Distributed Execution of Matrix Multiplication

Victor Blanco

December 2025

Project repository: `https://github.com/Viblancoda/Distributed-Execution-of-Matrix-Multiplication.git`

1 Introduction

Matrix multiplication is a fundamental operation in scientific computing, machine learning, and data analytics. While its sequential implementation is straightforward, the computational and memory requirements grow rapidly with matrix size, making single-node execution impractical for very large matrices.

Distributed computing frameworks provide a solution by partitioning data and computation across multiple nodes. This project investigates the distributed execution of matrix multiplication using two different approaches:

- A distributed Python implementation simulating a MapReduce-style block multiplication.
- A Java implementation using the Hazelcast distributed in-memory data grid.

The main objectives are to analyze scalability, resource utilization, and network overhead when multiplying matrices that cannot fit into the memory of a single machine.

2 Distributed Design

2.1 Block-Based Matrix Multiplication

Both implementations rely on a block-based decomposition strategy. Given a matrix of size $N \times N$, it is divided into square blocks of size $B \times B$. Matrix multiplication is then performed as:

$$C_{i,j} = \sum_k A_{i,k} \times B_{k,j}$$

Each block multiplication can be computed independently, making it suitable for distributed execution.

2.2 Distributed Execution Model

- Each block is treated as a distributed data unit.
- Computation is parallelized across workers or cluster nodes.
- Intermediate results are either aggregated (Python) or discarded after reduction (Java) to limit memory usage.

This design closely resembles the MapReduce paradigm, where map tasks compute partial products and reduce tasks aggregate them.

3 Python Implementation

The Python implementation simulates distributed execution using block partitioning and parallel processing. The main characteristics are:

- Matrices are split into square blocks.
- Each block multiplication is executed independently.
- Resource usage (CPU, memory) and estimated network transfer are measured.
- Performance is evaluated for increasing matrix sizes.

The estimated network transfer grows proportionally to the number of blocks exchanged between workers, highlighting communication overhead as matrix size increases.

4 Java Implementation with Hazelcast

The Java solution uses Hazelcast as a distributed in-memory data grid:

- Matrix blocks are stored in distributed maps (`IMap`).
- Hazelcast automatically partitions and distributes blocks across cluster nodes.
- Multiple Hazelcast instances can be launched to form a cluster without manual configuration.

To avoid memory exhaustion, result blocks are computed but not stored permanently, simulating a reduce-and-discard strategy. This allows the execution of much larger matrices when multiple nodes are used.

5 Experimental Results

5.1 Python Results

Table 1 summarizes the results obtained with the Python distributed implementation.

Matrix Size	Time (s)	Network (MB)	CPU (%)	Memory (MB)	Blocks
1000^2	0.05	16	13.0	96.9	4
2500^2	0.71	100	13.3	224.8	25
5000^2	5.22	400	48.0	694.7	100
10000^2	36.18	1600	10.0	2561.8	400

Table 1: Python distributed matrix multiplication results

The execution time grows super-linearly with matrix size, primarily due to increased computation and communication overhead. Memory usage scales proportionally with the number of blocks.

5.2 Java (Hazelcast) Results

Table 2 shows the performance of the Java Hazelcast implementation using a single node.

Matrix Size	Time (ms)	CPU (%)	Memory (MB)
1000^2	1405	18.7	119
2500^2	15492	12.6	300
5000^2	86925	13.0	625

Table 2: Java Hazelcast results (1 node)

For very large matrices, single-node execution becomes impractical due to memory constraints. However, Hazelcast allows multiple nodes to be launched easily, distributing both memory and computation across the cluster.

6 Performance Analysis

6.1 Scalability

Both implementations show that:

- Execution time increases rapidly with matrix size.
- Communication and memory overhead become dominant factors for large matrices.

- Distributed execution is necessary once matrices exceed single-node memory capacity.

6.2 Network Overhead

The estimated network transfer grows quadratically with the number of blocks. This explains the diminishing CPU utilization observed for very large matrices, where nodes spend more time waiting for data.

6.3 Resource Utilization

CPU utilization does not scale linearly due to synchronization and communication costs. Memory usage grows predictably with the number of blocks, confirming the correctness of the block-based model.

7 Plots

Figures 1 and 2 show the execution time and memory usage as functions of matrix size for the Python implementation.

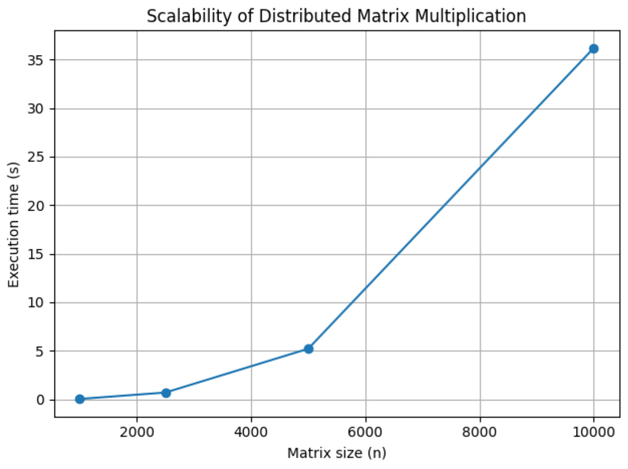

Figure 1: Execution time vs matrix size (Python)

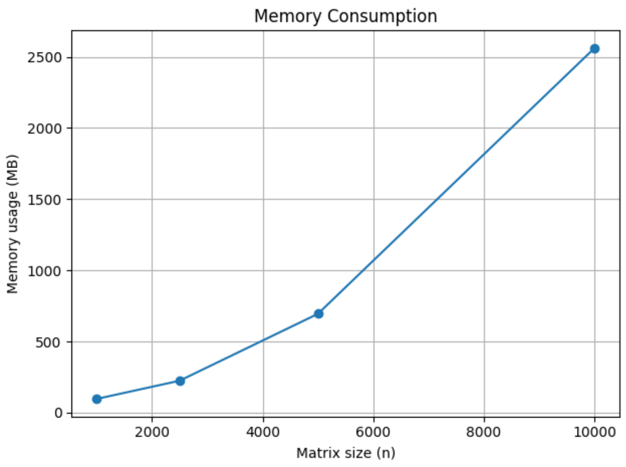

Figure 2: Memory usage vs matrix size (Python)

8 Conclusions

This project demonstrates that distributed matrix multiplication is essential for handling very large matrices that exceed the memory capacity of a single machine. Both Python and Java implementations successfully apply block-based decomposition and distributed execution.

Hazelcast provides an efficient and scalable solution in Java, particularly when multiple nodes are used. The Python implementation offers flexibility and ease of experimentation but is limited by higher communication overhead.

Overall, the results confirm that scalability in distributed systems is constrained not only by computation but also by memory availability and network communication costs.