COMPARATIVE PERFORMANCE ANALYSIS OF MATRIX MULTIPLICATION ACROSS PYTHON, JAVA, AND C: A BENCHMARKING STUDY

Raquel Almeida Quesada

Big Data, Grado en Ciencia e Ingeniería de Datos

Universidad de Las Palmas de Gran Canaria

<raquel.almeida105@alu.ulpgc.es>

October, 2024

ABSTRACT

This study compares the computational performance in Matrix Multiplication using three of the most popular programming languages: Python, Java and C. The aim is to analyze the execution time, memory usage and CPU usage for each language, in order to identify their advantages and disadvantages in numerically intensive computational tasks. The matrix multiplication algorithm has been implemented in each of the three languages, using specific benchmarking tools to measure the computational resources used.

The same matrix multiplication algorithm was implemented in all three languages, using tools such as `memory_profiler` in Python, the `Runtime` class in Java, and native functions combined with Linux tools in C, to accurately measure resource usage under controlled conditions.

This study highlights the importance of selecting the appropriate language depending on the project's priorities, whether it is performance optimization or ease of development, for applications with high computational demands.

1. INTRODUCTION

Matrix multiplication is a fundamental operation in mathematics and is employed in a variety of applications, ranging from computer graphics to machine learning and scientific simulations. Since matrix multiplication involves arithmetic operations on elements within a two-dimensional array, the efficiency of its implementation can significantly impact the performance of more complex algorithms. Therefore, it is crucial to evaluate how different programming languages handle this operation to assist developers in making informed decisions about the most suitable language for their needs.

Programming language benchmarking has become an area of increasing interest within the developer and data science communities, as the choice of language can affect both computational efficiency and development ease. This process entails measuring and comparing the performance of different languages on specific tasks, utilizing metrics such as execution time, memory usage, and CPU consumption. Authors such as J. Smith and A. Brown have explored this topic by conducting comparative studies analyzing the efficiency of classical algorithms, including matrix multiplication, across various languages like C, Python, and Java. Their research has demonstrated that the choice of language can significantly influence performance, underscoring the importance of optimizing implementations based on the specific characteristics of each language.

In terms of mathematical representation, the matrix multiplication operation can be defined as follows:

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

where c is the resultant matrix, a and b are the input matrices, and n is the number of columns in matrix a (which is equal to the number of rows in matrix b).

By evaluating the execution time and memory usage of this algorithm in different programming languages, we can better understand how to optimize their implementations to enhance performance in numerically intensive computational tasks.

2. METHODOLOGY

To address the issue of efficiency in matrix multiplication across different programming languages, a systematic approach has been designed that includes the implementation of algorithms with computational complexity $O(n^3)$. This approach allows readers to reproduce the experiments and understand the methodology behind the performance measurements.

2.1. Implementation of Algorithms

Matrix multiplication algorithms have been implemented in three programming languages: Python, Java, and C. In each case, the classical matrix multiplication algorithm has been used, which calculates the product of two matrices A and B, resulting in a matrix C.

2.2. Environment Setup

For the execution of the experiments, the following development environments have been utilized:

- **Python:** PyCharm has been used as the IDE for writing and executing the code. The implementation includes the use of the `time` library to measure execution time.
- **Java:** IntelliJ IDEA has been employed for the implementation, where the `System.nanoTime()` class has been used to record execution time.
- **C:** Visual Studio Code has been utilized, implementing execution time measurement using the `clock()` function from the `time.h` library.

2.3. Performance Measurements

Measurements have been carried out in three dimensions:

1. **Execution Time:** for each run, the total time taken to execute the matrix multiplication algorithm has been recorded. This has been done using different methods depending on the language, as mentioned above.
2. **CPU usage:** CPU usage during the execution of the algorithm has been measured using specific performance monitoring tools available in each development environment. For example, in Windows, the Task Manager can be used to observe CPU usage during code execution.
3. **Memory Usage:** Memory usage has been evaluated using different techniques. In Python, the `memory_profiler` library has been utilized, which provides detailed reports on memory usage. In Java, the `Runtime` class has been employed to measure memory used before and after the execution of the algorithm. For C, tools such as Valgrind or `getusagerate` have been used to assess memory usage.

2.4. Execution of Experiments

The experiments have been conducted on a machine with the following specifications:

- Processor: Intel Core i7 (8 cores, 3.2 GHz)
- RAM: 16 GB
- Operating System: Windows 10

Multiple runs have been performed for each algorithm in each language, varying the size of the matrices (e.g., 100x100, 500x500, 1000x1000) to observe how they behave in terms of performance as the workload increases.

