Parallel and Distributed Computing - First Project Report

FEUP 24/25 - 2nd Semester

Turma 07 Grupo 11

Carlos Sanches, up202107694
João Rebelo, up202107209
Luís Ganço, up202004196

Project Theme: Performance Comparison of Single-Core and Multi-Core Systems

March 21, 2025

Contents

1 Introduction

In modern computing, the performance of processors is heavily influenced by the memory hierarchy, particularly when handling large datasets. Matrix multiplication, a fundamental operation in many scientific and engineering applications, serves as a case study to explore these performance dynamics. This project aims to evaluate the impact of memory hierarchy on processor performance by implementing and analyzing different versions of matrix multiplication algorithms in multiple programming languages.

The project is divided into two main parts: single-core and multi-core performance evaluations. In single-core evaluation, we implement and compare matrix multiplication algorithms in C/C++ with Java. We measure processing times for matrices of varying sizes and analyze the effects of different algorithmic approaches, including a block-oriented implementation. The Performance API (PAPI) is utilized to collect relevant performance metrics, such as cache misses and floating-point operations, to provide deeper insights into the behavior of the memory hierarchy.

For the multi-core evaluation, we parallelize the matrix multiplication algorithms using OpenMP and analyze their performance in terms of Flops, speedup, and efficiency. Two parallelization strategies are explored, and their results are compared to understand the trade-offs between different parallel implementations.

This report documents the implementation details, performance metrics, and analysis of the results obtained from both single-core and multi-core evaluations.

2 Algorithms Explanation

2.1 Single-Core Algorithms

2.1.1 Naive Matrix Multiplication

- Element-wise multiplication and summation: Each element of a row is multiplied by the corresponding element of a column, and all of them are summed up;
- Time complexity: $O(n^3)$, as the algorithm has three nested loops.

2.1.2 Line-by-Line Matrix Multiplication

- In this variant, the multiplication is performed by iterating over the rows of the first matrix and the columns of the second matrix;
- For each row in the first matrix, the algorithm multiplies the row by each column in the second matrix, summing the results to produce the corresponding element in the result matrix;
- This approach improves cache locality compared to the naive element-wise multiplication, as it accesses the elements of the second matrix in a more sequential manner;
- Time complexity: $O(n^3)$, as it still involves three nested loops, but with a different order of iteration.

2.1.3 Block-Oriented Matrix Multiplication

- The block-oriented approach divides the matrices into smaller submatrices (blocks) and performs multiplication on these blocks;
- The algorithm iterates over the blocks of the matrices, multiplying corresponding blocks and accumulating the results;
- This approach is particularly useful for large matrices, as it improves cache efficiency by operating on smaller blocks that fit into the cache;
- The block size should be chosen carefully to balance cache utilization and overhead. Common block sizes are 128, 256, or 512, depending on the cache size of the system;
- Time complexity: $O(n^3)$, as the overall number of operations remains the same, but the block size affects the performance due to cache effects.

2.2 Multi-Core Algorithms

The parallel implementation uses OpenMP to distribute the workload across multiple CPU cores. Two variants were implemented:

2.2.1 Outer Loop Parallelization

- The outermost loop is parallelized using OpenMP's **#pragma omp parallel for** directive;
- This approach divides the rows of the first matrix among the available threads;
- Each thread computes a portion of the result matrix by iterating over its assigned rows;
- This method is effective for large matrices, as it reduces the workload per thread and improves scalability.

2.2.2 Inner Loop Parallelization

- The innermost loop is parallelized using OpenMP's **#pragma omp parallel for** directive;
- This approach divides the columns of the second matrix among the available threads;
- Each thread computes a portion of the result matrix by iterating over its assigned columns;
- This method can improve cache utilization, as threads work on smaller portions of the matrix.

