Distributed Execution of Matrix Multiplication

Richard Raatz: L0NC8J5G

https://github.com/RGR-repo-cloud/ULPGC_BigData/tree/main/Task_4

Big Data (40386) - 3. year

Universidad de Las Palmas de Gran Canaria

Submission date: 2025-12-20

ULPGC Universidad de Las Palmas de Gran Canaria

Contents

ULPGC Universidad de Las Palmas de Gran Canaria

List of Figures

List of Tables

1 Introduction

Matrix multiplication, the operation of calculating $C = A \times B$, serves as the fundamental computational engine for diverse fields including machine learning and scientific simulations [1]. However, as dataset sizes scale into the "Big Data" domain, memory requirements often exceed the capacity of a single machine's Random Access Memory (RAM), necessitating a shift from in-core to distributed computing [2]. While traditional parallel computing utilizes multiple CPU cores on a single host, it remains bound by the hardware limits of that specific machine [3].To address these limitations, this project implements a distributed matrix multiplication system using Hazelcast, an In-Memory Data Grid (IMDG). The system is evaluated across three paradigms: basic serial execution, parallel multi-threaded execution, and a distributed approach. To simulate a distributed environment, a pseudo-distributed cluster was designed by running multiple Hazelcast server processes on a single machine, isolated via distinct TCP ports [4]. This setup—tested with both 2-node and 4-node configurations—allows for the analysis of network serialization overhead and resource distribution within a controlled environment. The implementation is provided in both Java and Python to compare how different language runtimes interact with the Hazelcast cluster architecture.

1.1 Research Objective

The primary objective of this assignment is to design and implement a distributed matrix multiplication system and evaluate its performance against basic (serial) and parallel (multi-threaded) versions. Apart from standard metrics like execution time and overall memory consumption the evaluation focuses on three critical metrics concerning the distributed approach:

- Scalability: Analyzing performance differences between a 2-node to a 4-node cluster configuration.
- Network Overhead: Quantifying the time cost of data distribution across different ports on the local host.
- Resource Utilization: Monitoring memory consumption per node and the data transferred

In the remainder of this paper, first, after a brief introduction to Hazelcast the design of the distributed computation system is described, then a short overview of the implementation of the suite utilized for the benchmarking is given. For the benchmarking the methodology is described first, and finally, the results are presented.

2 Overview of Hazelcast

Hazelcast is an open-source In-Memory Data Grid (IMDG) designed to provide high-speed, horizontally scalable data storage and processing. Unlike traditional databases or distributed frameworks that rely on disk I/O, Hazelcast pools the Random Access Memory (RAM) of multiple nodes to form a unified, distributed memory resource [5].

2.1 Core Architecture

The primary strength of Hazelcast lies in its partitioned storage model. Data is distributed across the cluster into partitions (shards) using a consistent hashing algorithm. In this project, the matrix blocks are stored in a Distributed Map (IMap), which acts like a standard hash map but is physically spread across the different server processes running on ports 5701, 5702, and so on.

Key features utilized in this implementation include:

- Peer-to-Peer Networking: Hazelcast nodes discover each other and form a cluster without a central "master" node, reducing bottlenecks.
- Data Locality: By using an "Entry Processor" or distributed tasks, computations can be sent to the node where the data actually resides, minimizing the need to move large matrix blocks over the network [5].
- Multi-Language Compatibility: Hazelcast provides native clients for both Java and Python, allowing the same underlying distributed data to be accessed and manipulated by different application runtimes.

3 Design of the Distributed Computation System

The distributed matrix multiplication system implemented for this project is built on Hazelcast. The architecture follows a client-server model where the Hazelcast cluster acts as a distributed storage and computation layer, while the client application orchestrates the matrix multiplication workflow.

The cluster consists of multiple Hazelcast server nodes that form a peer-to-peer network using TCP/IP discovery. Each node participates in data partitioning and storage through Hazelcast's distributed IMap data structure, which automatically shards data across nodes using consistent hashing. The client application connects to the cluster and utilizes two distributed maps: one for matrix A and one for matrix B. Data is serialized and transferred over the network using Hazelcast's built-in serialization mechanisms, ensuring that matrix blocks are accessible from any node in the cluster.

The implementation employs a block-based decomposition strategy where the result matrix C is divided into equal-sized blocks (e.g., 4×4 blocks for efficient parallelization). Each block C[i,j] is

computed independently by retrieving the necessary row blocks from matrix A and column blocks from matrix B from the distributed storage. This approach enables parallel computation of multiple blocks, as each block calculation is independent of others.

4 Architecture and Implementation Overview

The project implements matrix multiplication algorithms in both Java and Python with strict separation between production code and benchmarking infrastructure. The codebase is organized into language-specific directories (java and python), each containing two distinct modules: production/ for core algorithm implementations and benchmarks for performance measurement.

Each language implements three multiplication methods within dedicated classes: basic (sequential $O(n^3)$ triple-loop), parallel (multi-threaded using 8 threads/processes), and distributed (Hazelcast-based cluster computing). Both implementations use identical algorithmic approaches—the basic triple-loop without optimizations—ensuring performance differences reflect language and infrastructure rather than algorithmic variations. The production code is completely independent of benchmarking concerns, focusing solely on correctness and efficiency.

The benchmarking modules are entirely separate from production implementations. Each suite performs systematic measurements with three warmup iterations to eliminate JIT effects, five measured runs to capture variability, and comprehensive metrics including execution time (nanosecond precision), memory usage (ThreadMXBean for Java, tracemalloc for Python), network time, and data transfer volume. Command-line arguments (–method, –sizes) enable flexible testing of individual algorithms or combined runs, with results saved to timestamped, method-specific CSV files.

Raw benchmark results can be aggregated by a dedicated script and another module generates 16 comparative visualizations. This modular architecture enables independent execution of production code, benchmarking, aggregation, and visualization phases, facilitating reproducible experimental workflows for distributed computing performance analysis.

5 Benchmarking Methodology

The benchmarking methodology follows rigorous performance measurement practices to ensure reproducible and statistically valid results. Each test configuration undergoes three warmup iterations to eliminate JIT compilation effects and reach steady-state performance, followed by five measured runs to capture variability and enable statistical analysis (mean, min, max, standard deviation).

The benchmark suite captures four key performance dimensions: (1) execution time using high-precision timers, (2) memory usage through platform-specific APIs, (3) network time measured separately for distributed methods by timing storage, retrieval, and cleanup operations, and (4) data transfer volume calculated from matrix sizes and cluster communication. Network overhead is com-

puted as the percentage of execution time spent on network operations.
The benchmarks systematically vary matrix size and cluster configuration. Java tests six matrix sizes (64×64 to 2048×2048), while Python tests four smaller sizes (16×16 to 128×128) due to slower performance with the basic $O(n^3)$ algorithm. Distributed methods are tested with both 2-node and 4-node Hazelcast clusters to analyze scalability.
The system used for benchmarking is charakterized by the following:

- CPU: AMD Ryzen 7 4700U (8 cores and threads, 2.0 GHz - 4.1 GHz)
- RAM: 16 GB DDR4 (3200 MHz)
  - L1 Instruction Cache: 8 x 32 KB
  - L1 Data Cache: 8 x 32 KB
  - L2 Cache: 8 x 512 KB
  - L3 Cache: 8 MB
- OS: Ubuntu 22.04.5 LTS
- Java version: 17.0.17

6 Results

6.1 Java

6.1.1 Time Metrics

matrix_size	method	cluster_nodes	time_ms_mean	network_time_ms_mean
64	basic	0	0.72	0.00
128	basic	0	5.42	0.00
256	basic	0	45.61	0.00
512	basic	0	159.43	0.00
1024	basic	0	3766.92	0.00
2048	basic	0	51247.49	0.00
64	distributed	2	30.45	23.72
128	distributed	2	54.41	47.33
256	distributed	2	121.24	102.49
512	distributed	2	512.09	364.43
1024	distributed	2	3052.06	1905.46
2048	distributed	2	23647.26	14018.34
64	distributed	4	28.82	21.56
128	distributed	4	40.02	33.75
256	distributed	4	97.83	77.15
512	distributed	4	578.74	380.51
1024	distributed	4	4237.00	2557.54
2048	distributed	4	32713.39	19143.19
64	parallel	0	1.67	0.00
128	parallel	0	2.50	0.00
256	parallel	0	7.12	0.00
512	parallel	0	39.18	0.00
1024	parallel	0	511.97	0.00
2048	parallel	0	8137.15	0.00

