TASK 1. Language Benchmark of matrix multiplication Big Data

Denys Kavkalo Gumeniuk

October 20, 2024

Abstract

The challenge of this paper is to compare the performance of a basic matrix multiplication algorithm across three programming languages: Python, Java, and C. Matrix multiplication is a fundamental computational task, and its efficiency can vary significantly depending on the language and environment used. The primary objective is to determine how each language performs in terms of execution time, memory usage, and computational overhead during matrix multiplication.

The experimentation involves implementing the same matrix multiplication algorithm in Python, Java, and C, and running benchmarks with varying matrix sizes. The tests capture execution times in milliseconds per operation (ms/op) and memory consumption rates in megabytes per second (MB/s) to analyze the languages' performance under comparable conditions.

Initial results show that C outperforms Python and Java in terms of execution time, especially for large matrix sizes, while Python demonstrates higher memory consumption rates. Java, while slightly slower than C, shows balanced memory usage.

In conclusion, the study highlights that lower-level languages like C are more suitable for high-performance computing tasks such as matrix multiplication, while higher-level languages provide ease of use at the cost of efficiency.

1 Background

Benchmarking is a technique used to measure and evaluate the performance of algorithms, software, or systems under various conditions. In the context of computing, benchmarking is particularly valuable when comparing the efficiency, speed, and resource usage of different implementations. For applications dealing with Big Data, where the scale and processing speed are critical, benchmarking helps identify performance bottlenecks and inefficiencies that can hinder scalability and real-time processing.

Given the varied performance characteristics of programming languages like Python, Java, and C, understanding how each language handles matrix multiplication at different scales provides valuable insights into their suitability for large-scale applications. Each language has distinct strengths and weaknesses, particularly in execution time, memory consumption, and computational overhead.

For more information and access to the source code, visit the GitHub repository:
https://github.com/DenysKavkalo/Matrix-Multiplication.

2 Problem Statement

Matrix multiplication, while fundamental to numerous computational applications, presents various challenges when scaling to larger datasets and more complex environments like those found in Big Data. The efficiency of matrix multiplication is heavily influenced by the computational resources available, the programming language used, and the optimization of the algorithm implemented. Given the critical role that matrix multiplication plays in data processing, scientific computing, and machine learning, understanding how different programming languages handle this operation becomes essential.

The problem lies in the trade-offs between execution time, memory consumption, and scalability across different programming environments. In the context of Big Data, where datasets can reach massive scales, identifying bottlenecks in computational performance becomes crucial. This paper addresses these challenges by performing a comparative analysis of matrix multiplication across Python, Java, and C. The goal is to determine which language provides the best performance under varying matrix sizes and computational loads, offering insights into their suitability for Big Data applications.

3 Methodology

The experiments will be conducted on a Windows operating system using the development environments CLion, IntelliJ, and PyCharm, corresponding to each programming language: C, Java, and Python. The performance measurements will focus on execution time and memory usage, which are critical factors in assessing the efficiency of the matrix multiplication implementations. The experiments will be performed for matrices of sizes 10, 100, and 1000.

Each experiment will include a warmup phase consisting of three preliminary runs to mitigate any initial performance fluctuations. Following the warmup, five iterations will be executed for each matrix size, allowing for a robust analysis of the performance metrics. All tests will utilize a single thread to ensure consistency across the experiments.

The data collected during these tests will provide insights into the performance characteristics of each programming language concerning matrix multiplication, facilitating a comparative analysis of their efficiency.

3.1 Java

It is strongly recommended to use "Oracle OpenJDK 23.0.1" for the project structure in IntelliJ, specifically in the Project settings. Additionally, Maven should also be installed.

The project is opened in IntelliJ within the `Matrix-Multiplication-master` folder. A terminal is opened in IntelliJ, and the command `mvn clean package` is executed to generate the JAR file. Subsequently, the command:

```
java -jar target/matrix-1.0-SNAPSHOT.jar org.example.MatrixMultBenchmarking -prof gc
-rf csv -rff java_results.csv
```

is executed, which will save the results in `java_results.csv` located in the `target` folder. It will take a while to execute, around 25 minutes, so wait patiently.

