Performance Comparison of Matrix Multiplication Algorithms in C, Java, and Python

Juan López de Hierro

October 11, 2024

Abstract

This paper presents a performance comparison of matrix multiplication algorithms implemented in C, Java, and Python. The goal is to evaluate and compare the execution time, memory usage, and efficiency for different dataset sizes across these programming languages. The experiments were performed using benchmarking tools, and we discuss language-specific optimizations. The results reveal that Python is the slowest language, while Java shows the best performance. The source code and experiments can be found in the GitHub repository linked below.

1 Introduction

Matrix multiplication is a fundamental operation in many scientific and engineering applications. The performance of matrix multiplication algorithms can vary significantly depending on the programming language, optimization techniques, and hardware used. This paper compares the performance of matrix multiplication implemented in C, Java, and Python. We test each language with increasingly larger matrix sizes and record the execution time and memory consumption.

A folder containing the source code for all languages and the resulting paper can be found on GitHub: MatrixMultiplication.

2 Problem Definition

Matrix multiplication is defined as:

$$C[i,j] = \sum_{k=1}^{n} A[i,k] \times B[k,j]$$

where matrices A, B, and C are of size $n \times n$, and the algorithm has a time complexity of $O(n^3)$.

The task involves:

- Implementing the algorithm in C, Java, and Python.
- Testing the performance with matrices of sizes 256, 512, 1024, and 2048.
- Analyzing the profiling results using performance tools.

3 Methodology

3.1 Environment Setup

All experiments were conducted on a machine with the following specifications:

- Processor: Intel Core i7, 2.8 GHz
- Memory: 16GB RAM
- Operating System: Windows 10
- IDEs: IntelliJ for Java, CLion for C, Pycharm for Python

3.2 Languages and Tools

- **C**: Compiled with GCC and profiled using `perf`.
- **Java**: Executed and profiled using the Java Flight Recorder (JFR) and the built-in garbage collection logs.
- **Python**: Profiling done using `cProfile` and `time` module for runtime measurements.

3.3 Experimental Setup

For each language, we measured the execution time and memory usage while multiplying matrices of different sizes. Each test was repeated five times, and the average time was recorded. The matrix sizes tested were:

- 256x256
- 512x512
- 1024x1024
- 2048x2048

The Python implementation used lists of lists to represent the matrices, while Java and C used arrays.

4 Code Implementations

4.1 C Implementation

The matrix multiplication in C uses two-dimensional arrays and the `gettimeofday` function for time measurement. The complete code can be found in the repository.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

#define MAX_SIZE 2048
#define NUM_TESTS 5

double a[MAX_SIZE][MAX_SIZE];
double b[MAX_SIZE][MAX_SIZE];
double c[MAX_SIZE][MAX_SIZE];

void initialize_matrices(int size) {
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            a[i][j] = (double) rand() / RAND_MAX;
            b[i][j] = (double) rand() / RAND_MAX;
            c[i][j] = 0;
        }
    }
}

void multiply_matrices(int size) {
    for (int i = 0; i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            for (int k = 0; k < size; ++k) {
                c[i][j] += a[i][k] * b[k][j];
            }
        }
    }
}

void run_tests(int size) {
    struct timeval start, stop;
    double total_time = 0;

    for (int test = 0; test < NUM_TESTS; ++test) {
        initialize_matrices(size);

        gettimeofday(&start, NULL);
        multiply_matrices(size);
        gettimeofday(&stop, NULL);

        double diff = stop.tv_sec - start.tv_sec
                      + 1e-6 * (stop.tv_usec - start.tv_usec);
        total_time += diff;

        printf("Test %d para tamaño %d: %0.6f segundos\n", test + 1, size, diff);
    }

    double average_time = total_time / NUM_TESTS;
    printf("Promedio para tamaño %d: %0.6f segundos\n\n", size, average_time);
}

int main() {
    int matrix_sizes[] = {256, 512, 1024, 2048};
    int num_sizes = sizeof(matrix_sizes) / sizeof(matrix_sizes[0]);

    for (int i = 0; i < num_sizes; ++i) {
        int size = matrix_sizes[i];
        printf("=== Pruebas para tamaño de matriz: %dx%d ===\n", size, size);
        run_tests(size);
    }

    return 0;
}
```

4.2 Java Implementation

The Java implementation uses arrays and `System.nanoTime()` for measuring the execution time. Each test was repeated five times.

```
public class MatrixMultiplication {
    private int size;
    private double[][] a;
    private double[][] b;
    private double[][] c;

    public MatrixMultiplication(int size) {
        this.size = size;
        a = new double[size][size];
        b = new double[size][size];
        c = new double[size][size];
    }

    public void initializeMatrices() {
        Random random = new Random();
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                a[i][j] = random.nextDouble();
                b[i][j] = random.nextDouble();
                c[i][j] = 0.0;
            }
        }
    }

    public void multiply() {
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                for (int k = 0; k < size; k++) {
                    c[i][j] += a[i][k] * b[k][j];
                }
            }
        }
    }
}
```

4.3 Python Implementation

The Python implementation, being the slowest, used nested lists to represent matrices and the `time` module for runtime measurements.

```
import random

class MatrixMultiplication:
    def __init__(self, size):
        self.size = size
        self.a = [[0.0 for i in range(size)] for i in range(size)]
        self.b = [[0.0 for i in range(size)] for i in range(size)]
        self.c = [[0.0 for i in range(size)] for i in range(size)]

    def initialize_matrices(self):
        for i in range(self.size):
            for j in range(self.size):
                self.a[i][j] = random.random()
                self.b[i][j] = random.random()
                self.c[i][j] = 0.0

    def multiply(self):
        for i in range(self.size):
            for j in range(self.size):
                for k in range(self.size):
                    self.c[i][j] += self.a[i][k] * self.b[k][j]
```

5 Results and Discussion

The results from the performance tests are summarized in Table 1. Python consistently showed the worst performance due to its interpreted nature and lack of low-level memory management. Java performed better due to optimizations in the JVM, while C, being a compiled language with direct memory access, should have outperformed the others, but it was way slower than Java performation.

Matrix Size	**C (seconds)**	**Java (seconds)**	**Python (seconds)**
256x256	0.116262	0,040	6.354
512x512	1.177748	0,442	33.059
1024x1024	22.267350	6,472	235.189
2048x2048	178.340334	82,445	Too much time

Table 1: Execution Time for Matrix Multiplication (Average of 5 Runs)

In terms of memory usage, Python was also less efficient due to the overhead of its dynamic typing and list objects. Java showed better memory management, while C had the least memory overhead.

5.1 Profiling Results

Using `perf` for C, JFR for Java, and `cProfile` for Python, we identified bottlenecks in memory allocation and computation time. Python's overhead in list access and dynamic type handling greatly contributed to its slower performance.

6 Conclusion

This study highlights the performance differences between C, Java, and Python in matrix multiplication. Java, offers the best performance, while Python, due to its interpreted nature and lack of memory control, performs the worst. C, surprisingly, performs slower than Java. For computationally intensive tasks like matrix multiplication, languages like Java and C should be preferred for optimal performance.