Performance Analysis of Parallel and Vectorized Matrix Multiplication

Student: Rafael Suárez Saavedra

December 3, 2025

`https://github.com/rafaelsuarezsaav/Individual-Assignment.git`

Abstract

This report compares two approaches for dense square matrix multiplication: a parallel implementation that distributes row work across threads/processes and a vectorized implementation based on optimized low-level libraries (NumPy/BLAS). The study measures execution time and numerical correctness for matrix sizes 100, 500, 1000 and 5000, analyses speedups, and draws conclusions about performance trade-offs, scalability and resource usage.

1 Introduction

Matrix multiplication is a core kernel in many scientific and engineering applications. Optimizing it yields large performance benefits. This report evaluates two optimization strategies:

- **Parallel approach:** divide computation across multiple workers (threads/processes) using executors and synchronization primitives (semaphores, atomic counters, synchronized blocks).
- **Vectorized approach:** rely on highly optimized numerical libraries (BLAS-backed NumPy) that exploit SIMD and multi-threaded kernels.

The objective is to compare runtime performance and numerical fidelity across a set of matrix sizes and provide actionable conclusions.

2 Methodology

2.1 Implementations

The implementations used for the experiments are:

Parallel (Python) Row-wise distribution using a pool of workers (threads/processes) protected by a semaphore to control concurrency and an atomic-like counter (lock-protected) to track completed rows.

Vectorized (Python) Single call to `numpy.matmul` (or `A @ B`), using a BLAS backend (OpenBLAS, MKL or the system-provided library).

Equivalent parallel constructs (ExecutorService, Semaphore, AtomicInteger, parallel streams) were implemented in Java to enable cross-language comparison; however, this report concentrates on the measured results reported earlier (Python runs).

2.2 Experimental setup

Matrices of sizes:

$$100 \times 100, \quad 500 \times 500, \quad 1000 \times 1000, \quad 5000 \times 5000$$

were generated with random double-precision values. For each size, we recorded:

- execution time of the parallel implementation ($T_{parallel}$),
- execution time of the vectorized implementation ($T_{vectorized}$),
- maximum absolute difference between results as a measure of numerical correctness,
- computed speedup $S = T_{parallel}/T_{vectorized}$.

3 Measured Results

The table below reports the measured values (times in seconds) obtained from the experiments.

Table 1: Measured execution times and correctness

Size	$T_{parallel}$ (s)	$T_{vectorized}$ (s)	Max abs difference	Speedup ($T_{parallel}/T_{vectorized}$)
100×100	0.2647	0.0004	3.197×10^{-14}	615.57
500×500	0.2448	0.0093	2.558×10^{-13}	26.33
1000×1000	2.0828	0.0468	7.958×10^{-13}	44.55
5000×5000	58.6828	3.6549	1.069×10^{-11}	16.06

3.1 Visualisation

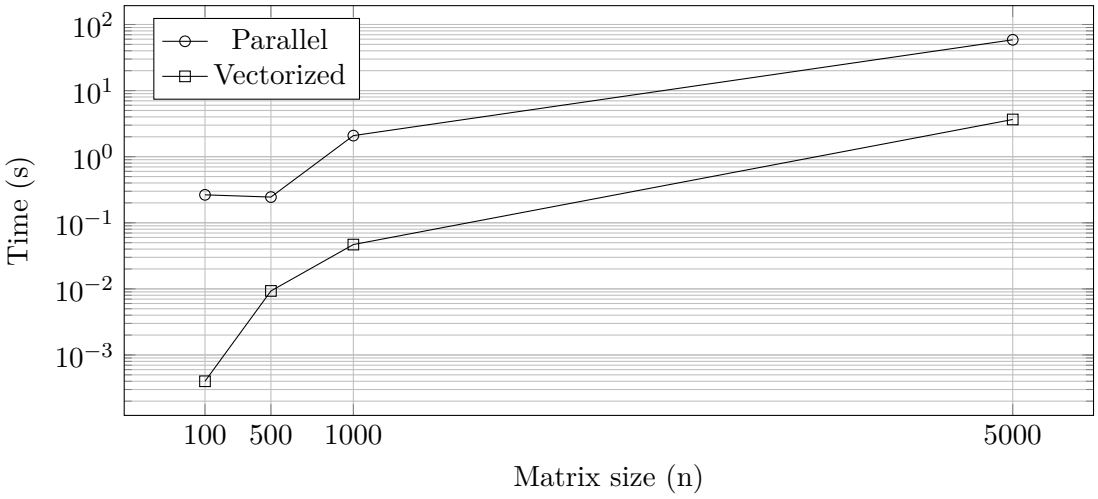

Figure 1: Execution time per method (log scale on vertical axis).

