1 GitHub Repository

The full source code and data for this project can be found at the following GitHub repository: https://github.com/daliaxx/BigData

Matrix Multiplication Benchmarking: Java, Python, and C

Dalia Valeria Barone

October 10, 2024

Abstract

Matrix multiplication is a core operation in numerous computational applications, particularly in the fields of Big Data and high-performance computing. The efficiency of matrix multiplication depends on the computational complexity of the algorithm, typically $O(n^3)$, and the underlying implementation in different programming languages. This paper presents a comparative study of matrix multiplication implemented in Python, Java, and C, focusing on execution time and memory usage as key performance metrics. The benchmarking was conducted on matrices of increasing sizes, ranging from 10x10 to 1024x1024. Using tools like JMH for Java, psutil for Python, and perf for C, we measured the performance of each language under controlled conditions. The results show that C performs the best in terms of execution time and CPU utilization, making it the most suitable for large-scale matrix operations. Java provides a balanced approach but consumes more memory, while Python offers ease of development at the cost of higher execution times. These findings provide valuable insights for developers and researchers, especially in scenarios involving Big Data, and highlight the importance of selecting the right language based on performance needs. Future work will explore optimized algorithms, parallelization, and GPU-based implementations to further improve efficiency.

2 Introduction

Matrix multiplication is a fundamental operation in many computational applications, particularly in Big Data and high-performance computing. It plays a crucial role in processing and analyzing large datasets, where efficient resource usage and execution speed are vital. In this assignment, we compare the performance of a basic matrix multiplication algorithm implemented in three different programming languages: Python, Java, and C.

This study aims to analyze the efficiency of each language in handling matrix multiplication by measuring key performance metrics such as execution time and memory usage across varying matrix sizes, from small (10x10) to large

(1024x1024). The goal is to provide insights into the computational overhead and scalability of each implementation.

3 Problem Statement

Matrix multiplication has a computational complexity of $O(n^3)$, meaning the time required for the operation increases significantly as the matrix size grows. In this assignment, we measure how Python, Java, and C implementations of matrix multiplication scale with matrix size, focusing on two key metrics:

- **Execution Time**: The time taken to multiply two matrices of increasing sizes.
- **Memory Usage**: The amount of memory required during the multiplication process.

The study provides insights into which language offers better scalability and resource efficiency in the context of Big Data matrix multiplication tasks.

4 Methodology

4.1 System Setup

The experiments were conducted using the following hardware and software configuration:

- **Processor**: Intel Core i7-1165G7 @ 2.80GHz
- **Memory**: 8 GB RAM
- **Operating System**: 64-bit Windows 10
- **Languages Tested**: Python 3.9, Java 21, and C (compiled with GCC)
- **Virtual Machine**: Debian was used for running the C experiments.
- **Development Environments**:
  - **Java**: IntelliJ IDEA was used for running the Java benchmarks with JMH.
  - **Python**: PyCharm was used for running the Python experiments.
  - **C**: The C program was run on a virtual machine with Debian installed, utilizing the terminal for compiling and executing the program.

5 Java

5.1 Benchmarking Tool

The Java implementation was tested using the Java Microbenchmark Harness (JMH), which allows for precise measurement of execution time and memory usage.

5.2 Benchmarking Parameters

Execution Time: Measured using `System.nanoTime()`, capturing precise start and end times for each matrix multiplication, with results expressed in milliseconds.

Memory Usage: Measured using `MemoryMXBean` from the Java Management API, which tracks the heap memory consumed by the JVM during the operation. Results are expressed in megabytes (MB).

Benchmark Configuration:

- **Benchmark Mode**: `Mode.SingleShotTime` was used to measure the time taken for a single matrix multiplication operation.
- **Time Unit**: Results were displayed in milliseconds using `@OutputTimeUnit(TimeUnit.MILLISECONDS)`.
- **Parameterization**: Matrix sizes (10x10, 100x100, 512x512, 1024x1024, and 2048x2048) were parameterized using `@Param`, automating the benchmarking process.