Table 1: Time metrics across algorithms and matrix sizes for Java

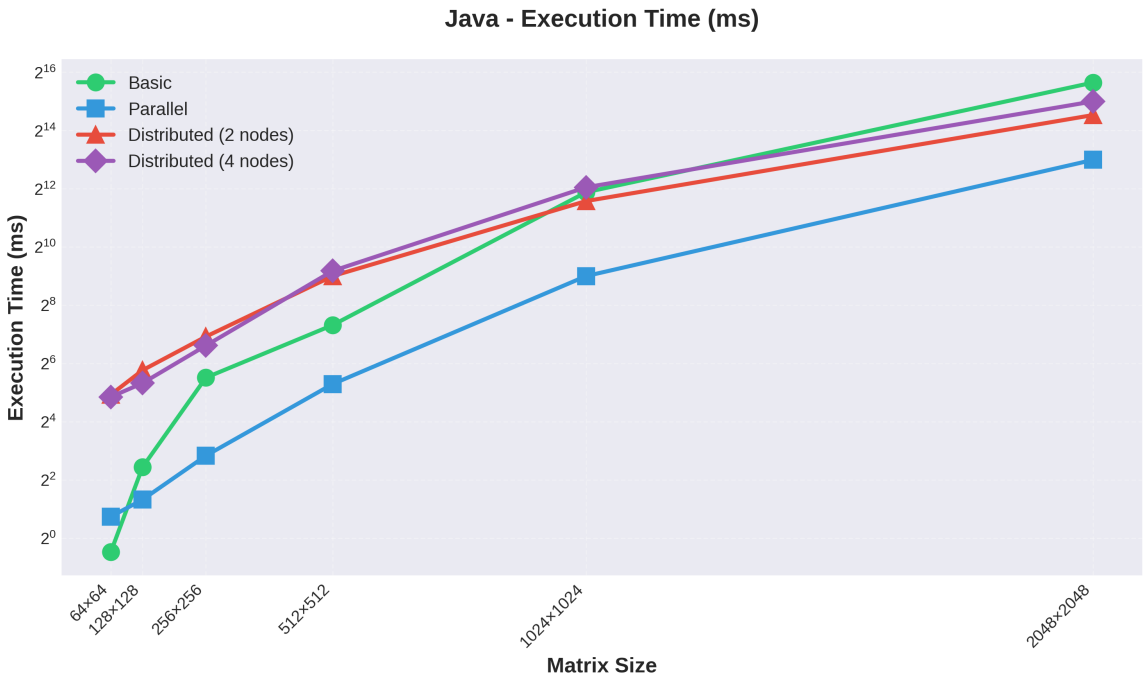

Figure 1: Execution time across algorithms and matrix sizes for Java

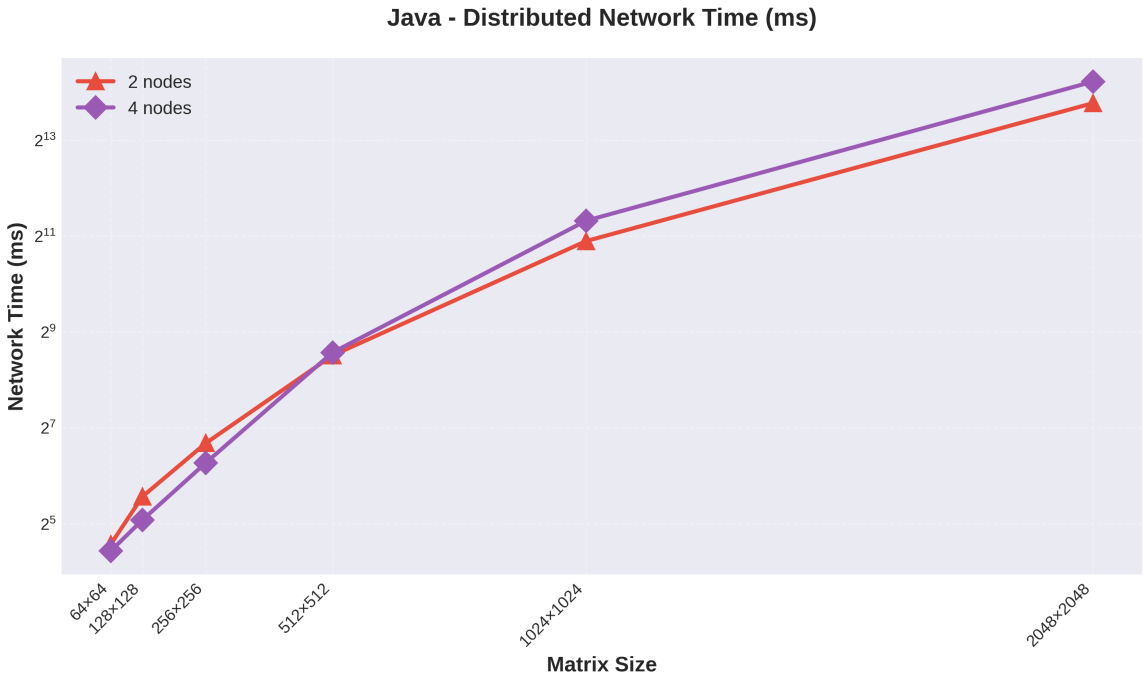

Figure 2: Network time of the distributed approach across matrix sizes for Java

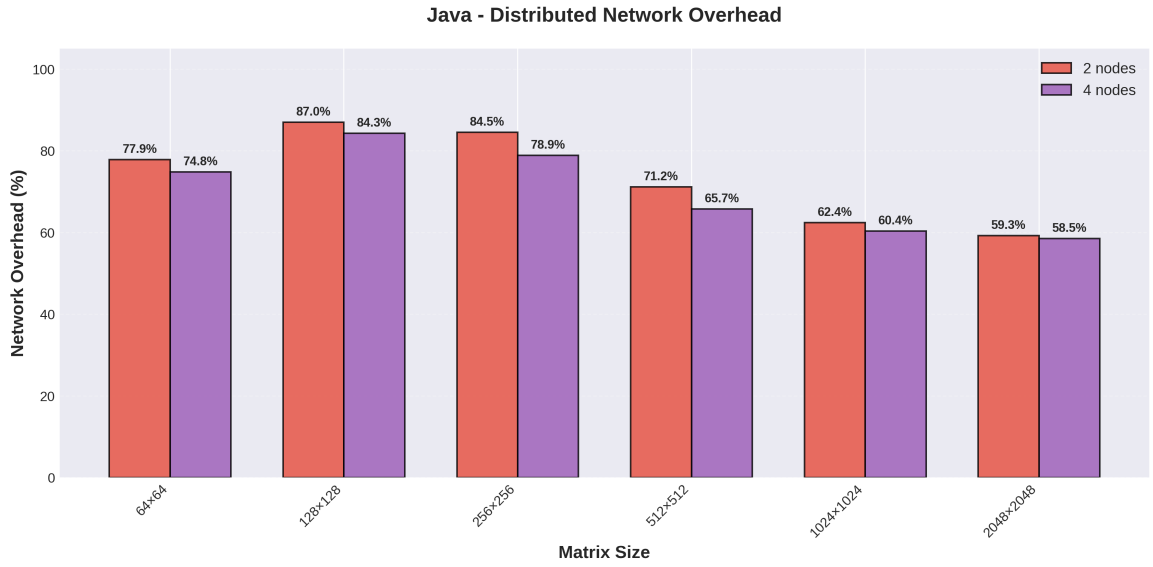

Figure 3: Network overhead of the distributed approach across matrix sizes for Java

The Java implementation demonstrates strong performance with execution times ranging from 0.72ms (64×64 basic) to 51,247ms (2048×2048 basic), exhibiting expected $O(n^3)$ scaling. The parallel method achieves consistent speedups: 2.3× (64×64), 2.2× (128×128), 6.4× (256×256), 4.1× (512×512), 7.4× (1024×1024), and 6.3× (2048×2048), demonstrating effective multi-threading on 8 cores.

The distributed approach reveals significant network overhead that dominates execution time. For 2-node clusters, network time accounts for 77.9% (64×64), 87.0% (128×128), 84.5% (256×256), 71.1% (512×512), 62.4% (1024×1024), and 59.3% (2048×2048) of total execution time. Network overhead decreases with matrix size as computation cost grows relative to fixed communication costs.

