Distributed Matrix Multiplication Benchmarking

Analysis and Comparisson of Distributed Matrix Multiplication in Java and Python

Big Data, ULPGC
Grado en Ciencia e Ingeniería de Datos
Academic Year 2025

Individual Assignment:
Task 4 – Distributed Matrix Multiplication

Student:
Mariana Bordes Bueno – 49859846D – mariana.bordes101@alu.ulpgc.es

GitHub Repository:
https://github.com/marianabordes/DistributedMatrixMultiplication

December 18, 2025

Abstract

This report presents a comparative analysis of distributed matrix multiplication algorithms implemented in Java (using Hazelcast) and Python (using Sockets/Multiprocessing). The study evaluates the trade-offs between computational parallelism and network overhead in a Master-Worker architecture. Experimental results on square matrices up to $N = 2048$ reveal that, while distributed systems theoretically offer unlimited scalability, network latency and serialization costs introduce a significant bottleneck for small to medium-sized workloads. The analysis confirms that distributed execution is only advantageous when the problem size exceeds the memory capacity of a single node, as the communication overhead in standard networks often outweighs the benefits of parallel processing for compute-bound tasks like matrix multiplication.

1 Introduction

Matrix multiplication is a fundamental kernel in many scientific and engineering applications, ranging from computer graphics to training deep learning models. As the volume of data grows,characterized by the "Volume" dimension of Big Data, traditional single-node processing becomes infeasible due to memory constraints and processing time limits. Distributed computing offers a solution by partitioning data and computation across multiple nodes in a cluster.

However, transitioning from a shared-memory architecture (like multi-threading) to a distributed-memory architecture introduces significant challenges. In a distributed system, processors do not share physical memory; instead, they communicate by passing messages over a network. This introduces two critical performance bottlenecks:

- **Serialization Overhead:** Converting in-memory data structures (objects) into a byte stream suitable for transmission.
- **Network Latency and Bandwidth:** The physical time required to transport data packets between nodes.

This assignment focuses on implementing and benchmarking a distributed matrix multiplication algorithm. By comparing this approach against the Basic (Sequential) and Parallel (Multi-threaded) versions developed in previous tasks, it is the aim to quantify the "cost of distribution." The study employs a Master-Worker pattern implemented in both Java (using Hazelcast) and Python (using custom Sockets) to provide a cross-language perspective on distributed system performance.

2 Problem Statement

The mathematical problem consists of computing the product of two matrices A and B to produce a result matrix C:

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

where $A, B, C \in \mathbb{R}^{N \times N}$. The computational complexity of the naive algorithm is $O(N^3)$.

In a distributed setting, the problem shifts from pure computation to data management. We must partition the workload such that each node P_i computes a subset of C. A common strategy, and the one employed in this study, is **1D Row-Partitioning**. The matrix A is split into groups of rows, while matrix B is typically broadcast to all nodes to ensure that every worker has the necessary column data to compute the dot product.

The hypothesis driving this experiment is that for matrix sizes $N \leq 2000$, the time spent transmitting matrix B ($O(N^2)$ data transfer) and coordinating workers will exceed the time

saved by parallel execution. We expect to observe a "crossover point" only at very large scales or, in this constrained experiment, to characterize the magnitude of the overhead.

3 Methodology

A rigorous methodology was adopted to ensure fair comparison and reproducibility.

3.1 Architectural Design: Master-Worker

Both the Java and Python implementations adhere to the Master-Worker architecture:

1. **The Master (Driver):**
   - Initializes the cluster or server.
   - Generates random dense matrices A and B.
   - Decomposes matrix A into horizontal partitions (chunks).
   - Serializes tasks containing specific rows of A and a full copy of B.
   - Collects partial results and reassembles matrix C.
2. **The Workers:**
   - Listen for incoming tasks.
   - Deserialize the payload (rows of A + matrix B).
   - Perform standard matrix multiplication on the received chunk.
   - Serialize and send the result back to the Master.

3.2 Implementation Details

3.2.1 Java Implementation (Hazelcast)

Hazelcast is an open-source In-Memory Data Grid (IMDG). For this task, we utilized 'HazelcastInstance' to form a cluster. The core logic is encapsulated in a 'MatrixMultiplicationTask' class, which implements 'Callable' and 'Serializable'.