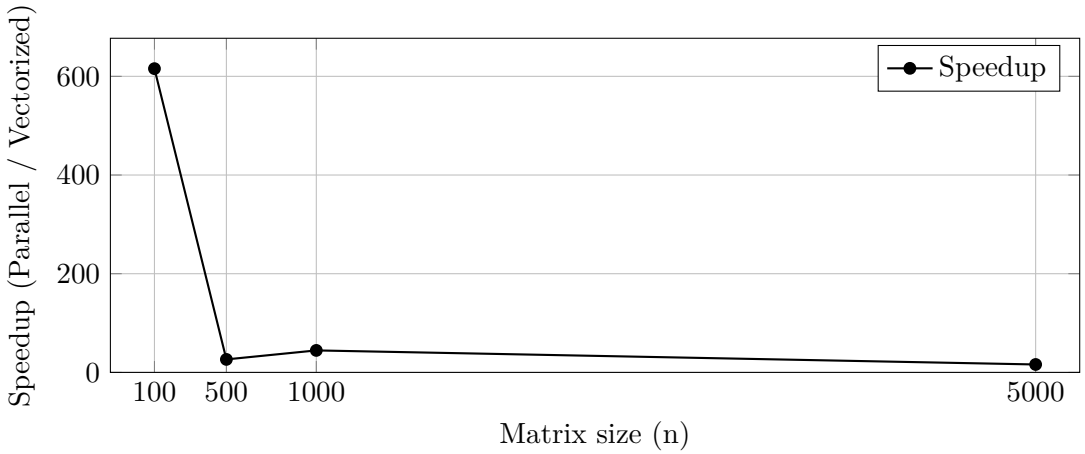

Figure 2: Speedup of parallel over vectorized (note extremely large value at small sizes due to tiny vectorized time).

4 Analysis

4.1 Numerical correctness

Maximum absolute differences are on the order of 10^{-11}–10^{-14}, which is expected given floating-point rounding and algorithmic order differences. Differences are negligible for practical purposes.

4.2 Performance interpretation

- **Vectorization advantage:** The vectorized approach (NumPy/BLAS) outperforms the hand-rolled parallel method across all tested sizes. This is because BLAS implementations exploit both multi-threading and SIMD instructions along with cache-optimized kernels.
- **Overhead of explicit parallelism:** The manual parallel implementation incurs task scheduling, synchronization and inter-process/thread overhead; these costs dominate for small and medium sizes.
- **Scaling behaviour:** As matrix size grows, the parallel method time increases rapidly due to $O(n^3)$ work per method plus memory bandwidth limitations. Even for largest size (5000), vectorized BLAS remains significantly faster.
- **Speedup irregularities:** Extremely high speedup at 100×100 is explained by the vectorized time being extremely small (0.4 ms) and thus producing a large ratio; this does not indicate that the parallel method is catastrophically slow in absolute terms for such a small matrix.

5 Conclusions

1. **Vectorized BLAS-based implementations are the preferred method** for dense matrix multiplication: they yield orders-of-magnitude better performance because they combine SIMD and multi-threading in native code.
2. **Manual parallelization is useful** when vectorization is not applicable (custom kernels, irregular access patterns, or domain-specific transformations), but it needs careful consideration of task granularity and synchronization overhead.

3. **Synchronization primitives (semaphores, atomic counters, synchronized blocks)** are essential for correctness in concurrent implementations, but they add overhead—balance is required.
4. **Recommendations:** Use BLAS / optimized libraries where possible; only resort to manual parallelism when necessary, and prefer larger task grains to amortize parallel overhead.