Stage 4 – Performance Analysis of Dense Matrix Multiplication Strategies

Agustín Darío Casebonne

December 2025

GitHub Repository

Abstract

This report presents a performance evaluation of dense matrix multiplication algorithms implemented in Java and Python. Several execution strategies are analyzed, including a naive sequential implementation, a cache-blocked algorithm, a shared-memory parallel version, and a fully distributed block-based implementation using Hazelcast. Execution times, memory usage, and correctness checks are reported and interpreted in terms of cache behavior, parallel overhead, and distributed execution costs.

1 Introduction

Dense matrix multiplication is a core kernel in scientific computing and serves as a benchmark for evaluating computational throughput, memory hierarchy utilization, and parallel scalability. Given two dense matrices A and B, the operation computes:

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj}$$

This operation has a computational complexity of $\mathcal{O}(N^3)$ and is typically compute-bound, although suboptimal memory access patterns can severely degrade performance.

The objective of this stage is to analyze how different implementation strategies affect performance:

- a naive triple-loop algorithm,
- a cache-aware blocked algorithm,
- a shared-memory parallel implementation,
- and a distributed block-based implementation using Hazelcast.

2 Implemented Algorithms

2.1 Naive Sequential Algorithm

The basic implementation follows the classical triple nested loop. While simple and easy to verify, this approach exhibits poor cache locality due to repeated accesses to matrix B by columns.

2.2 Cache-Blocked Algorithm

The blocked algorithm partitions the matrices into square sub-blocks of fixed size. This improves temporal and spatial locality by reusing blocks that fit into the CPU cache, significantly reducing memory traffic.

2.3 Parallel Shared-Memory Algorithm

The parallel version distributes rows of the result matrix across a fixed thread pool. Each thread computes independent rows, ensuring no synchronization during computation. However, for moderate matrix sizes, thread creation and scheduling overhead may outweigh the benefits.

2.4 Distributed Hazelcast Block-Based Algorithm

A fully distributed implementation was developed using Hazelcast. Matrices are partitioned into square blocks stored in distributed maps. Independent block multiplication tasks are submitted to a cluster-wide executor service. Each task multiplies two blocks and accumulates partial results into the corresponding result block.

All nodes execute the same program. The first node acts as coordinator by initializing the distributed matrices and dispatching tasks, while all nodes participate as workers executing remote tasks.

3 Experimental Setup

All benchmarks were executed on the same machine to ensure comparability. Random matrices were generated using a fixed seed. Execution time was measured using high-resolution timers. For the distributed implementation, both total wall-clock time and aggregated task computation time were recorded.

The evaluated matrix sizes were:

$$N = 300 \quad \text{(local benchmarks)}, \qquad N = 2000 \quad \text{(distributed Hazelcast)}$$

Correctness was verified by computing the Euclidean norm of the difference between result matrices.

4 Java Benchmark Results

Table 1: Java Dense Matrix Multiplication Results (300 × 300)

Algorithm	Execution Time (s)
Naive (Basic)	0.0305
Parallel	0.0478
Cache-Blocked	0.0246

Correctness verification:

- $||C_{\text{basic}} - C_{\text{blocked}}|| = 0.0$
- $||C_{\text{basic}} - C_{\text{parallel}}|| = 0.0$

5 Distributed Hazelcast Results

The distributed Hazelcast implementation was executed using a block size of 64 and matrix size 2000 × 2000.

Table 2: Hazelcast Distributed Execution Results

Metric	Value
Nodes	1–3 (local JVMs)
Total wall-clock time	≈ 3.0 s
Aggregated compute time	≈ 45.0 s
Memory usage (per node)	180–400 MB

5.1 Interpretation of Timing Metrics

The aggregated computation time corresponds to the sum of execution times of all distributed tasks. Since tasks are executed in parallel across threads and nodes, this value is expected to exceed the wall-clock execution time. Therefore, the difference between aggregated compute time and total execution time does not represent negative overhead, but rather reflects effective parallelism.

6 Python Benchmark Results

Random matrices A and B of size 300 × 300 were generated. Execution times are summarized below.

Table 3: Python Dense Matrix Multiplication Results (300 × 300)

Algorithm	Execution Time (s)
Naive (Basic)	0.0014
Parallel	0.6425
Distributed	0.0034

Correctness verification:

- $\|C_{\text{basic}} - C_{\text{parallel}}\| = 6.88 \times 10^{-12}$
- $\|C_{\text{basic}} - C_{\text{distributed}}\| = 4.59 \times 10^{-12}$

7 Discussion

The cache-blocked Java implementation achieves the best performance for local execution due to improved cache utilization. The shared-memory parallel version performs worse for moderate matrix sizes, as parallel overhead dominates computation time.

The distributed Hazelcast implementation demonstrates scalability at the architectural level. While communication and serialization introduce overhead, the design allows computation to be spread across multiple JVMs and nodes, making it suitable for significantly larger problem sizes.

In Python, naive execution is fast due to optimized low-level operations, while parallel execution suffers from the Global Interpreter Lock (GIL). The distributed Python approach remains competitive for small matrices but does not scale efficiently.

8 Conclusions

1. Cache-aware blocking significantly improves Java matrix multiplication performance.
2. Parallel execution does not guarantee speedup for small or medium-sized matrices.
3. Distributed execution introduces overhead but enables scalability beyond single-node limits.
4. Aggregated task execution time in distributed systems must be interpreted carefully.
5. Python parallelism is constrained by the GIL, while Java offers more flexibility for scalable execution.

9 Future Work

Future extensions could include:

- benchmarking larger matrix sizes on multi-node clusters,
- adaptive block size tuning based on cache hierarchy,
- fault tolerance evaluation in distributed execution,
- comparison against optimized BLAS libraries.