3 Performance Metrics

We decided to use the following metrics to evaluate performance:

- **Execution Time -** Measures the total time taken to complete matrix multiplication for different matrix sizes. This metric provides a straightforward way to compare the overall efficiency of the implementations in C++ and Java.
- **Data Cache Misses (DCM) -** Tracks the number of times data requested by the CPU are not found in the cache, requiring slower memory access. We analyze both L1 and L2 cache misses to better understand how memory access patterns impact performance, especially for larger matrix sizes. Specifically, we use the following PAPI events:
  - **L1 Data Cache Misses (PAPI_L1_DCM)**: Measures the number of Level 1 data cache misses.
  - **L2 Data Cache Misses (PAPI_L2_DCM)**: Measures the number of Level 2 data cache misses.
- **Floating-Point Operations (PAPI_FP_OPS) -** Measures the number of floating-point operations performed during matrix multiplication. This metric helps us evaluate the computational intensity of the algorithm and its efficiency in utilizing the CPU's floating-point units.
- **Total Instructions (PAPI_TOT_INS) -** Measures the total number of instructions executed during matrix multiplication. This metric provides insight into the overall workload and how efficiently the CPU processes the instructions.
- **GFlops -** Measures the number of floating-point operations per second, reflecting the raw computational throughput. Although the assignment description mentions MFlops, we will use GFlops for our performance evaluation, as it provides a cleaner and more intuitive representation of performance, especially for larger matrices and higher-performance systems. Since 1 GFlops equals 1000 MFlops, this change in unit does not affect the underlying metric, but simplifies the visualization of results.

This combination of metrics allowed us to assess not only how fast the algorithms run but also how efficiently they utilize the hardware, considering both computation and memory access patterns.

4 Results and Analysis

Notes:

- The following results were obtained using an AMD Ryzen 7 7730U processor and Linux as the operating system (Ubuntu 24.04);
- All the tables containing the measured metrics for each algorithm, which are referenced below, can be foundhere.

4.1 Single-Core

4.1.1 Naive Matrix Multiplication

After comparing the performance results of C++ and Java (Tables 1 & 2), it is clear that C++ is faster. To confirm this, we plot relevant graphs that compare the performance metrics of both languages, as shown below.

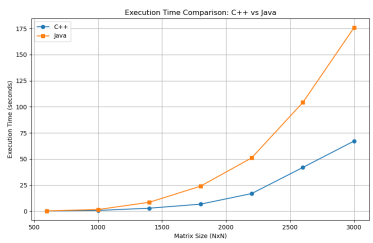

Figure 1: Execution Time Comparison

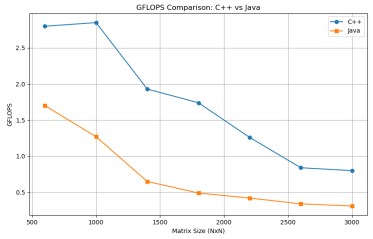

Figure 2: Line-by-Line GFlops Comparison

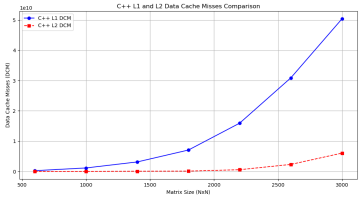

Figure 3: C++ DCM Comparison

Figure 1 supports our initial observation, while Figures 2 and 3 provide valuable insight into the underlying factors that affect performance.

- C++ outperforms Java consistently across all matrix sizes, with higher GFlops values;
- As the matrix size grows, performance (GFlops) decreases for both languages, but the drop is sharper for Java;
- This shows that C++ handles large matrix multiplications more efficiently, likely due to better memory management and lower runtime overhead;
- L1 cache misses grow exponentially as matrix size increases, indicating that larger matrices don't fit well in the smaller, faster L1 cache;
- L2 cache misses are much lower, but they still increase gradually, highlighting that more data spills into higher cache levels;
- This pattern suggests that performance degradation for larger matrices is tied to cache inefficiencies, with more memory accesses going to slower cache layers or RAM.