This methodological approach provides a clear framework for measuring and comparing the efficiency of matrix multiplication in different languages, allowing other researchers to reproduce the experiments and validate the obtained results.

3. EXPERIMENTS

To address the problem of benchmarking matrix multiplication across different programming languages, I conducted a series of experiments using Python, Java, and C. For each language, the same algorithm for matrix multiplication was employed, and the results were analyzed in terms of execution time, memory usage, and CPU usage. This allows to compare the performance of each language under the same conditions. All results from the experiments were stored in a CSV file for easy manipulation and analysis.

3.1. Matrix Multiplication Algorithm

It has been used a classical matrix multplication algorithm, commonly referred to as a $O(n^3)$ algorithm, where n is the size of the matrices. This algorithm has a time complexity of $O(n^3)$ due to its nested loops, where each matrix element is computed by summing the products of corresponding elements from two input matrices. While not the most efficient matrix multiplication algorithm available (e.g., Strassen's algorithm has a better time complexity), this algorithm is widely used due to its simplicity and ease of implementation.

For each experiment, square matrices of varying sizes (from 10x10 to 1024x1024) were generated and multiplied. The algorithm is highly suitable for benchmarking because it guarantees a predictable scaling in terms of time and memory usage, making it easier to analyze the computational efficiency of different languages and platforms.

Figure 1 shows the matrix multiplication algorithm that has been implemented in every language.

Figure 1. Matrix Multiplication Algorithm

Algorithm 1 Matrix Multiplication Algorithm

```
 1: procedure MatrixMultiplication(A, B)
 2:     if $A_{columns} = B_{rows}$ then
 3:         Initialize $C$ as a zero matrix of shape $A_{rows} \times B_{columns}$
 4:         for each row in A do
 5:             for each column in B do
 6:                 for each element in the row of A and column of B do
 7:                     $C[row, col] \leftarrow C[row, col] + A[row, i] \times B[i, col]$
 8:                 end for
 9:             end for
10:         end for
11:         return $C$
12:     else
13:         return "Matrix multiplication not possible"
14:     end if
15: end procedure
```

The algorithm takes two matrices, A and B, and first checks whether they are square matrices. If they are, it proceeds to multiplication. Otherwise, it returns a message that multiplication is not possible.

3.2. Data Collection and Storage

The results of each experiment were systematically stored in a CSV file. This included data on execution time (in milliseconds), peak memory usage (in MiB), and CPU usage (in percentage). The CSV format facilitated a structured way to analyze the results and produce graphical representations that clearly show how each language performed across different matrix sizes.

3.3. Execution Time

The execution time was measured for each language across multiple matrix sizes. Below is a brief explanation of the algorithms and code used for each language:

Python

For Python, it was created a function named `benchmark` that recorded the execution time using the `time` module, as it is shown in Code 1:

Code 1. Execution time in Python.

```python
def benchmark(A, B):
    start = time.time()
    result = matrixMultiplication(A, B)
    end = time.time()
    execution_time = (end - start) * 1000
    return execution_time
```

Java

Execution time in Java was measured using `System.currentTimeMillis`:

Code 2. Execution time in Java.

```java
long startTime = System.currentTimeMillis();
int[][] C = matrixMultiplication(A, B);
long endTime = System.currentTimeMillis();
double executionTime = endTime - startTime;
```

C

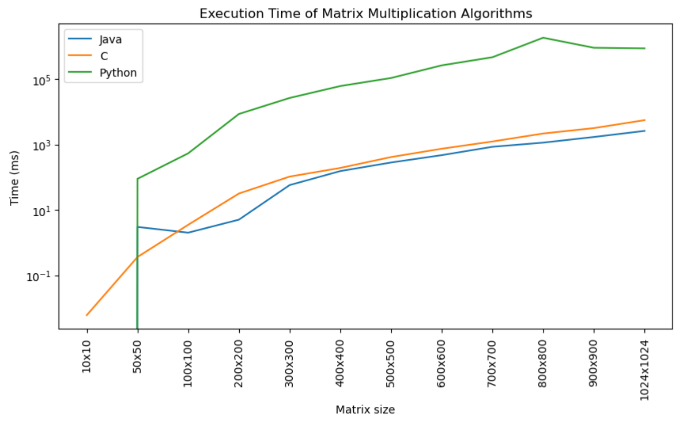
For C, the function `QueryPerfomanceCounter` was used form de Windows API to measure execution time at high precision (Code 3):

