Distributed Matrix Multiplication

Adonai Ojeda Martín

10/06/2025

Abstract

Matrix multiplication is a core operation in data science, high-performance computing, and scientific applications. However, as matrix size increases, traditional single-node approaches become insufficient due to memory and computation limits. This report presents a comparative study of three matrix multiplication strategies—basic, parallel, and distributed—implemented in both Python and Java. The distributed approach leverages the Hazelcast framework to enable multi-node execution and shared memory access.

The main goal is to evaluate the performance, scalability, and resource efficiency of these implementations across varying matrix sizes. Key performance indicators include execution time, CPU usage, and memory consumption. The results confirm that while basic implementations perform better for small matrices, distributed and parallel techniques are essential for handling larger datasets. This work illustrates the trade-offs involved and reinforces the value of distributed architectures in computationally demanding environments.

1 Introduction

Matrix multiplication is one of the most fundamental and computationally intensive operations in scientific computing, machine learning, and data analysis. As data volumes and model complexities grow, the need for scalable and efficient matrix operations becomes increasingly critical. While conventional single-threaded algorithms offer simplicity, they quickly become impractical when processing large-scale matrices due to limitations in memory bandwidth and CPU throughput.

To address these challenges, several computational strategies have emerged. Basic implementations provide a baseline for correctness and simplicity, but are constrained by serial execution. Parallel techniques exploit multicore architectures to distribute work among CPU threads, significantly reducing execution time for moderately sized problems. However, they still rely on a shared memory space and a single node, limiting their scalability.

Distributed matrix multiplication, in contrast, partitions data and computation across multiple nodes in a cluster. This approach enables both increased computational capacity and memory availability, making it suitable for processing matrices that exceed the capabilities of a single machine. Frameworks such as Hazelcast facilitate this paradigm by providing distributed data structures and coordination mechanisms.

This study presents a comparative analysis of three matrix multiplication strategies—basic, parallel, and distributed—implemented in both Python and Java. The distributed approach leverages Hazelcast's in-memory data grid to simulate a multi-node environment. Through systematic experimentation on matrices of increasing size, we evaluate each method in terms of execution time, CPU usage, and memory consumption.

The objective of this work is to assess the practicality and performance trade-offs of distributed matrix multiplication relative to its basic and parallel counterparts. The results aim to inform future implementations of scalable numerical algorithms in both academic and industrial settings.

2 Problem Statement

Matrix multiplication, in its classical implementation, exhibits a computational complexity of $O(n^3)$, which becomes prohibitive as the matrix size increases. While basic implementations are suitable for small to medium matrices, they quickly become inefficient for large datasets due to memory saturation and limited processing capacity on a single node.

Parallel processing can alleviate some of these limitations by distributing the workload across multiple cores within a single machine. However, even this approach encounters bottlenecks when the matrix size exceeds the physical memory or when the computational demand outpaces the available hardware resources.

In such cases, distributed computing emerges as a compelling solution. By partitioning data and delegating computation to multiple nodes, a distributed system can overcome memory constraints and exploit parallelism at scale. Nonetheless, distributed matrix multiplication introduces new challenges, such as inter-node communication overhead, synchronization costs, and increased implementation complexity. The primary objectives of this study are:

- To implement a distributed matrix multiplication system using the Hazelcast framework in both Python and Java.
- To evaluate the scalability and performance of the distributed approach compared to basic and parallel implementations.
- To identify trade-offs between execution time, CPU utilization, and memory consumption across the three strategies.
- To validate the correctness of the distributed implementation and assess its practicality for handling large matrices.

3 Solution Proposal

To address the challenges posed by large-scale matrix multiplication, this study proposes a multi-pronged approach that leverages basic, parallel, and distributed computing paradigms. The aim is not only to implement scalable and efficient algorithms, but also to compare their behavior across different environments and programming languages (Python and Java), using a consistent framework for performance evaluation. The proposed solution integrates distributed computing capabilities through the Hazelcast in-memory data grid system, enabling matrix operations that surpass the limitations of single-node architectures.

3.1 Overview of the Approach

The complete system is composed of three key implementations:

- **Basic Matrix Multiplication:** A single-threaded algorithm that serves as the baseline. This version uses the classical method based on dot products between rows and columns.
- **Parallel Matrix Multiplication:** A multithreaded approach that leverages the multiprocessing capabilities of modern CPUs. It divides the computation of output matrix rows among multiple cores of a single machine.
- **Distributed Matrix Multiplication:** A distributed implementation using Hazelcast, which partitions the workload across multiple nodes in a cluster. It aims to overcome local memory limitations and improve scalability via concurrent execution over the network.