Counterintuitively, increasing from 2 to 4 nodes degrades performance for larger matrices: 512×512 (512ms → 579ms), 1024×1024 (3,052ms → 4,237ms, +38.8%), and 2048×2048 (23,647ms → 32,713ms, +38.3%). Network overhead increases to 58.5% for 2048×2048 with 4 nodes. This negative scaling occurs because all nodes share the same physical machine, creating CPU and memory bandwidth contention while increasing coordination overhead. The distributed approach never outperforms local parallel execution, highlighting that distributed computing on shared hardware introduces overhead without computational benefits, making it suitable only for deployment on truly separate physical machines or significantly larger problem sizes.

6.1.2 Resource Usage Metrics

matrix_size	method	cluster_nodes	memory_kb_mean	data_transferred_mb_mean	memory_per_node_kb
64	basic	0	120.73	0.00	0.0
128	basic	0	242.47	0.00	0.0
256	basic	0	611.42	0.00	0.0
512	basic	0	2080.13	0.00	0.0
1024	basic	0	8550.76	0.00	0.0
2048	basic	0	33746.84	0.00	0.0
64	distributed	2	877.30	0.12	32.0
128	distributed	2	3008.85	0.75	128.0
256	distributed	2	15851.78	5.00	512.0
512	distributed	2	26871.84	36.00	2048.0
1024	distributed	2	28409.94	272.00	8192.0
2048	distributed	2	67583.59	2112.00	32768.0
64	distributed	4	853.36	0.12	16.0
128	distributed	4	3011.92	0.75	64.0
256	distributed	4	15974.67	5.00	256.0
512	distributed	4	29212.19	36.00	1024.0
1024	distributed	4	26730.16	272.00	4096.0
2048	distributed	4	93626.81	2112.00	16384.0
64	parallel	0	129.77	0.00	0.0
128	parallel	0	212.34	0.00	0.0
256	parallel	0	536.11	0.00	0.0
512	parallel	0	2230.82	0.00	0.0
1024	parallel	0	8451.30	0.00	0.0
2048	parallel	0	33782.23	0.00	0.0

Table 2: Resource usage metrics across algorithms and matrix sizes for Java

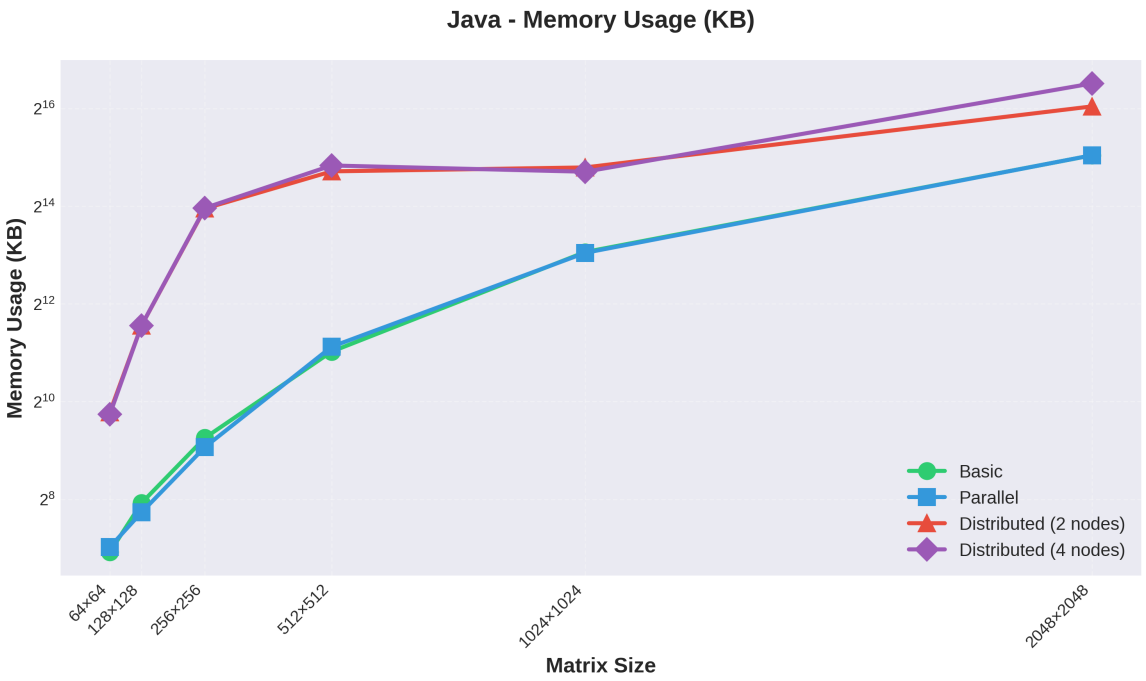

Figure 4: Memory usage across algorithms and across matrix sizes for Java

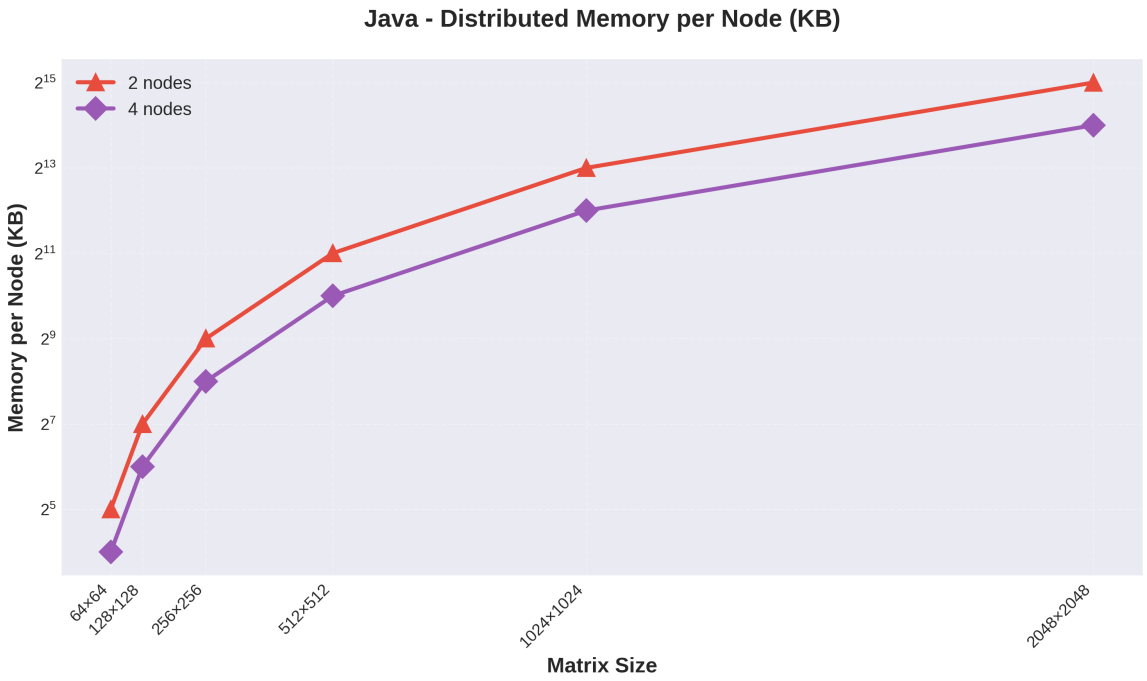

Figure 5: Memory usage per node of the distributed approach across matrix sizes for Java