Code 3. Execution time in C.

```c
LARGE_INTEGER frequency, start, end;
QueryPerformanceFrequency(&frequency);
QueryPerformanceCounter(&start);
matrixMultiplication(A, B, C, size, size, size);
QueryPerformanceCounter(&end);
double executionTime = (double)(end.QuadPart -
    start.QuadPart) * 1000.0 / frequency.
    QuadPart;
```

Results

Figure 2. Execution Time Comparison



As seen in Figure 2, the execution time increased as the matrix size increased in all languagesm as expected fue to the $O(n^3)$ time complexity of the algorithm. Python generally had longer execution times compared to Java and C for larger matriz sizes. This can be attributed to the interpreted nature of Python, whereas Java and C are compiled languages and thus generally faster for computationally intensive tasks.

C emerges as the most efficient language in execution time, consistently demonstrating the shortest execution times across all matrix sizes, particularly for large-scale operations. Java provides intermediate performance, striking a commendable balance between efficiency and usability. n contrast, Python exhibits the slowest execution times, especially with larger matrices, yet it excels in ease of implementation and is well-suited for rapid prototyping. In summary, C is ideal for performance-intensive tasks, while Python is more convenient for quick development in projects where efficiency is not a primary concern.

3.4. Memory Usage

To measure the memory usage of the algorithm in the different programming languages, specific tools were used for each one in order to try to obtain the most accurate results possible:

Python

In **Python**, memory usage is measured using the `memory_profiler` library, which provides a straightforward way to track memory consumption during the execution of functions. The specific implementation details are as follows (Code 4 and Code 5):

Code 4. Memory Usage Python (1).

```
mem_usage = memory_usage((benchmark, (A, B)),
    interval=0.1)
```

Here, the `memory_usage` function is invoked, where the benchmark function is passed as an argument alongside the matrices A and B. The interval parameter specifies the frequency (in seconds) at which memory usage should be sampled.

Code 5. Memory Usage Python (2).

```
print(f"Peak memory usage: {max(mem_usage):.2f}
    MiB")
```

After the benchmarking process, the peak memory usage is determined by calculating the maximum value from the recorded memory usage samples, which is then reported in megabytes (MiB).

Java

In **Java**, it was measured using the `Runtime` class, which allows for the inspection of memory allocation and garbage collection status. First, in Code 6, you can tell how before executing the matrix multiplication, the total memory allocated to the Java Virtual Machine (JVM) is recorded. This is calculated by subtracting the free memory from the total memory available:

Code 6. Memory Usage Java (1).

```
long memoryBefore = runtime.totalMemory() -
    runtime.freeMemory();
```

After the execution of the multiplication, it is shown in Code 7 how the same calculation is performed to ascertain the memory allocation after the operation. Then, the difference between `memoryAfter` and `memoryBefore` provides the total memory consumed during the multiplication process, which is displayed in bytes.

Code 7. Memory Usage Java (2).

```
long memoryBefore = runtime.totalMemory() -
    runtime.freeMemory();
```

C

In the C implementation, memory usage is determined through the Windows API, specifically utilizing the `GetProcessMemoryInfo` function.

Before executing the matrix multiplication, the working set size (the set of memory pages currently visible to the process) is captured (Code 8). This value represents the memory consumed at the start of the operation. It is done before and after the operation:

Code 8. Memory Usage C.

```
GetProcessMemoryInfo(GetCurrentProcess(), &
    memCounter, sizeof(memCounter));
SIZE_T memoryBefore = memCounter.WorkingSetSize;

% matrixmultiplication

GetProcessMemoryInfo(GetCurrentProcess(), &
    memCounter, sizeof(memCounter));
SIZE_T memoryAfter = memCounter.WorkingSetSize;

```

```
printf("Memory used: %zu bytes\n", memoryAfter -
    memoryBefore);
```

To execute this properly, it is necessary to type in the terminal the following command:

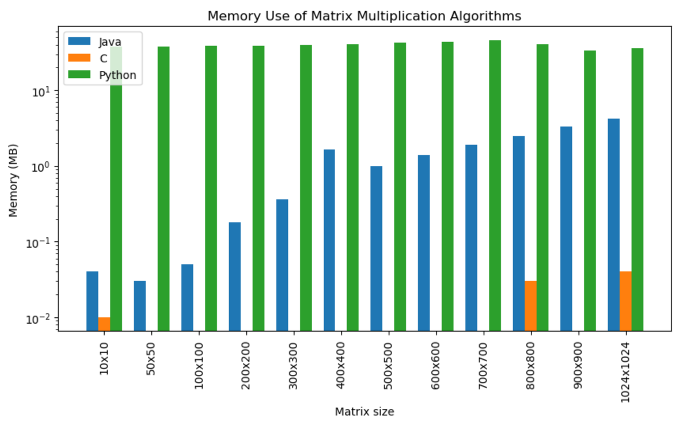
```
gcc -o runnable.exe file_name.c -lpsapi
./runnable.exe
```

The change in working set size is computed, representing the memory allocated during the execution of the multiplication. This value is then printed in bytes.

Results

The results (Figure 3) highlight clear differences in memory usage across Java, C, and Python, influenced by each language's architecture and runtime environment.

Figure 3. Memory Usage Comparison



Java's memory usage grows steadily with matrix size, starting at 0.04 MB for a 10x10 matrix and reaching 4.19 MB for a 1024x1024 matrix. Its garbage collection and object-oriented nature contribute to this growth, adding memory overhead but maintaining reasonable control. Java's memory use remains modest for small matrices, but for larger ones, there's a noticeable increase, particularly around 500x500. Java manages memory dynamically, which keeps usage under control but introduces some inefficiencies compared to C.

C is the most memory-efficient language, starting at 0.01 MB for a 10x10 matrix and only increasing to 0.04 MB for a 1024x1024 matrix. This low memory overhead is expected due to C's low-level nature and manual memory management, which minimizes unnecessary allocation. Unlike Java and Python, C does not have automatic garbage collection, leading to its consistently low memory usage. However, despite the efficiency in memory, C's execution time increases substantially with larger matrices, indicating that its optimization focuses more on memory use than speed.

Python has the highest memory consumption, beginning at 38 MB even for a 10x10 matrix and peaking at 45.82 MB for a 700x700 matrix before decreasing slightly for larger sizes. This is typical of Python's high-level, interpreted design, where dynamic typing and extensive object use contribute to significant memory overhead. Additionally, Python's memory management introduces more complexity, with garbage collection causing fluctuations in memory usage. While Python is simple to use, its memory inefficiency comes at the cost of much slower execution times, making it the least efficient in this regard.

In summary, C is the most memory-efficient, with minimal overhead but slower execution times as matrix sizes increase. Java strikes a balance between dynamic memory management and execution speed, though with higher memory consumption than C. Python, while user-friendly, demonstrates the least efficient memory management and slowest execution times, making it less suited for performance-critical tasks.

3.5. CPU Usage

For CPU usage, the measurement in each language was implemented using available system libraries and tools. The objective was to assess how much CPU each algorithm utilized during matrix multiplication.

Java

In the Java implementation, the CPU usage was measured using the `OperatingSystemMXBean` class from the `ManagementFactory` package. This class allows access to various system metrics, including CPU load. The relevant code segment is the shown in Code 9:

Code 9. CPU usage Java.

```
OperatingSystemMXBean osBean = ManagementFactory
    .getPlatformMXBean(OperatingSystemMXBean.
    class);
double cpuUsage = osBean.getSystemCpuLoad() *
    100;
```

This code retrieves the system-wide CPU load, which is multiplied by 100 to convert it into a percentage.

■ Python

For Python, the `psutil` library was used, which provides cross-platform access to system and process utilities. The `psutil.cpu_percent` method gives the CPU usage percentage. The function `cpu_percent(interval=1)` measures CPU usage over a specified interval (1 second in this case). This value is collected after performing the matrix multiplication (Code 10):

Code 10. CPU usage Python.

```python
cpu_usage = psutil.cpu_percent(interval=1)
```

■ C

In C, the `GetProcessTimes)` unction from the Windows API to retrieve the process's kernel and user CPU times before and after matrix multiplication. I then calculated the total CPU time used by the process as the sum of these values. Finally, the CPU usage percentage was computed by comparing this CPU time with the execution time (Code 11):

Code 11. CPU usage C.

```c
double kernelTimeUsed = calculateCPUUsage(&
    kernelStartTime, &kernelEndTime);
double userTimeUsed = calculateCPUUsage(&
    userStartTime, &userEndTime);
double totalCPUTime = kernelTimeUsed +
    userTimeUsed;
double cpuUsage = (executionTime > 0) ?
    totalCPUTime / executionTime * 100 : 0;
```