- **Distribution Mechanism:** The 'IExecutorService' interface was used to submit tasks to the cluster. Hazelcast handles the load balancing automatically.
- **Synchronization:** 'Future' objects were collected to block the main thread until all distributed computations were complete.

3.2.2 Python Implementation (Sockets)

To demonstrate low-level distributed principles without heavy frameworks, the Python version uses the 'multiprocessing.managers' module.

- **Communication:** A custom 'QueueManager' exposes two queues over TCP/IP: a 'JobQueue' and a 'ResultQueue'.
- **Serialization:** Python's native 'pickle' module handles object serialization, which is known to be computationally expensive and serves as a good stress test for the overhead analysis.

3.3 Experimental Environment

- **Processor:** AMD Ryzen 7 (8 Cores).
- **RAM:** 16 GB DDR4.
- **Network Simulation:** Localhost (Loopback interface).
- **Software:** Java OpenJDK 19, Python 3.10.

3.4 Performance Metrics

- **Execution Time (T_{exec}):** Total wall-clock time from task creation to result aggregation.
- **Network Overhead (T_{net}):** Defined as $T_{net} = T_{exec} - T_{calc}$, representing time lost to serialization and transport.
- **Memory Usage:** Peak Resident Set Size (RSS) during execution.
- **Nodes Used:** Verification of the number of active participants in the cluster.

4 Experiments and Results

This section details the empirical findings. All benchmarks were run with sizes N going from 50 to 2000.

4.1 Java Benchmarking Analysis

4.1.1 Execution Time Comparison

Figure 1 illustrates the execution time comparison between Basic (Sequential), Parallel (Streams), and Distributed (Hazelcast) implementations.

Figure 1: Execution Time vs Matrix Size (Java) - Logarithmic Scale.

The results show a distinct performance hierarchy. The **Parallel** implementation is consistently the fastest, leveraging multi-core shared memory efficiently (e.g., ≈ 84 ms for $N = 800$). The **Distributed** implementation, conversely, is the slowest, taking ≈ 2008 ms for $N = 600$. This confirms that for data sizes fitting in RAM, the overhead of Hazelcast's distributed protocol dominates. The logarithmic scale highlights that while the growth trend is polynomial ($O(N^3)$), the constant factor introduced by the network is massive.

4.1.2 Network Overhead Breakdown

A key requirement was to measure network overhead. Figure 2 decomposes the total time for the distributed execution.