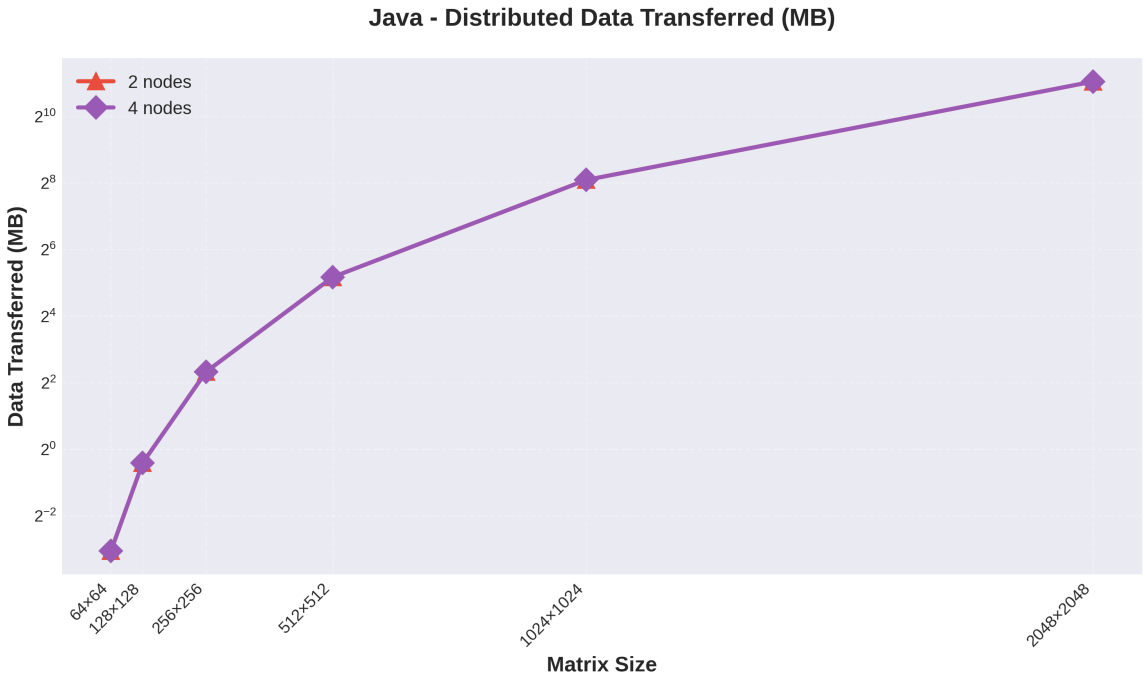

Figure 6: Data tranferred for the distributed approach across matrix sizes for Java

The Java implementation exhibits O(n²) memory scaling, ranging from 120.73 KB (64×64 basic) to 33,746.84 KB (2048×2048 basic). The parallel method shows comparable memory consumption with minimal thread management overhead (e.g., 129.77 KB vs 120.73 KB for 64×64).

The distributed approach shows significantly higher memory usage due to Hazelcast client overhead and network buffers. For 2-node clusters: 877.30 KB (64×64), 3,008.85 KB (128×128), 15,851.78 KB

(256×256), 26,871.84 KB (512×512), 28,409.94 KB (1024×1024), and 67,583.59 KB (2048×2048)—consistently 7-12× higher than basic methods. High standard deviations (22,105.65 KB for 512×512) indicate significant variability from dynamic allocation. Memory per node scales proportionally with cluster size: 32,768 KB (2048×2048) with 2 nodes halves to 16,384 KB with 4 nodes, demonstrating effective data partitioning.

Data transferred grows with matrix size: 0.12 MB (64×64), 0.75 MB (128×128), 5.0 MB (256×256), 36.0 MB (512×512), 272.0 MB (1024×1024), and 2,112.0 MB (2048×2048), remaining constant across cluster sizes. However, client memory paradoxically increases with more nodes: for 2048×2048, memory grows from 67,583.59 KB (2 nodes) to 93,626.81 KB (4 nodes, +38.5%), with higher standard deviations (36,132.85 KB vs 18,953.50 KB), indicating increased coordination overhead. This confirms that scaling challenges stem from coordination complexity rather than data movement efficiency.

6.2 Python

6.2.1 Time Metrics

matrix_size	method	cluster_nodes	time_ms_mean	network_time_ms_mean
16	basic	0	102.61	0.00
32	basic	0	810.52	0.00
64	basic	0	6400.33	0.00
128	basic	0	50860.64	0.00
16	distributed	2	184.91	45.53
32	distributed	2	1220.75	139.81
64	distributed	2	9088.00	491.15
128	distributed	2	70282.58	2926.92
16	distributed	4	194.71	52.98
32	distributed	4	1232.13	148.06
64	distributed	4	9115.87	493.97
128	distributed	4	71358.44	2934.35
16	parallel	0	45.86	0.00
32	parallel	0	161.61	0.00
64	parallel	0	1082.51	0.00
128	parallel	0	8156.40	0.00

Table 3: Time metrics across algorithms and matrix sizes for Python

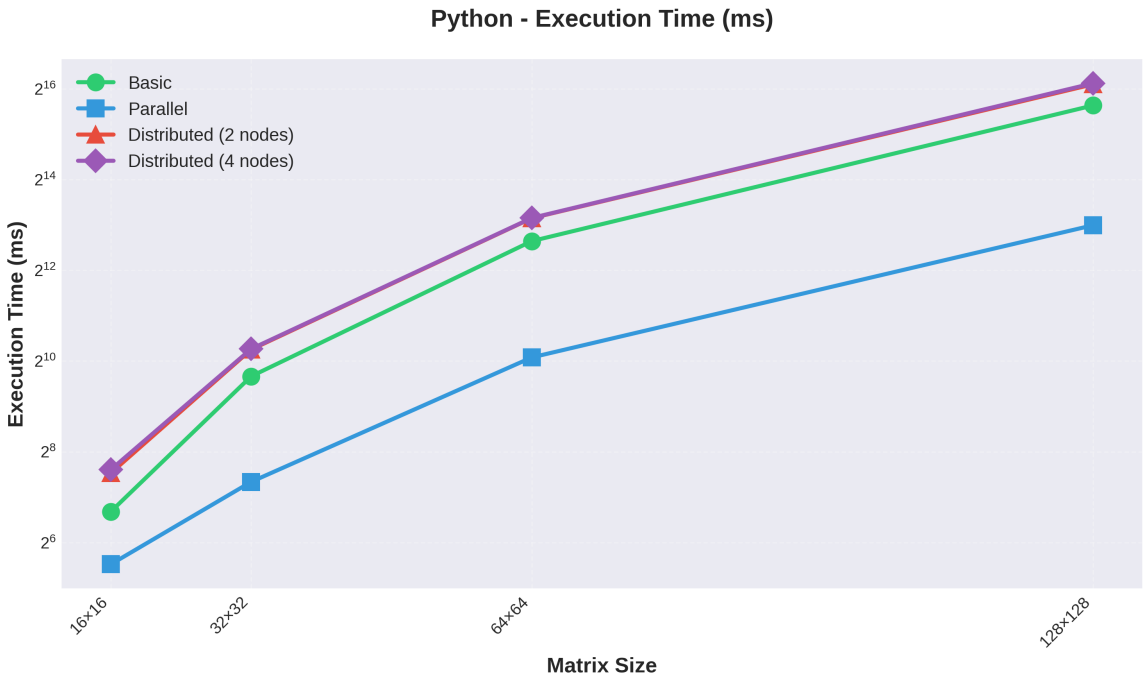

Figure 7: Execution time across algorithms and matrix sizes for Python

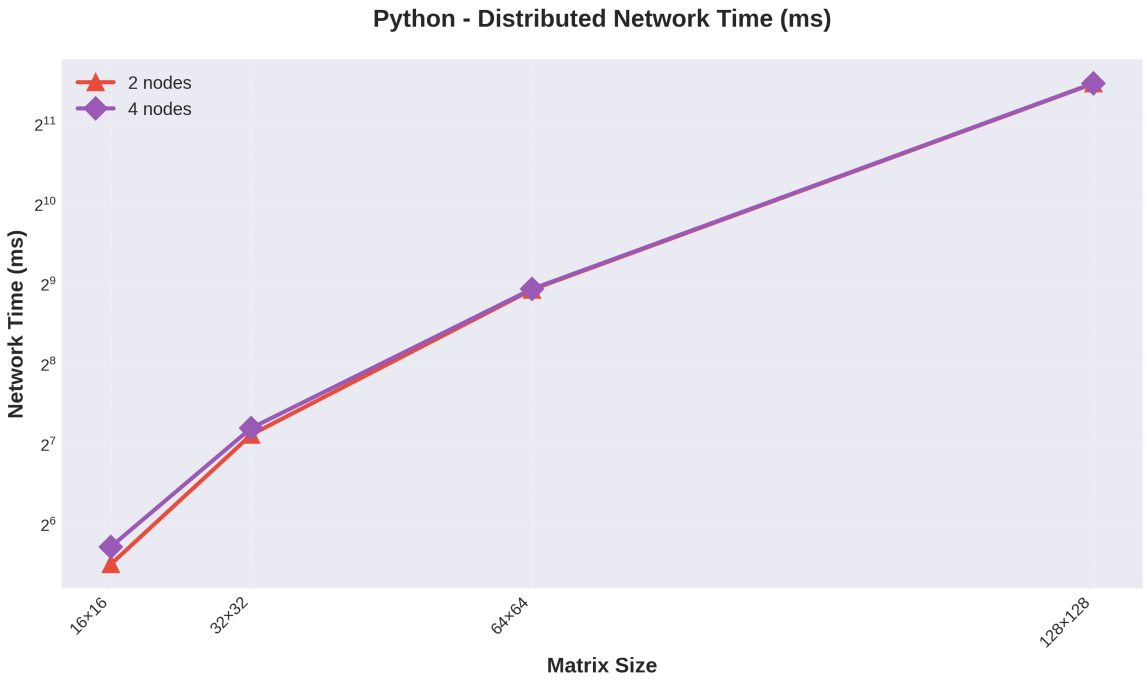

Figure 8: Network time of the distributed approach across matrix sizes for Python

