Individual Assignment 3: Parallel Matrix Multiplication Analysis

Martyna Chmielińska

December 5, 2025

Abstract

This report investigates the performance improvements gained by applying parallel computing techniques to matrix multiplication. We implemented a standard sequential algorithm and compared it with a parallel implementation utilizing Java's `ExecutorService`. The study focuses on execution time, speedup, and efficiency metrics for large matrices (1024×1024 and 2048×2048). The results demonstrate a significant reduction in computation time, achieving a speedup of approximately $4x$ on an 8-core processor.

1 Introduction

Matrix multiplication is a fundamental operation in linear algebra with a computational complexity of $O(N^3)$. As the matrix size increases, the execution time for a sequential algorithm grows cubically, making it inefficient for large datasets. The objective of this assignment is to implement a parallel version of the algorithm using Java concurrency mechanisms and analyze the performance gain.

The complete source code and the project structure for this assignment are available in the GitHub repository:

https://github.com/palantir00/IndividualAssignment3

2 Methodology

2.1 Environment

The tests were conducted on a machine with the following specifications:

- **CPU:** Apple Silicon (8 Cores)
- **Operating System:** macOS
- **Language:** Java

2.2 Implementation Details

Two versions of the algorithm were developed:

1. **Sequential Algorithm:** A classic triple-nested loop implementation where each element of the resulting matrix is computed sequentially by a single thread.

2. **Parallel Algorithm:** Implemented using `java.util.concurrent.Executors` with a fixed thread pool matching the number of available cores (8). The workload is partitioned by rows: each thread computes an entire row of the result matrix. This strategy minimizes synchronization overhead, as threads write to disjoint memory locations.

3 Results

3.1 Performance Data

The table below presents the recorded execution times, speedup, and efficiency for two different matrix sizes. Speedup (S) is calculated as $S = T_{seq}/T_{par}$, and Efficiency (E) is S/N_{cores}.

Matrix Size	Sequential (s)	Parallel (s)	Speedup	Efficiency
1024×1024	1.8969	0.4410	4.30	0.54
2048×2048	20.1225	5.0173	4.01	0.50

Table 1: Performance metrics comparison.

3.2 Resource Usage

During the parallel execution, a high utilization of all 8 CPU cores was observed. This contrasts with the sequential execution, which utilized only a single core. The memory usage also scaled with the matrix size, as expected for storing three $N \times N$ double-precision arrays.

4 Analysis and Discussion

The experimental results confirm the benefits of parallelization.

- **Speedup:** For the largest tested matrix (2048×2048), the execution time dropped from over 20 seconds to approximately 5 seconds, resulting in a speedup of roughly 4.01.

- **Efficiency:** The efficiency of the parallel execution is around $0.50 - 0.54$. While the theoretical maximum speedup on 8 cores is 8, practical performance is limited by memory bandwidth. Matrix multiplication is a memory-intensive operation; when multiple cores attempt to access the RAM simultaneously to fetch rows and columns, the memory bus becomes a bottleneck.

- **Scalability:** The speedup remains consistent (around $4x$) as the problem size increases, indicating that the implementation scales reasonably well for this hardware configuration.

5 Conclusion

We successfully implemented a parallel matrix multiplication algorithm using Java Executors. The tests showed that utilizing multi-threading on an 8-core processor can reduce computation time by approximately 75% (4x speedup) compared to the sequential approach. The results highlight the importance of parallel computing in handling computationally intensive tasks, while also revealing hardware limitations such as memory bandwidth.