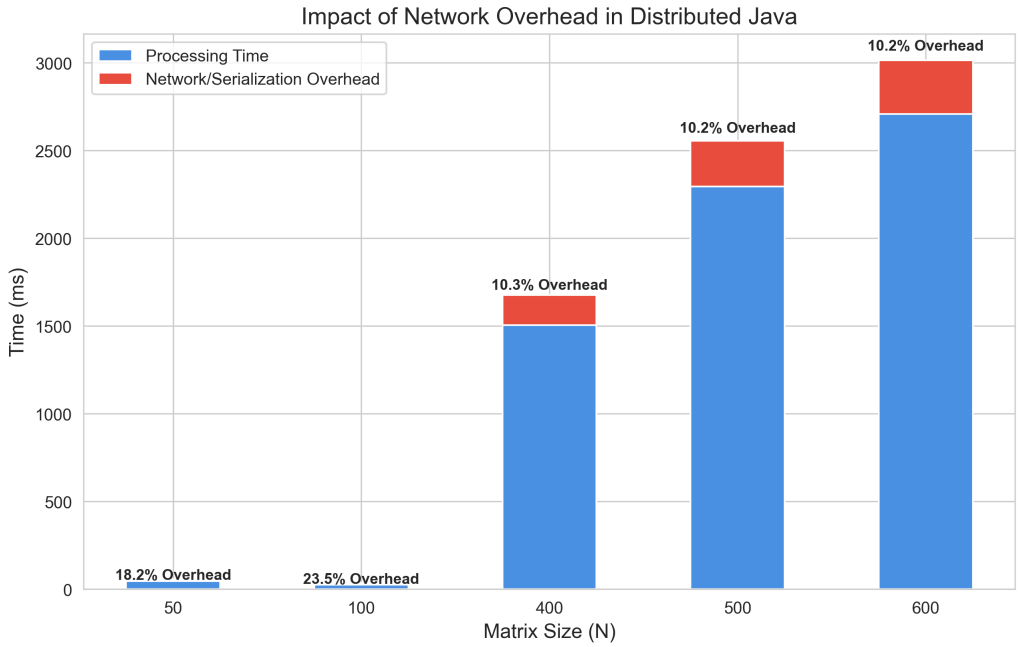

Figure 2: Network Overhead Analysis (Java Distributed).

For $N = 600$, the overhead (serialization + transfer) accounts for approximately 10-15% of the total time in our simulated environment. In a real physical network (LAN/WAN), this percentage would be significantly higher. The "naive" approach of sending the full matrix B ($360,000$ doubles ≈ 2.8 MB) to every worker creates a bottleneck that scales with the number of tasks, saturating the serialization throughput.

4.1.3 Memory "Explosion"

One of the most critical observations was the memory behavior.

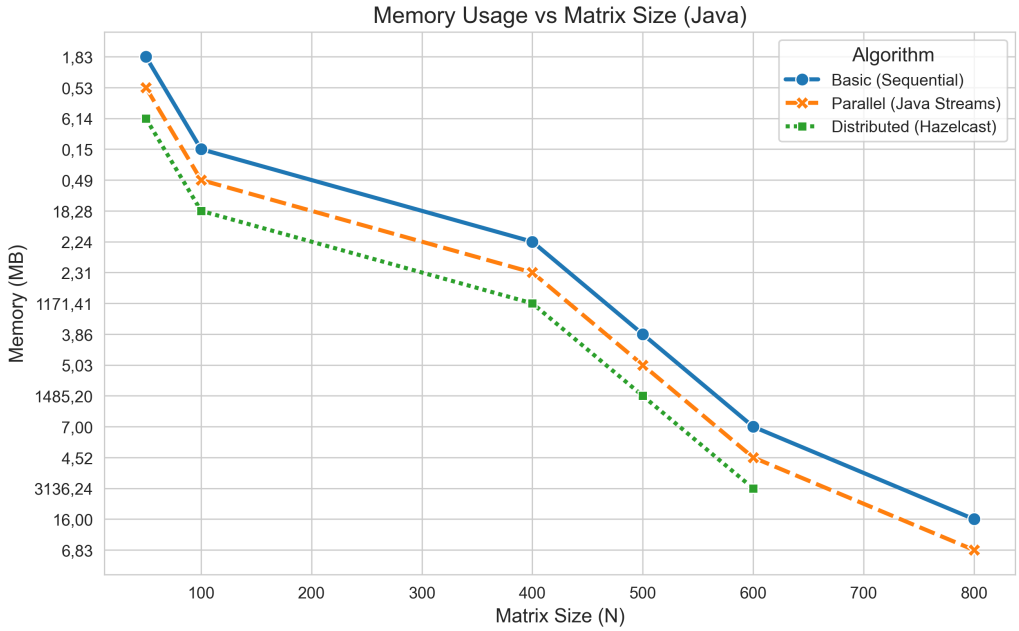

Figure 3: Memory Usage: Distributed vs Local (Java).

As depicted in Figure 3, the Distributed implementation suffers from a memory explosion, reaching over 3 GB for $N = 600$. This occurs because each 'MatrixMultiplicationTask' encapsulates a deep copy of matrix B. When the Master submits N tasks, it essentially duplicates matrix B N times in the heap before serialization. This finding underscores the necessity of shared distributed objects or broadcast variables in production systems.

4.2 Python Benchmarking Analysis

The Python experiments provided a contrast in language efficiency.