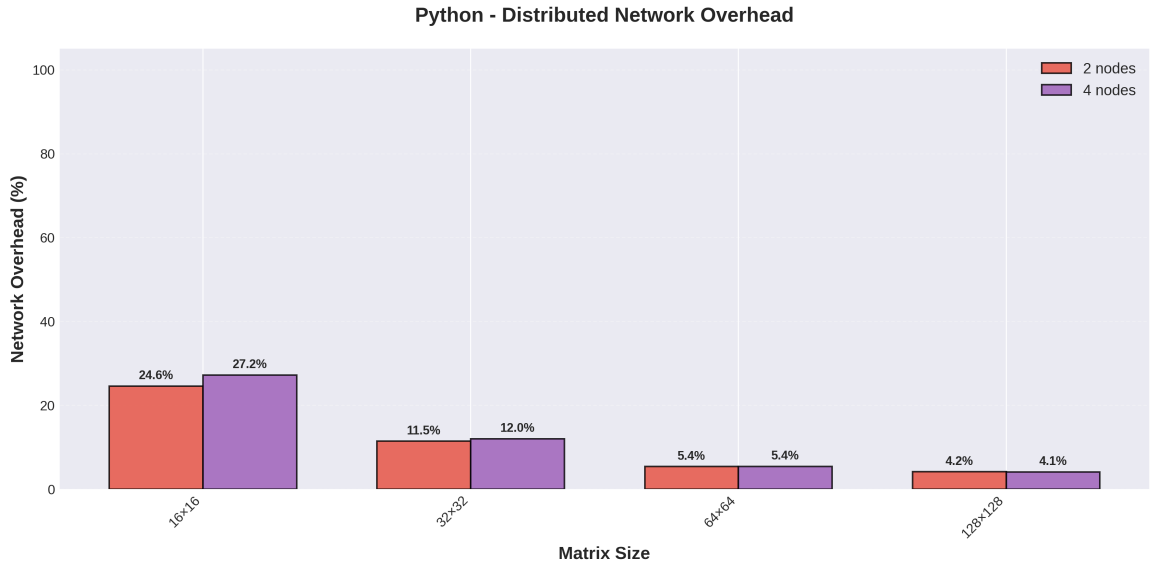

Figure 9: Network overhead of the distributed approach across matrix sizes for Python

The Python implementation shows significantly slower performance compared to Java, with execution times ranging from 102.61ms (16×16 basic) to 50,860ms (128×128 basic), reflecting Python's interpreted nature and lack of low-level optimizations. The parallel method using multiprocessing achieves notable speedups: 2.2× (16×16), 5.0× (32×32), 5.9× (64×64), and 6.2× (128×128), demonstrating effective parallelization despite Python's Global Interpreter Lock (GIL) limitations through process-based concurrency.

The distributed approach exhibits exceptionally high network overhead that severely impacts performance. For 2-node clusters, network time represents 24.6% (16×16), 11.5% (32×32), 5.4% (64×64), and 4.2% (128×128) of total execution time. Unlike Java's increasing absolute network overhead with matrix size, Python's network overhead percentage decreases as computation dominates for larger matrices, though absolute network time grows substantially (45.53ms for 16×16 to 2,926ms for 128×128).

Similar to Java, increasing from 2 to 4 nodes provides minimal benefit or degrades performance: 16×16 (184.91ms → 194.71ms, +5.3%), 32×32 (1,220ms → 1,232ms, +0.9%), 64×64 (9,088ms → 9,116ms, +0.3%), and 128×128 (70,282ms → 71,358ms, +1.5%). Network time increases with cluster size (e.g., 128×128: 2,926ms with 2 nodes → 2,934ms with 4 nodes), indicating coordination overhead outweighs parallelization benefits. The distributed approach is consistently slower than both basic and parallel methods across all matrix sizes, performing 1.8× slower than basic for 16×16 but improving to 1.4× slower for larger matrices. This demonstrates that for the tested matrix sizes on shared hardware, the network communication cost completely negates any distributed computing advantages, making local parallel execution the most efficient choice.

6.2.2 Resource Usage Metrics

matrix_size	method	cluster_nodes	memory_kb_mean	data_transferred_mb_mean	memory_per_node_kb
16	basic	0	2.45	0.00	0.0
32	basic	0	8.44	0.00	0.0
64	basic	0	32.44	0.00	0.0
128	basic	0	128.44	0.00	0.0
16	distributed	2	142.74	0.07	2.0
32	distributed	2	176.55	0.08	8.0
64	distributed	2	331.57	0.12	32.0
128	distributed	2	493.14	0.75	128.0
16	distributed	4	142.86	0.07	1.0
32	distributed	4	176.61	0.08	4.0
64	distributed	4	331.59	0.12	16.0
128	distributed	4	494.15	0.75	64.0
16	parallel	0	49.10	0.00	0.0
32	parallel	0	66.21	0.00	0.0
64	parallel	0	147.03	0.00	0.0
128	parallel	0	351.59	0.00	0.0