5.3 Setup and Running the Benchmark

To set up and run the Java benchmark, follow these steps:

1. Setting up JMH:
   - Add the following to your `pom.xml` file:

```
<dependencies>
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-core</artifactId>
        <version>1.35</version>
    </dependency>
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-generator-annprocess</artifactId>
        <version>1.35</version>
    </dependency>
</dependencies>
```

2. Running the Benchmark: After setting up the dependencies, you can build and run the project using the following commands:
    - `mvn clean install`
    - `mvn exec:java -Dexec.mainClass=org.example.MatrixMultiplicationBenchmarking`

5.4 Results

Matrix Size	Execution Time (ms)	Memory Usage (MB)
10x10	10.865	8 MB
100x100	23.110	9 MB
512x512	737.384	14 MB
1024x1024	8036.693	26 MB

Table 1: Java Benchmarking Results

5.5 Conclusion

The Java benchmarking analysis shows that performance scales with matrix size, as expected given the $O(n^3)$ complexity. Both execution time and memory usage increase with matrix size, highlighting the computational demands of matrix multiplication in Java. These results provide a solid baseline for comparing Java with Python and C.

6 Python

6.1 Methodology: Python Matrix Multiplication Benchmarking

Python's matrix multiplication benchmarking was conducted with measurements for both execution time and memory usage across varying matrix sizes.

6.2 Prerequisites

Install the required Python libraries before running the experiments:

- `psutil`: Used to measure memory usage.
- `matplotlib`: Used for generating graphical plots.

Installation commands:

```
pip install psutil
pip install matplotlib
```

6.3 Code Location

The full Python code for matrix multiplication benchmarking is available on GitHub: `https://github.com/daliaxx/BigData`

6.4 Experimental Setup

- **Matrix Sizes Tested**: 10x10, 100x100, 512x512, 1024x1024
- **Performance Metrics**:
  - **Execution Time**: Measured using `time.time()`, capturing the time taken for the matrix multiplication.
  - **Memory Usage**: Measured using the `psutil` library.

6.5 Results

Matrix Size	Execution Time (s)	Memory Usage (MB)
10x10	0.000000	0.00 MB
100x100	0.066934	0.57 MB
512x512	16.711976	2.85 MB
1024x1024	177.793322	2.80 MB

Table 2: Python Benchmarking Results

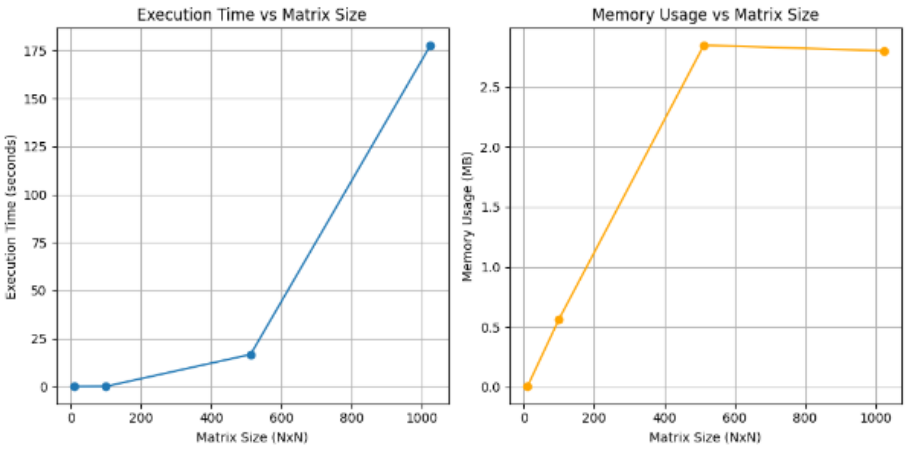

Figure 1: Execution Time vs Matrix Size and Memory Usage vs Matrix Size.

7 Conclusion

Python's benchmarking results demonstrate that both execution time and memory usage increase with matrix size, consistent with the $O(n^3)$ complexity of matrix multiplication.