3.2 C

Upon opening the project in the `Matrix-Multiplication-c` folder within CLion, an Open Project Wizard will appear, to which "OK" should be selected. Subsequently, the build process is initiated by clicking on the hammer icon in the program window. In a new terminal within CLion, the command `cd .\cmake-build-debug\` is entered, followed by executing:

```
.\matrix_mult_benchmark.exe > c_results.csv
```

to save the results in the file `c_results.csv` that will be generated in the `cmake-build-debug` folder.

3.3 Python

It is highly recommended to use a virtual environment with Python 3.9 in PyCharm before proceeding.

Once you are done, it is necessary to run `pip install -r requirements.txt` in the terminal to install the required dependencies. Afterward, `pytest benchmark_matrix_mult.py` should be executed. The results will be saved in `python_results.csv` in the same folder as the .py files.

4 Experiments

The following section compares the results of matrix multiplication across Python, C, and Java, organized by matrix size.

4.1 Java Implementation

In the Java implementation, the **Java Microbenchmark Harness (JMH)** framework is employed to measure the performance of matrix multiplication accurately.

While JMH increases the duration of benchmarks due to multiple factors, it ultimately leads to more reliable and accurate performance measurements. The trade-off is generally worth it, especially for microbenchmarking, where precision is crucial for understanding the performance characteristics of Java code.

The following components are utilized:

1. **Benchmark Annotations**: The `@BenchmarkMode`, `@OutputTimeUnit`, and `@State` annotations define how the benchmarks are executed and the scope of the state. The benchmarking mode is set to measure average execution time, and the output time unit is specified as milliseconds.

2. **Parameterized Tests**: The `@Param` annotation allows the benchmark to be run with varying matrix sizes (10x10, 100x100, and 1000x1000).

3. **Setup Method**: The `@Setup` annotation is used to initialize the matrices with random values before each trial. This ensures that each benchmark run uses different input data.

4. **Matrix Multiplication Logic**: The multiplication logic is implemented in a separate class (`MatrixMult`), which follows a straightforward three-loop approach to compute the product of two matrices.

4.2 C Implementation

The C implementation consists of multiple source files that collectively facilitate matrix multiplication benchmarking. The key components include:

- **Benchmarking Function**: A custom benchmarking function, `benchmark_matrix_mult`, was implemented to measure the execution time and memory allocation rate for matrix multiplication. This function executes a series of warm-up iterations followed by the actual benchmark executions, calculating the average time taken.

- **Matrix Operations**: A set of functions was created to handle matrix operations, including `create_matrix`, `free_matrix`, `matrix_mult`, and `init_matrix`. These functions allow for dynamic matrix creation, initialization with random values, and the multiplication of matrices.

- **Manual Benchmarking**: Due to significant challenges encountered while attempting to integrate external benchmarking tools, a manual approach was adopted for benchmarking. This decision arose from the various errors faced during project setup, making it necessary to utilize the `clock` function from the standard library to time the execution, which provides a basic measurement of processing time.

- **CMake Configuration**: The `CMakeLists.txt` file was created to manage the build process, ensuring that the appropriate compiler settings and source files were included for successful compilation of the benchmarking executable.

- **Warmup Iterations**: The function `benchmark_matrix_mult` incorporates a loop to execute a specified number of warmup iterations before the actual benchmarking begins. This is achieved through a `for` loop that runs for `num_warmups` times. During these iterations, the matrix multiplication operation is performed, allowing the system to stabilize and potentially optimize resource allocation.

- **Execution Iterations**: Following the warmup phase, the function then conducts a series of benchmark executions. This is implemented with another `for` loop, which iterates `num_executions` times. Within each iteration, the execution time for the matrix multiplication is measured using the `clock` function. The time taken for each execution is accumulated to compute the average execution time at the end of the loop.

- **Memory Allocation Rate Calculation**: During the execution iterations, the total allocated memory is tracked. This is done by calculating the size of the matrices and summing it over the number of executions. The memory allocation rate is then derived from the total allocated memory divided by the total execution time, providing insights into the memory efficiency of the matrix multiplication operation.