Table 4: Resource usage metrics across algorithms and matrix sizes for Python

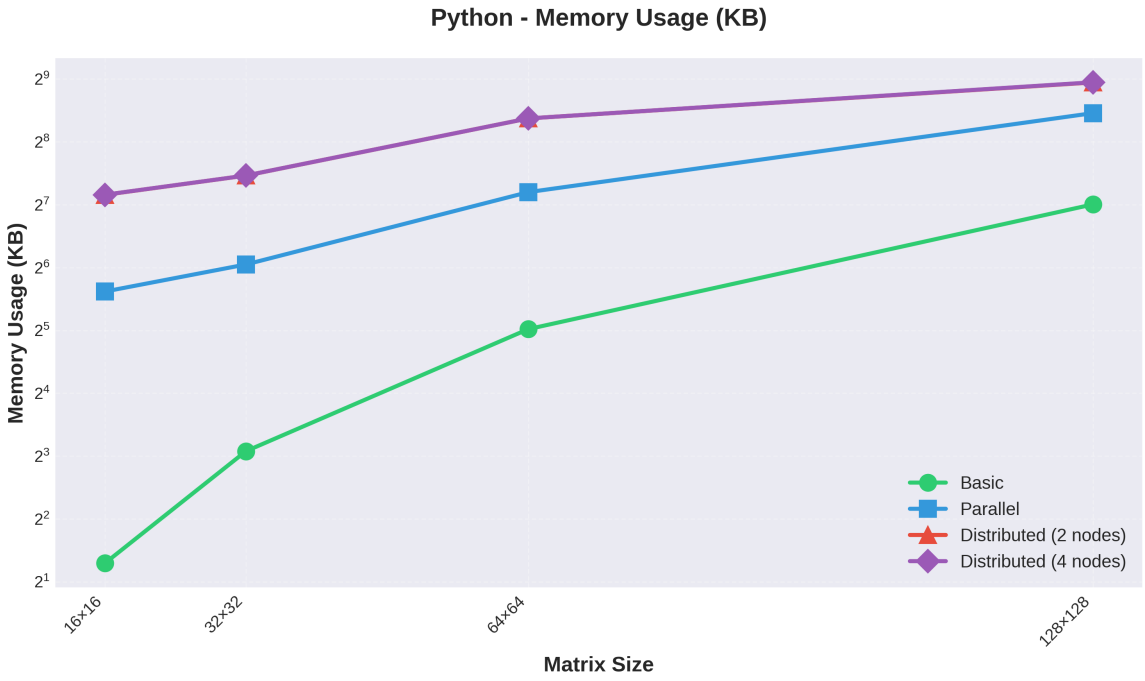

Figure 10: Memory usage across algorithms and across matrix sizes for Python

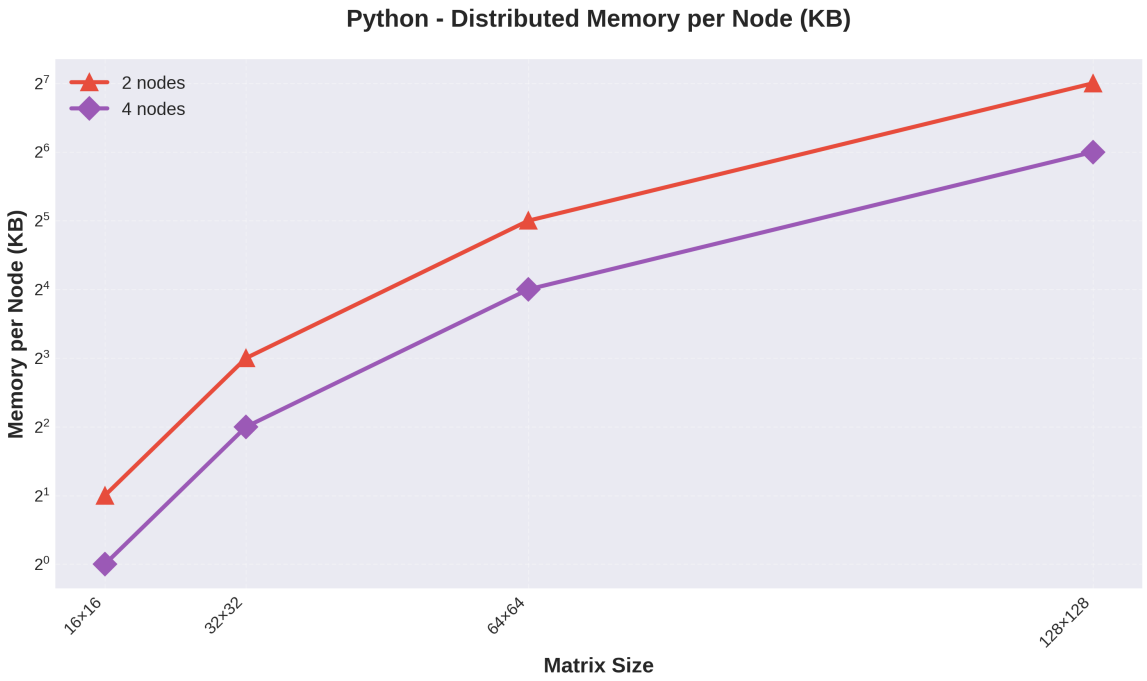

Figure 11: Memory usage per node of the distributed approach across matrix sizes for Python

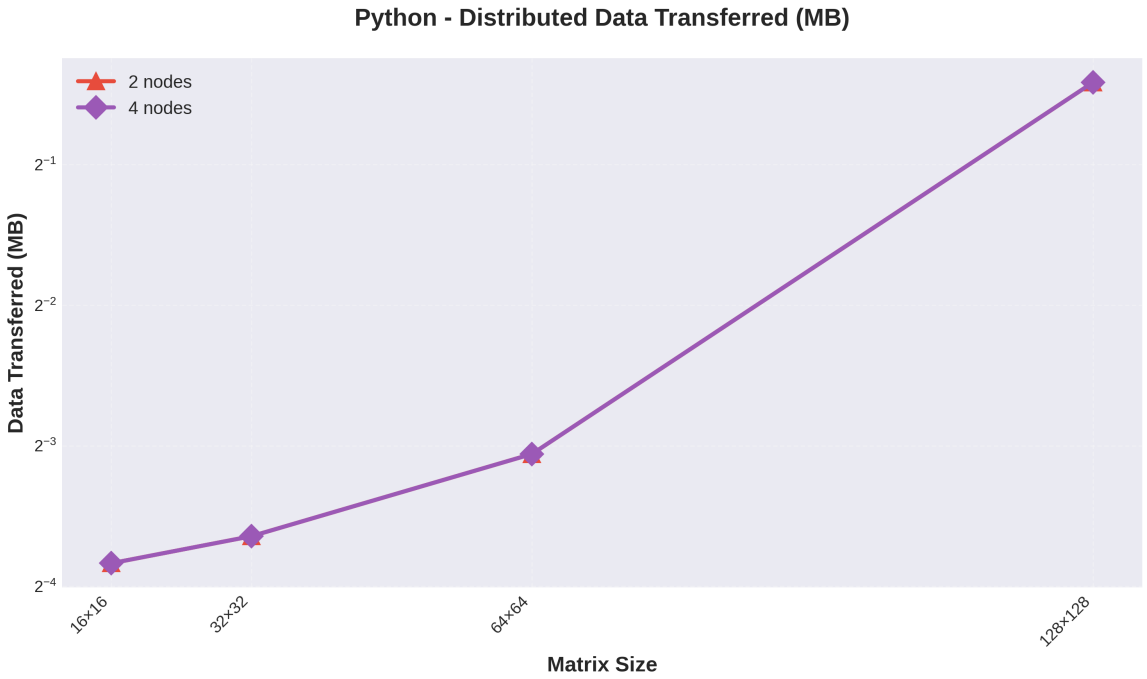

Figure 12: Data tranferred for the distributed approach across matrix sizes for Python

The Python implementation shows minimal memory usage due to tracemalloc tracking only Python object allocations, ranging from 2.45 KB (16×16 basic) to 128.44 KB (128×128 basic), exhibiting $O(n^2)$ scaling. The parallel method shows higher memory consumption (49.10 KB for 16×16, 351.59 KB for 128×128) due to multiprocessing overhead requiring separate memory spaces for each worker process, representing 20× and 2.7× increases respectively over basic methods.

The distributed approach exhibits significantly elevated memory usage compared to basic methods but lower than Java due to Python's lighter-weight Hazelcast client. For 2-node clusters: 142.74 KB (16×16), 176.55 KB (32×32), 331.57 KB (64×64), and 493.14 KB (128×128)—representing 58×, 21×, 10×, and 3.8× increases over basic methods. Standard deviations are minimal (0.10-0.19 KB), indicating consistent memory patterns across runs, contrasting sharply with Java's high variability. Memory per node scales proportionally: 2 KB (16×16), 8 KB (32×32), 32 KB (64×64), and 128 KB (128×128) with 2 nodes, halving to 1 KB, 4 KB, 16 KB, and 64 KB respectively with 4 nodes, demonstrating effective data partitioning.

Data transferred follows matrix size growth: 0.07 MB (16×16), 0.08 MB (32×32), 0.12 MB (64×64), and 0.75 MB (128×128), remaining constant across cluster configurations. Unlike Java's paradoxical memory increase, Python shows minimal impact from cluster size scaling: client memory remains nearly identical between 2 and 4 nodes (142.74 KB → 142.86 KB for 16×16, 493.14 KB → 494.15 KB for 128×128), with slightly higher standard deviations for 4 nodes (0.23-0.33 KB vs 0.10-0.19 KB) indicating marginally increased coordination variability. This stability, combined with low absolute memory consumption (100× less than Java for distributed operations), demonstrates Python's efficient client implementation, though the fixed data transfer volumes confirm that network communication rather than memory management dominates distributed performance bottlenecks.

6.3 Language Comparison

6.3.1 Time Metrics

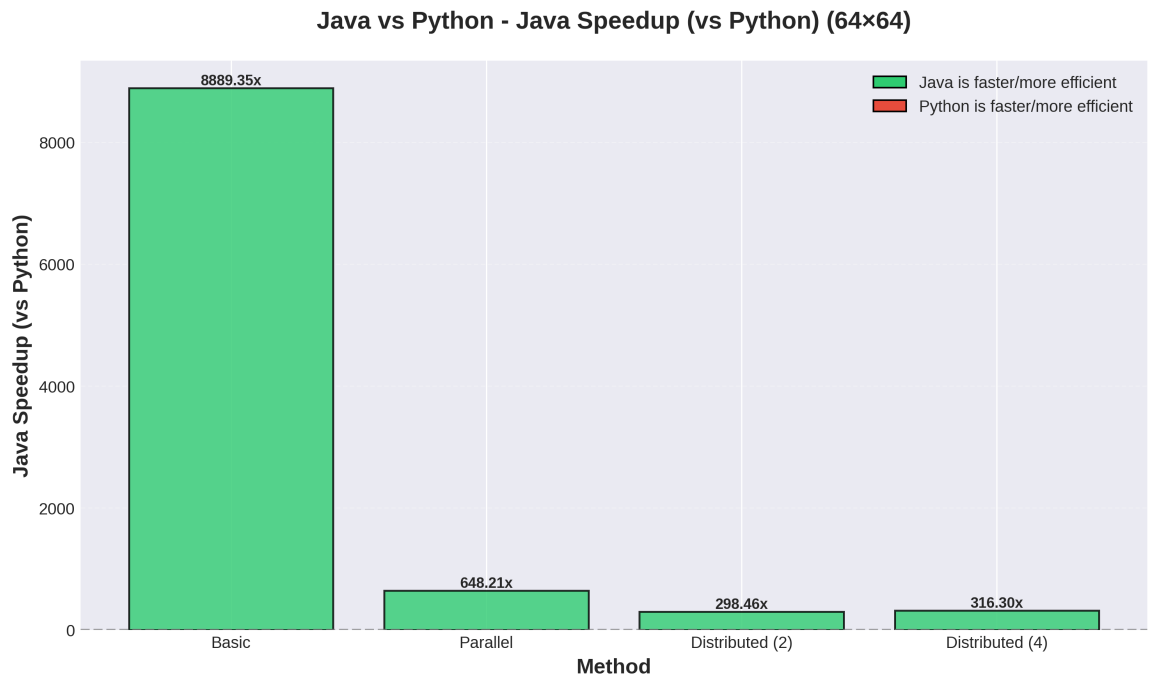

Figure 13: Speedup between Java and Python concerning execution time across algorithms for matrix size 64x64