8 C

8.1 Prerequisites

Before running the experiment, ensure that the necessary performance monitoring tools are installed:

- `sudo apt-get update`
- `sudo apt-get install gcc linux-tools-common linux-tools-generic linux-tools-`$(uname -r)$

8.2 Code Location

The full C code for the matrix multiplication benchmarking is available on GitHub: `https://github.com/daliaxx/BigData`

8.3 Running the Experiment

First, adjust kernel settings to allow performance monitoring:

- `echo 0 | sudo tee /proc/sys/kernel/perf`$_{event_paranoid}$ Compile the program:
  - `gcc -o matrix_multiplication matrix_multiplication.c -O2`

Run the experiment using `perf` to gather CPU and memory usage data:

- `perf stat ./matrix_multiplication 10`
- `perf stat ./matrix_multiplication 100`
- `perf stat ./matrix_multiplication 512`
- `perf stat ./matrix_multiplication 1024`

8.4 Results

8.5 Conclusion

The benchmarking results in C show a significant increase in execution time and memory usage with matrix size, consistent with the $O(n^3)$ complexity of matrix multiplication. These results provide insights into C's efficiency in matrix multiplication and allow for comparison with Python and Java.

Matrix Size	Execution Time (s)	Memory Usage (Page Faults)	CPU Usage (Task Clock)
10x10	0.000002	60	2.77 msec (0.687 CPUs)
100x100	0.001022	122	3.52 msec (0.764 CPUs)
512x512	0.212873	1,606	242.51 msec (0.993 CPUs)
1024x1024	2.528887	6,222	2,705.76 msec (0.998 CPUs)

Table 3: C Benchmarking Results

9 Main Conclusion

In this study, we investigated the performance of matrix multiplication in three programming languages: Python, Java, and C. The primary challenge lies in the computational complexity of matrix multiplication, which is $O(n^3)$, meaning the time and resources required for the operation increase drastically with the size of the matrix.

To address this, we implemented matrix multiplication in each language and measured key performance metrics such as execution time and memory usage across varying matrix sizes (10x10, 100x100, 512x512, and 1024x1024). We utilized benchmarking tools like JMH for Java, psutil for Python, and perf for C to ensure accurate and consistent results.

9.1 Main Results

- **Execution Time**: As expected, execution time increased in all languages as the matrix size grew, demonstrating the impact of the $O(n^3)$ complexity. C exhibited the fastest execution times, followed by Java and Python, making C the most suitable choice for large-scale matrix multiplication tasks.
- **Memory Usage**: Python showed slightly more efficient memory usage for larger matrices due to its dynamic memory handling, but C provided the best balance between speed and resource use. Java's memory consumption grew significantly as matrix sizes increased, primarily due to the overhead introduced by the Java Virtual Machine (JVM).
- **CPU Utilization**: C showed the highest CPU utilization, efficiently using available resources, while Java's and Python's performance were more limited by their respective runtime environments.

9.2 Discussion

The results highlight the strengths and weaknesses of each language when performing matrix multiplication, especially when applied to Big Data and high-performance computing tasks. C demonstrated the best performance in terms of speed and resource management, making it ideal for environments where low-level control and optimization are necessary. Java offered a reasonable balance,

with JMH allowing precise measurements but at the cost of higher memory usage. Python, while slower, remains a more accessible option with efficient memory handling, making it a useful choice for smaller-scale matrix operations or scenarios where development speed is more critical than raw performance.

9.3 Importance of the Experiment

This experimentation is crucial because matrix multiplication is a common operation in fields such as machine learning, data analytics, and scientific computing, where the ability to efficiently handle large datasets is critical. By comparing the performance of different languages, this study provides valuable insights for developers and researchers into which tools and languages are best suited for specific tasks in the context of Big Data.

These results also serve as a foundation for future research, including exploring optimizations, parallel processing, and distributed computing techniques to further improve matrix multiplication performance in all three languages.

10 Future Work

There are several ways to improve and extend this study:

- **Faster Algorithms**: Trying out faster algorithms, like Strassen's, could give better results than the basic method.
- **Parallel Processing**: Running the multiplication on multiple CPU cores or in a distributed system could speed things up.
- **Using GPUs**: Testing matrix multiplication on GPUs could make it much faster, especially for large matrices.
- **Memory Optimization**: Finding better ways to manage memory and use the cache could improve performance, especially in C.
- **Bigger Matrices and Stronger Hardware**: Testing even larger matrices and using more powerful computers would give a better idea of how these methods perform at scale.
- **Machine Learning**: Applying these methods to real-world tasks like machine learning could show how useful the optimizations are.
- **Further CPU Insights**: Looking more closely at CPU cycles and specific optimizations for each language could uncover additional improvements.