4.3 Python Implementation

The Python implementation employs several libraries and functions to facilitate the benchmarking of matrix multiplication. The key components include:

- **Matrix Creation**: The function `create_matrix` generates an $n \times n$ matrix populated with random values using NumPy's `rand` function. This approach allows for efficient generation of large matrices.
- **Matrix Multiplication**: The `matrix_multiply` function utilizes NumPy's optimized `dot` method to perform matrix multiplication. This method provides an efficient implementation, leveraging underlying libraries for performance gains.
- **Memory Tracking**: The `tracemalloc` library is employed to track memory usage during the benchmarking process. It captures the current and peak memory usage, providing insights into memory allocation efficiency.
- **Benchmarking Framework**: The `pytest` framework is utilized for performance testing. The `test_matrix_multiply` function is decorated with `@pytest.mark.parametrize`, enabling the execution of tests for various matrix sizes (10, 100, 1000). This facilitates the comparison of performance across different scenarios.
- **Execution Timing**: The time taken to perform matrix multiplication is measured using the `time` library. The execution time is calculated before and after the multiplication operation to determine the total time taken.
- **Result Storage**: The `save_results_csv` function is responsible for storing the benchmark results in a CSV file. It includes parameters such as current memory, peak memory, memory allocation rate, and execution time per operation. This provides a structured way to log and analyze the benchmarking results.
- **Warmup Iterations**: To ensure that the benchmarking results reflect stable performance, the multiplication function is executed three times before the actual benchmarking begins. This allows the system to stabilize and prepare for accurate measurements.
- **Execution Iterations**: The actual benchmarking of the matrix multiplication is executed using the `benchmark` fixture from `pytest-benchmark`. This records the performance metrics during the execution of the multiplication operation.

4.4 Python Results

Matrix Size	Execution Time (ms/op)	Memory Allocation Rate (MB/s)
10	11.793556	0.001947
100	0.040783	0.394026
1000	0.001036	15.442764

4.5 C Results

Matrix Size	Execution Time (ms/op)	Memory Allocation Rate (MB/s)
10	0.000000	inf
100	2.600000	88.031475
1000	8239.400000	2.777894

4.6 Java Results

Matrix Size	Execution Time (ms/op)	Memory Allocation Rate (MB/s)
10	0.001424	722.646316
100	1.668254	44.877835
1000	4918.324804	1.506268

4.7 Comparison and Observations

1. **Execution Time (ms/op)**:
    - **C** demonstrates exceptional performance for the 10x10 matrix (0.000000 ms/op), which seems to be an error or an outlier.
    - **Java** shows relatively low execution times, especially for the 10x10 and 100x100 matrices, but execution time increases significantly for the 1000x1000 size.
    - **Python** presents the highest execution time compared to C and Java, particularly for larger matrices.
2. **Memory Allocation Rate (MB/s)**:
    - **C** reports an infinite memory allocation rate for the 10x10 matrix, which could suggest that memory is being handled differently or not recorded correctly.
    - **Python** shows varying memory allocation rates depending on matrix size, with high rates for small matrices and decreasing rates for larger ones.
    - **Java** displays similarly decreasing memory allocation rates, with significantly lower usage for the 1000x1000 matrix.
3. **Overall Memory Usage**:
    - **Python** exhibits significant memory usage, especially for larger matrices, suggesting that memory management optimizations could be beneficial.
    - The memory allocation characteristics in Python indicate a growing demand for memory resources with larger matrix sizes, highlighting the need for better memory management strategies.

5 Conclusion

- **Efficiency**: C is the most efficient in terms of execution time, while Python exhibits the highest memory usage.
- **Scalability**: Python's results indicate that matrix multiplication becomes computationally expensive in terms of time and memory as matrix size increases.
- **Optimization**: Investigating ways to optimize both memory usage and execution time in Python would be beneficial to make it more competitive with C and Java.

These results provide guidance in selecting a language for applications requiring intensive matrix operations and offer a starting point for further optimizations and comparisons.

5.1 Future Work

The experimentation conducted serves as a foundation for further research and optimization in matrix multiplication across the three programming languages. Future work could include:

- **Integration of Professional Benchmarking Tools**: Incorporating established benchmarking frameworks for C and Python could provide more accurate and reliable performance metrics. This would enhance the understanding of execution time, memory allocation, and overall efficiency.
- **Exploration of Optimization Techniques**: Investigating algorithmic optimizations, such as block matrix multiplication or parallel processing, could lead to significant performance improvements, particularly for larger matrices.