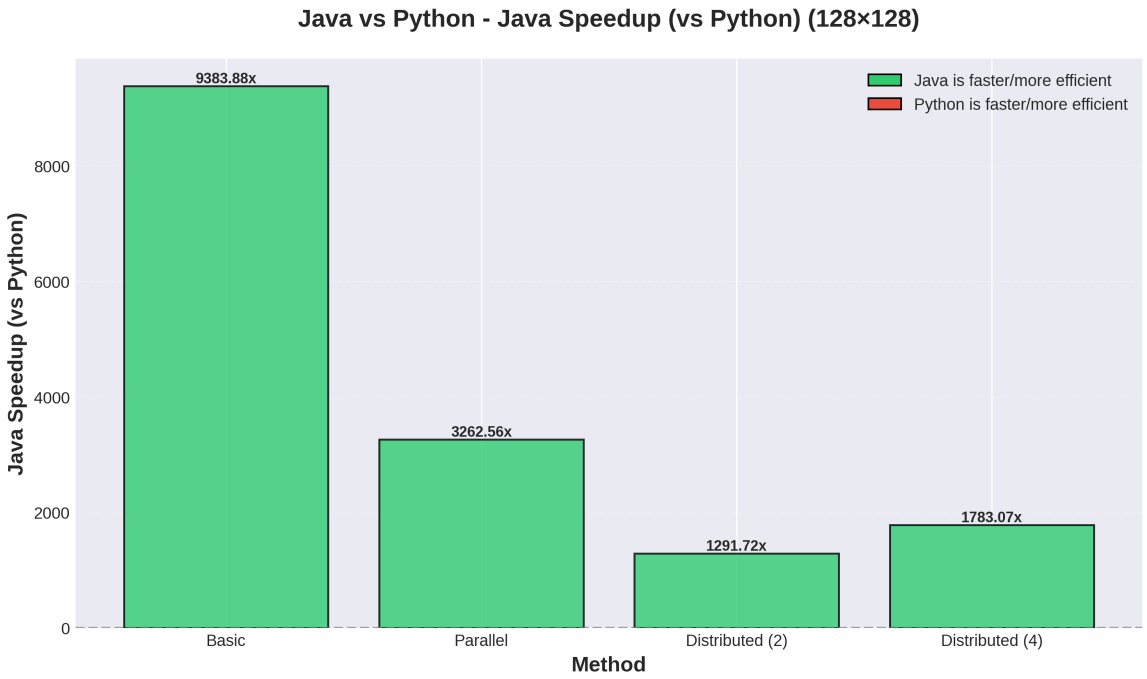

Figure 14: Speedup between Java and Python concerning execution time across algorithms for matrix size 128x128

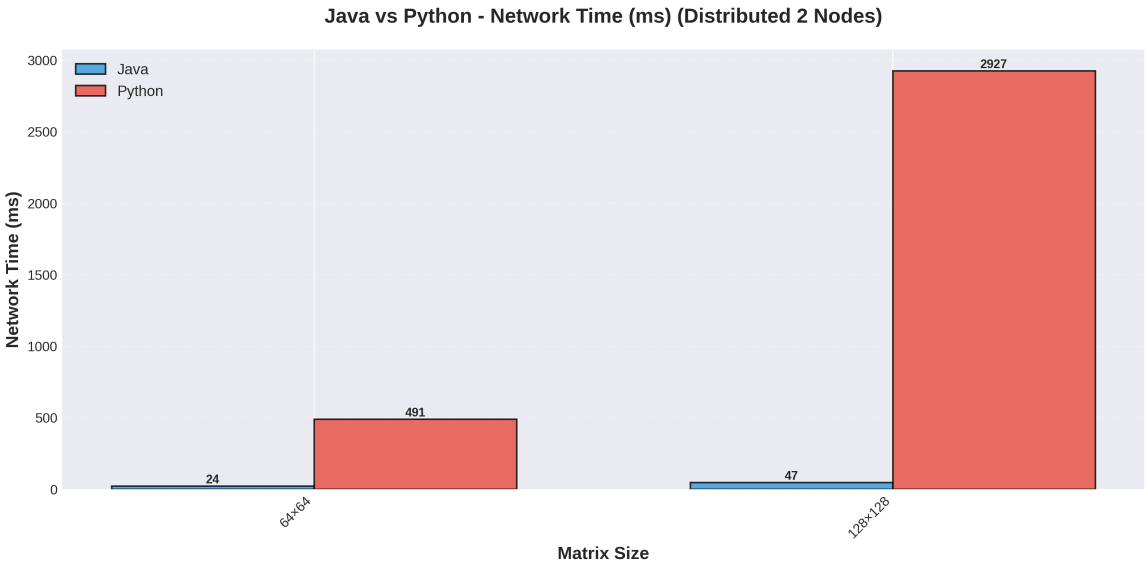

Figure 15: Network time comparison between Java and Python of the distributed approach with 2 nodes across matrix sizes

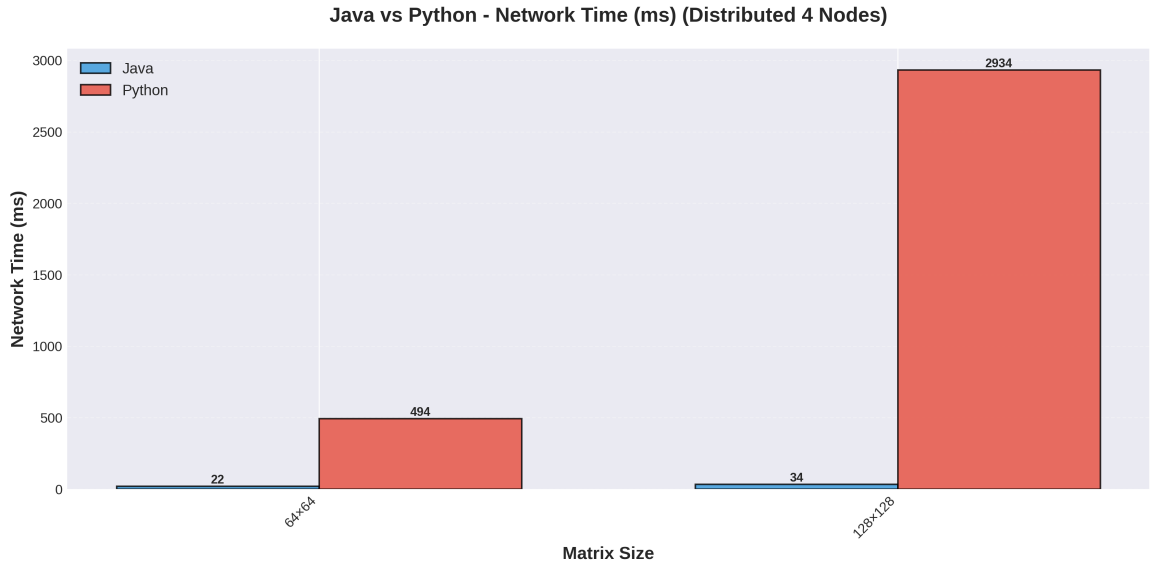

Figure 16: Network time comparison between Java and Python of the distributed approach with 4 nodes across matrix sizes

Java consistently outperforms Python in execution time across all methods and matrix sizes, with differences most pronounced in the distributed approaches. For distributed 2-node clusters, Java completes 64×64 multiplication in 30.45ms versus Python's 9,088ms; for 128×128, Java takes 54.41ms while Python requires 70,283ms. Network time also favors Java: for 2 nodes, Java's network time is 23.72ms (64×64) and 47.33ms (128×128), compared to Python's 491ms and 2,927ms respectively. With 4 nodes, both execution and network times increase for both languages, but Java remains much faster (e.g., 128×128: Java 40.02ms vs Python 71,358ms; network time: Java 33.75ms vs Python 2,934ms). Overall, Java's compiled nature and efficient Hazelcast client yield much lower execution and network times than Python, especially in distributed scenarios, where Python's overheads are magnified by serialization and process management.