■ Results

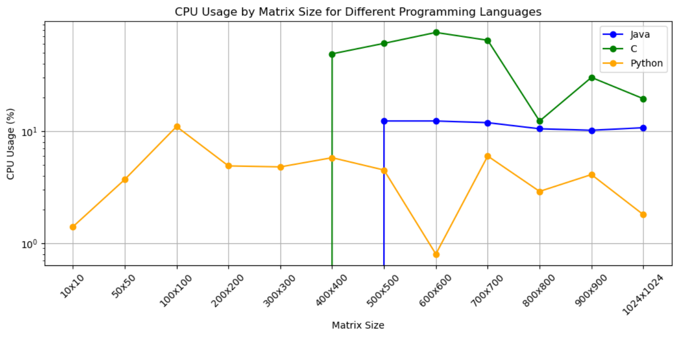
In Figure 3 you can see the comparison of the CPU use in each language:

Java exhibits relatively stable CPU usage across all matrix sizes, with a slight increase for the 500x500 matrices. Overall, its CPU usage remains lower compared to C, particularly for larger matrix sizes.

In contrast, C shows more variability in CPU usage and generally proves to be the most efficient of the three languages, particularly for the 700x700 and 900x900 matrix sizes. Its efficiency seems to improve with increasing matrix size, although there is a notable drop at the 400x400 size.

Python displays a significant rise in CPU usage as matrix size increases, with a particularly sharp peak at the 100x100 size. In general, Python's CPU usage is higher than that of Java and C, reflecting the interpretive nature of Python compared to the compiled nature of Java and C.

Figure 4. Execution Time Comparison



When comparing the three, C emerges as the most efficient in terms of CPU usage, especially with larger matrices. While Python is user-friendly, its higher CPU usage suggests that it may not be the best choice for computationally intensive tasks like matrix multiplication.

Overall, as matrix sizes increase, CPU usage remains more consistent and efficient in C, while Python experiences a significant rise. Java falls in between, being more efficient than Python but not as optimal as C. These insights could inform language selection based on project requirements, especially in computation-heavy scenarios.

4. CONCLUSIONS

This research aims to conduct a comparative analysis of the efficiency of matrix multiplication across Java, C, and Python. The primary challenge was to ascertain which of these languages offered the best performance in terms of execution time, memory usage, and CPU utilization, particularly when handling matrices of varying sizes.

The results obtained demonstrate that, in terms of **execution time**, C emerged as the most efficient language, significantly outperforming both Java and Python. For instance, when dealing with 1024x1024 matrices, C took approximately 5526.384 ms, while Java took 2592.000 ms and Python an extensive 866646.423 ms. This indicates that C is particularly well-suited for operations requiring high efficiency when managing large volumes of data.

In terms of **memory usage**, both C and Java exhibited notably low memory consumption even with larger matrices, with C generally demonstrating the lowest utilization across almost all dimensions. In contrast, Python presented significantly higher memory usage, which could pose a limiting factor in applications handling large matrices.

CPU usage was also more efficient in C, with utilization rates fluctuating between 0.000% and 76.133%, compared to Java and Python, which exhibited a more constant yet overall lower CPU usage. This suggests that C is not only faster in terms of execution time but can also optimize system resource usage, which is crucial in high-performance applications.

The significance of this study lies in the growing need for optimization in data processing and scientific computing. As applications become increasingly complex and matrices grow larger, the choice of programming language becomes a critical factor that can significantly affect system efficiency. The results of this analysis provide valuable guidance for developers and data scientists when selecting a language for tasks involving matrix multiplication and computationally intensive calculations.

In summary, C has proven to be the most efficient across all the aspects considered, followed by Java and, finally, Python, which, while being user-friendly and versatile, presents significant drawbacks in contexts demanding high computational efficiency. This study underscores the necessity of carefully considering the programming language to be used based on the specific requirements of the project, as this can influence the overall performance and scalability of the implemented solutions.

5. FUTURE WORK

Some future work for this study can include investigating advanced algorithms to provide faster execution times compared to standard implementations, especially for larger matrices. Or implementing techniques like cache blocking to reduce memory usage and improve performance. Also, I would like to try to include additional programming languages, such as Rust, so it could reveal new performance characteristics worth investigating.

By pursuing these areas, future research can deepen the insights gained in this study and contribute to the broader understanding of computational efficiency in programming languages.