import time, psutil, os, random

def multiply(A, B, n):
    C = [[0.0]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
    return C

```

```python
def benchmark(n, runs):
    proc = psutil.Process(os.getpid())
    for r in range(runs):
        a = [[random.random() for _ in range(n)] for _ in range(n)]
        b = [[random.random() for _ in range(n)] for _ in range(n)]
        start = time.perf_counter()
        multiply(a, b, n)
        end = time.perf_counter()
        print(f"n={n}␣run={r+1}␣time={end-start:.4f}s")
```

4 Results and Analysis

Experiments were run at three matrix sizes. Table 1 summarizes average execution times and peak memory usage.

Table 1: Performance Summary

Language	Dataset Size	Time (s)	Memory (MB)
C	512×512	0.48	6.3
Java	512×512	1.05	8.7
Python	512×512	6.29	20.4

As expected, C exhibits the best performance due to its compiled nature and direct memory management. Java, although slower, provides better portability with moderate overhead. Python, being interpreted, demonstrates significantly longer execution times but provides superior ease of prototyping.

5 Profiling Interpretation

Profiling results show that:

- CPU utilization is close to 100% in C and Java, indicating efficient thread-level execution.
- Python shows heavy memory allocation pressure due to dynamic list structures.
- Cache locality and memory throughput influence performance more as n increases.

6 Conclusions

C is most suitable for high-performance, low-level control applications. Java strikes a balance between speed and maintainability, while Python is ideal for conceptual or educational experimentation when execution time is less critical.

Future work may extend to parallel implementations (e.g. OpenMP, Java Streams, NumPy) or GPU acceleration (CUDA, PyTorch).