6.3.2 Resource Usage Metrics

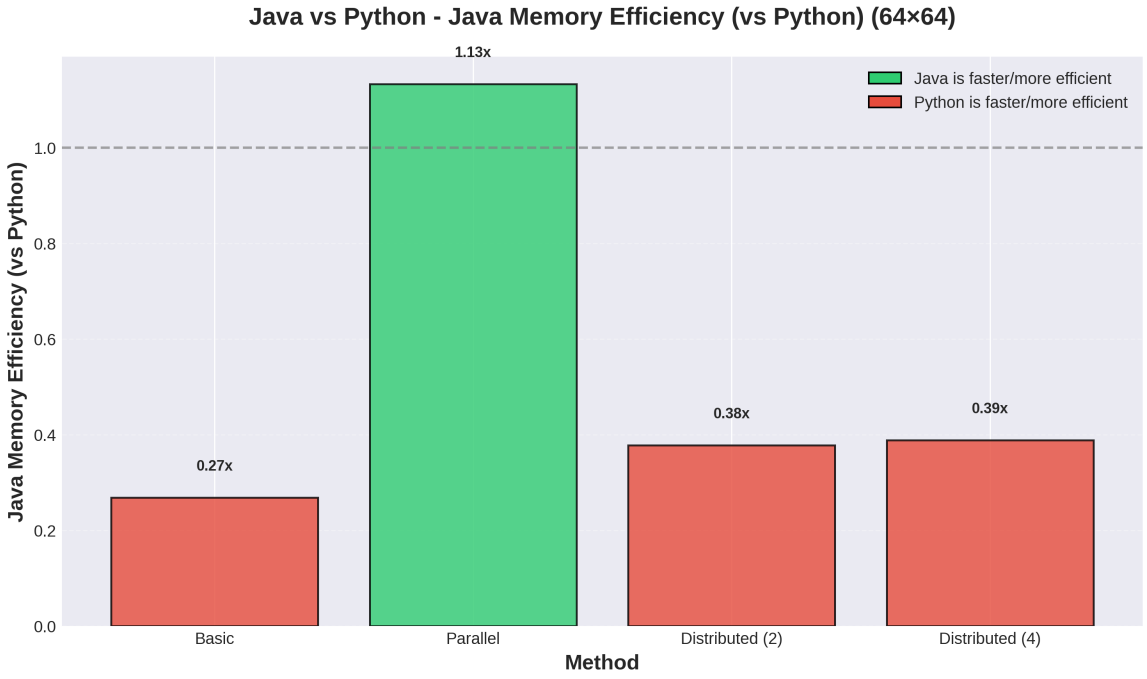

Figure 17: Memory efficiency between Java and Python across algorithms for matrix size 64x64

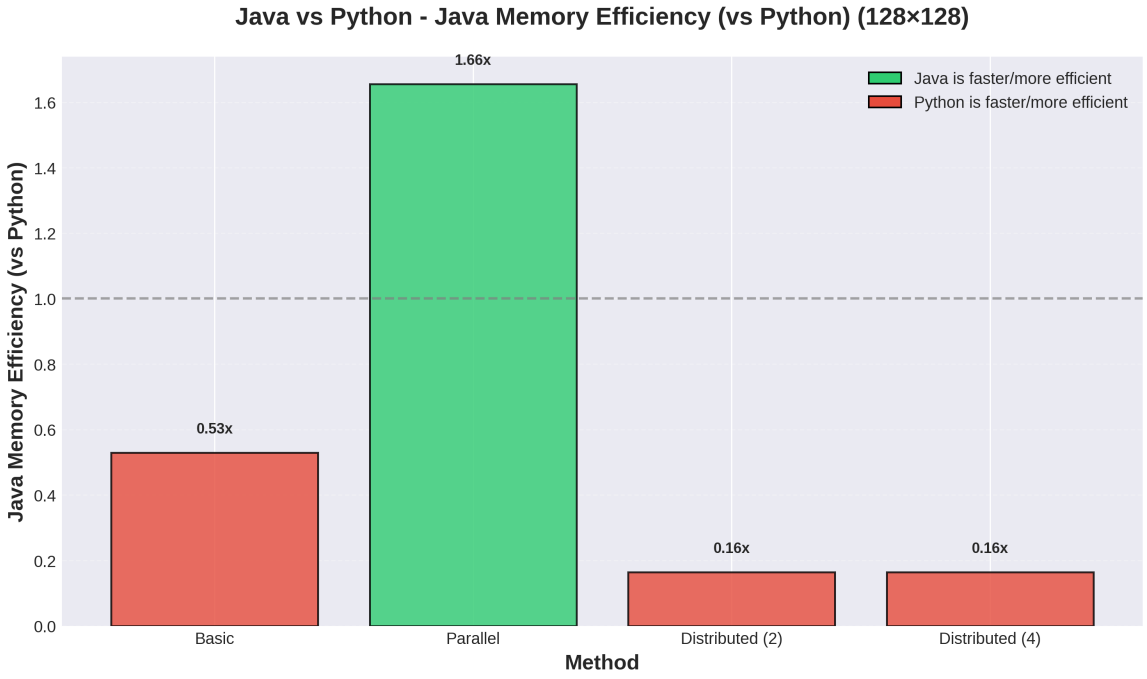

Figure 18: Memory efficiency between Java and Python across algorithms for matrix size 128x128

Java uses significantly more memory than Python for all matrix sizes and methods, especially in distributed approaches. For distributed 2-node clusters, Java's memory usage for 128×128 matrices is 3,008 KB, while Python uses only 493 KB. This gap widens with larger matrices: for 2048×2048,

Java consumes 67,584 KB (2 nodes) and 93,627 KB (4 nodes), whereas Python's largest distributed memory usage is just 494 KB (128×128, 4 nodes). Memory per node scales as expected in both languages, halving when moving from 2 to 4 nodes, but Java's Hazelcast client and JVM overheads result in much higher and more variable memory consumption compared to Python's lightweight client. Overall, Python is far more memory-efficient, but Java's higher memory use is offset by much faster execution.

7 Summary and Conclusion

This assignment successfully achieved its objective of designing and implementing a distributed matrix multiplication system using Hazelcast, with a clear comparison to basic (serial) and parallel (multi-threaded) approaches in both Java and Python. By simulating a cluster on a single machine—running multiple Hazelcast nodes on different ports—the project enabled detailed analysis of distributed computing behavior in a controlled environment.

The results demonstrate that, while the distributed approach effectively partitions data and prevents "Out of Memory" errors by distributing memory usage across nodes, it introduces substantial network overhead and coordination costs, especially when all nodes share the same physical resources. Scaling from 2 to 4 nodes did not yield performance improvements; instead, it often increased execution time and memory consumption due to resource contention and additional communication overhead. Java consistently outperformed Python in both execution and network times, but at the cost of higher memory usage, while Python proved more memory-efficient but much slower overall.

The evaluation highlights that, on a single machine, distributed matrix multiplication is dominated by network and coordination overhead, making it less efficient than local parallelization for the tested problem sizes. However, the system's architecture and metrics collection provide a robust foundation for scaling to true multi-machine clusters, where the benefits of distributed computation and Hazelcast's data partitioning would become more pronounced. This project underscores the importance of considering both software architecture and deployment environment when designing scalable big data solutions.

List of sources

[1] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. 4th ed. Baltimore, MD: Johns Hopkins University Press, 2013.

[2] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. 2nd. Cambridge University Press, 2014.

[3] Ananth Grama et al. *Introduction to Parallel Computing*. 2nd. Pearson Education, 2003.

[4] Tom White. *Hadoop: The Definitive Guide*. 4th. Discusses pseudo-distributed mode for simulating clusters on a single machine. " O'Reilly Media, Inc.", 2015.

[5] Hazelcast. *Hazelcast Reference Manual: In-Memory Computing*. 2024. URL: `https://docs.hazelcast.com/`.