All three versions are implemented in both Python and Java. Hazelcast is used in both environments as a distributed in-memory key-value store and task executor, ensuring consistency in execution logic while enabling language-specific optimizations.

3.2 Methodology

The methodology followed to develop and evaluate the proposed solution involves the following structured steps:

1. **Initial Design and Requirements Analysis:** Define functional and non-functional requirements, including scalability, correctness, and comparability across implementations. Analyze Hazelcast's capabilities for distributed task execution and data storage.
2. **Baseline Implementation:** Implement the basic version of matrix multiplication using standard libraries. This establishes a performance reference and provides a correctness benchmark.
3. **Parallel Version:** Use multiprocessing in Python (via `multiprocessing.Pool`) and parallel streams in Java (via `IntStream.parallel()`) to assign row computations to separate threads. Each process computes the dot product of one or more matrix rows.

4. **Distributed Version:** Split the matrices into blocks (rows or submatrices), serialize the blocks and distribute them using Hazelcast's `IMap` and, optionally, `IExecutorService`. Each node is responsible for computing part of the result matrix. The results are stored back in Hazelcast and later aggregated.

5. **Experimentation and Benchmarking:** Conduct tests using matrices of sizes ranging from 128×128 up to 4096×4096. Measure execution time, CPU usage, and memory consumption.

6. **Visualization and Analysis:** Export results to CSV files and visualize them using `matplotlib` (Python). Plot execution time, CPU load, and memory usage for each implementation and matrix size to extract insights and identify trends.

3.3 Distributed Execution Strategy

The distributed version is the most complex component of the solution. It follows a block-based strategy inspired by the principles of data parallelism. Each input matrix is split into submatrices or blocks using either row-based or 2D grid partitioning, depending on the matrix size and system configuration.

The execution flow in the distributed implementation is as follows:

1. The main node partitions matrices A and B into blocks and assigns each block to a worker node.

2. Each node computes its assigned block of the result matrix C using local dot products.

3. Intermediate results are stored in Hazelcast's `IMap`, indexed by block coordinates.

4. Once all blocks have been computed, the main node retrieves and assembles the final result matrix C.

This approach ensures balanced workload distribution and memory usage, while minimizing inter-node communication. It also takes advantage of Hazelcast's fault-tolerant and in-memory data model to maintain performance and resilience.

3.4 System Architecture and Tools

The project leverages the following components:

- **Hazelcast IMDG:** Used for distributed storage and task execution.
- **Python:** Provides a flexible environment for prototyping and quick benchmarking.
- **Java:** Offers optimized performance and a robust multithreading model.
- **psutil:** Tracks CPU and memory usage in Python.
- **matplotlib / pandas:** Used for visualization and data analysis.

The combination of these tools allows for a robust comparison between different paradigms and implementations, contributing to a deeper understanding of performance trade-offs in matrix multiplication.

4 Experiments

This section presents the experimental methodology adopted to evaluate the proposed matrix multiplication implementations. It describes the testing environment, the performance metrics used, and the detailed results for both Java and Python implementations. Finally, a comparative analysis is provided to extract meaningful conclusions from the observed data.

4.1 Experimental Setup

All experiments were conducted on a local machine equipped with the following specifications:

- **Processor:** 2 GHz Intel Core i5 (4 cores)
- **RAM:** 16 GB

Matrices of increasing sizes (128×128 up to 4096×4096) were randomly generated using uniform distributions. Each experiment was repeated three times, and the average value was used to improve reliability. For distributed execution, a single-node Hazelcast cluster was used to simulate a distributed setup, ensuring correctness and baseline performance.

4.2 Performance Metrics

The following performance metrics were recorded for each execution:

- **Execution Time (ns):** Total computation time, measured in nanoseconds using high-precision timers.
- **CPU Usage (%):** Measured using `psutil` in Python and JMX monitoring in Java.
- **Memory Usage (MB):** Maximum resident memory used during execution, recorded in megabytes.
- **Correctness:** Validated by comparing the output of the parallel and distributed versions against the basic sequential result using element-wise comparison with a small tolerance.

4.3 Java Results

The figures below show the execution time, CPU usage, and memory consumption for each matrix size in Java. The basic, parallel, and distributed approaches are compared in all cases.