4.1.2 Line-by-Line Matrix Multiplication

After analyzing the performance results of C++ and Java for line-by-line matrix multiplication, Java achieves better execution times and higher GFLOPS for smaller matrices, while C++ demonstrates more consistent performance as the matrix size grows (Tables 3 & 4). To validate these findings, we have plotted relevant graphs comparing the performance metrics of both languages, as illustrated below.

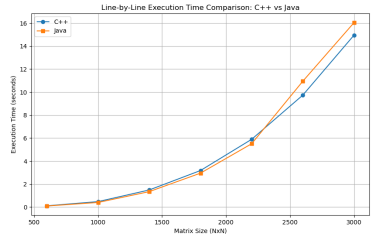

Figure 4: Line-by-Line Execution Time Comparison

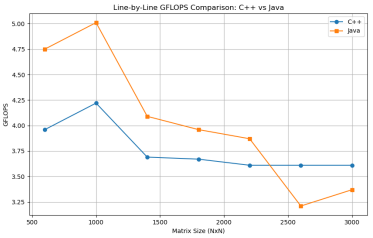

Figure 5: Line-by-Line GFlops Comparison

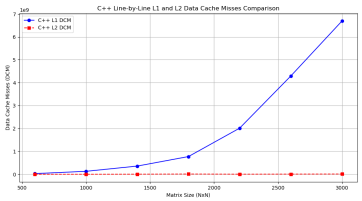

Figure 6: C++ DCM Comparison

Additionally, it was asked to register the processing time from 4096x4096 to 10240x10240 with intervals of 2048 in the C++ version (Table 5.

For line-by-line multiplication:

- C++ is more efficient for large matrix computations, maintaining consistent performance (around 3.6-3.9 GFLOPS) as matrix size increases, thanks to better memory management and lower runtime overhead.

- Java performs well for small matrices, achieving higher GFLOPS (e.g., 4.75 GFLOPS at 600x600) due to effective JIT optimization, but its performance degrades significantly as the matrix size grows, likely due to higher memory overhead and garbage collection;
- Cache performance is a key factor in execution time, as a higher number of cache misses leads to increased memory access delays. When a cache miss occurs, the processor must retrieve data from slower/lower cache layers or main memory, consuming extra clock cycles. Additionally, Java's performance is further impacted by its runtime overhead, including Just-In-Time (JIT) compilation and garbage collection.

4.1.3 Block-Oriented Matrix Multiplication

According to Table 6:

- Larger block sizes improve performance by reducing execution time and improving cache efficiency;
- Cache optimization plays a crucial role in matrix multiplication performance, as indicated by changes in L1 and L2 cache misses;
- GFLOPS values are relatively low, suggesting potential memory bottlenecks that could be addressed with further optimizations like multi-threading or better memory access patterns.

4.2 Multi-Core

Relevant plots:

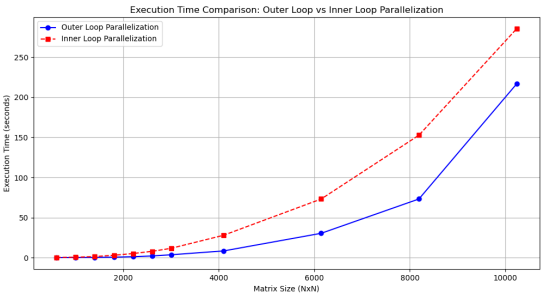

Figure 7: Parallel Execution Tine Comparison

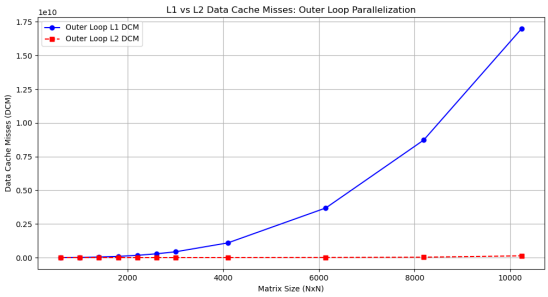

Figure 8: Outer Parallelization L1 vs L2 DCM Comparison

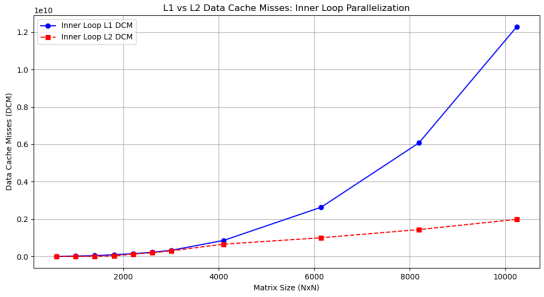

Figure 9: Inner Parallelization L1 vs L2 DCM Comparison

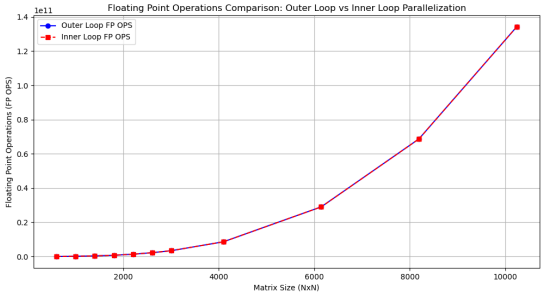

Figure 10: Parallel FP OPS Comparison

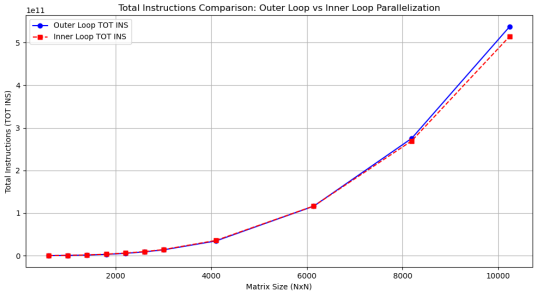

Figure 11: Parallel TOT INS Comparison

After analyzing the execution time, cache performance, floating-point operations, and instruction count across different parallelization strategies (Tables 7 & 8), we can draw the following conclusions:

Outer Loop Parallelization is the better approach for larger matrix computations due to:

- Faster execution time with lower computational overhead.
- Better cache efficiency, reducing L1 misses.
- Fewer floating-point operations and total instructions.
- Improved scalability with increasing matrix size.

Inner Loop Parallelization is less efficient for large matrices, leading to:

- Higher cache misses and inefficient memory access.
- Increased synchronization overhead and poor scalability.
- Higher instruction count and computational redundancy.

Verdict: Outer loop parallelization is the preferred method for performance and scalability.

5 Conclusions

The results highlighted the critical role of memory hierarchy and cache efficiency in computational performance. In single-core execution, C++ consistently outperformed Java for larger matrix sizes due to better memory management and lower runtime overhead. Java, on the other hand, leveraged JIT(Just In Time) optimizations to achieve better performance for smaller matrices. However, Java's efficiency dropped significantly as the matrix size increased due to higher memory overhead and garbage collection impacts.

In multi-core execution, outer loop parallelization proved to be the more efficient strategy, offering better execution time, cache utilization, and computational efficiency. On the other hand, inner loop parallelization exhibited higher cache misses and synchronization overhead, making it less suitable for large-scale computations.

Regarding block-oriented matrix multiplication, it did not scale well for larger matrices, with exponentially increasing execution times and suboptimal GFLOPS performance, primarily due to poor cache utilization.

Overall, the findings emphasized that hardware-conscious optimizations are crucial for improving matrix multiplication performance. Additionally, choosing the right parallelization strategy is essential for maximizing efficiency, particularly when working with large datasets on modern multi-core processors.