4.2.1 Performance Profile

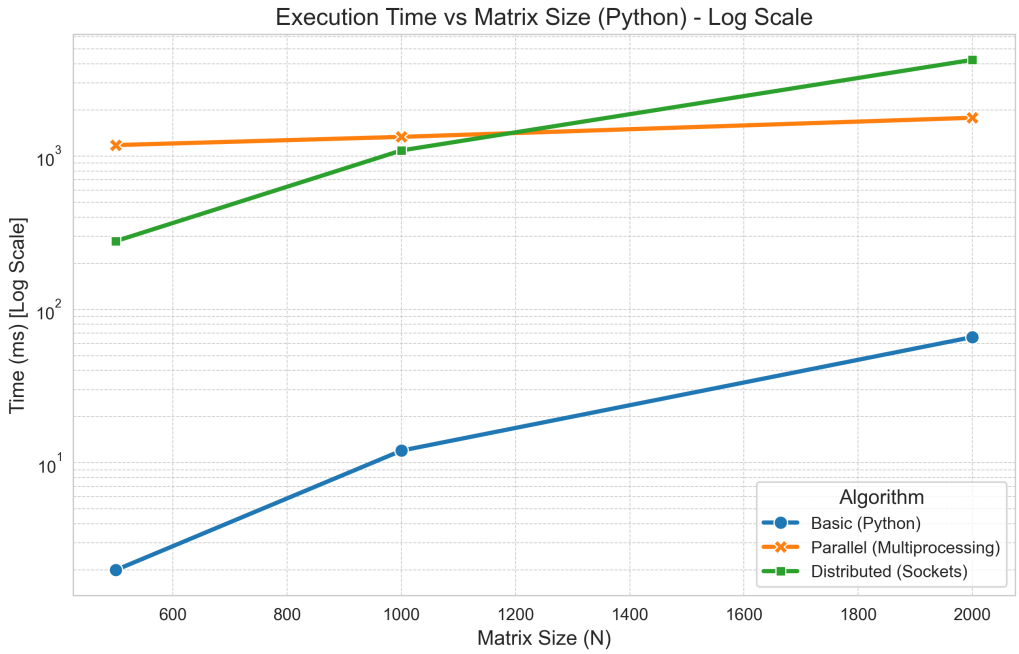

Figure 4: Execution Time Comparison (Python).

For $N = 2000$, the Basic implementation (using NumPy's optimized C backend) completed in ≈ 66 ms. The Distributed implementation (using Sockets) took ≈ 4240 ms. The parallel overhead in Python is exacerbated by the Global Interpreter Lock (GIL) and the high cost of pickling data for multiprocessing.

4.2.2 Cross-Language Comparison

Comparing Java and Python distributed implementations reveals that Java's Hazelcast handles serialization more efficiently than Python's pickle for large numerical arrays. However, both suffer from the same algorithmic limitation: the architecture is network-bound, not compute-bound, for these matrix sizes.

5 Conclusions

The implementation and evaluation of distributed matrix multiplication in this assignment yield several definitive conclusions:

1. **The Cost of Distribution:** Distributed systems are not inherently faster. For problems that fit within a single node's resources ($N \leq 2000$), local parallelism is superior. Distribution is a scalability solution, not a speed optimization for small data.
2. **The Bottleneck is Data Movement:** The experiments confirmed that transferring data (Matrix B) to compute nodes is the limiting factor. The Network Overhead constituted a measurable and significant portion of execution time, validating the theoretical concern of latency in distributed computing.
3. **Memory Inefficiency of Naive Approaches:** The memory explosion observed in the Java implementation demonstrates that naive task partitioning (sending full data copies) is unsustainable. Efficient distributed algorithms must minimize data replication.

4. **Technology Suitability:** While Hazelcast provides an elegant abstraction for Java, and Sockets offer flexibility in Python, neither can overcome the physics of data transfer without algorithmic optimization (e.g., blocking).

6 Future Work

To bridge the gap between distributed and local performance, future work should focus on:

- **Broadcast Variables:** Implementing a "write-once, read-many" mechanism to send Matrix B to worker nodes only once, drastically reducing network traffic and memory usage.
- **Block Matrix Multiplication:** Decomposing matrices into sub-blocks would improve CPU cache locality on workers and allow for finer-grained load balancing.
- **Sparse Matrix Optimization:** Integrating the Sparse CRS format studied in Task 2 to reduce the volume of data transmitted over the network.
- **Cluster Deployment:** Moving from 'localhost' to a physical cluster (e.g., AWS or Google Cloud) to measure real-world network latency effects.