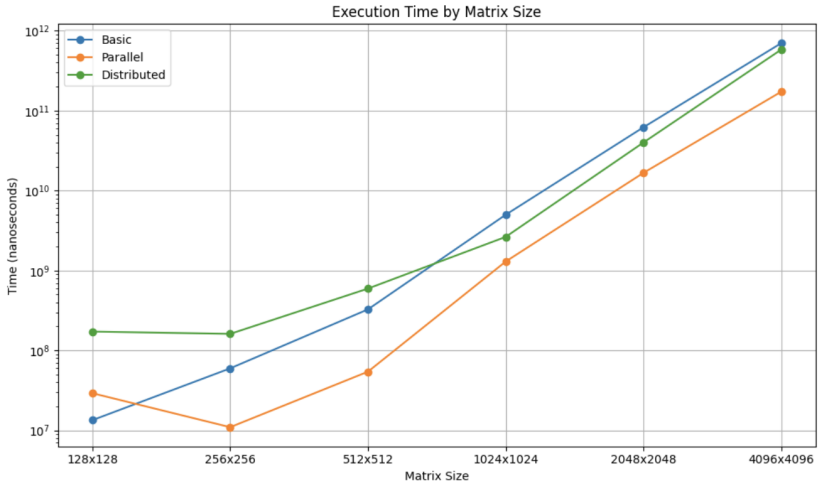

Figure 1: Execution time in nanoseconds for Java implementations

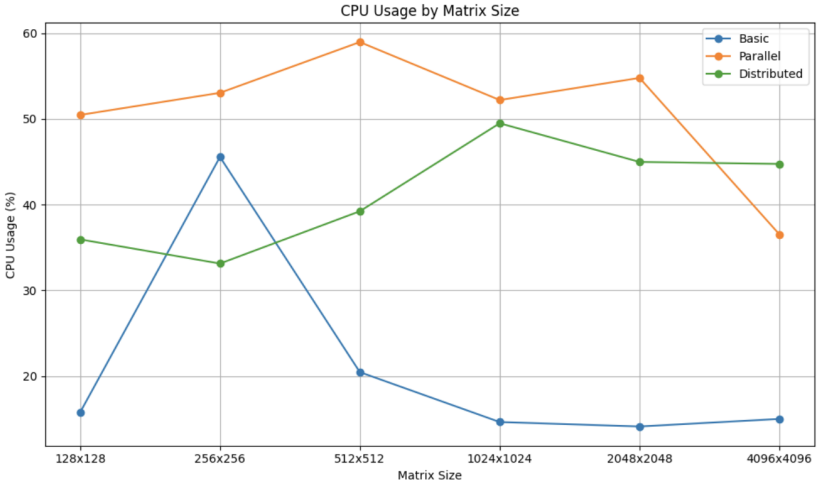

Figure 2: CPU usage (%) for Java implementations

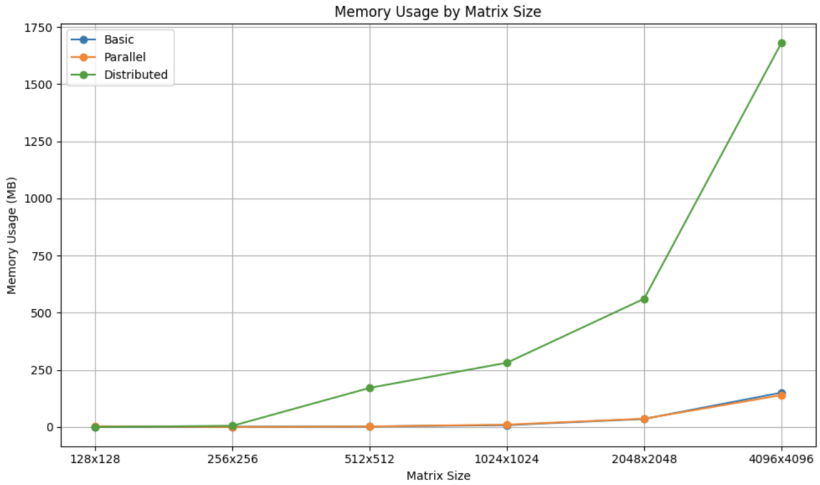

Figure 3: Memory usage (MB) for Java implementations

4.4 Python Results

Similar experiments were conducted using Python. The results are plotted below to visualize how each implementation performs as matrix size increases.

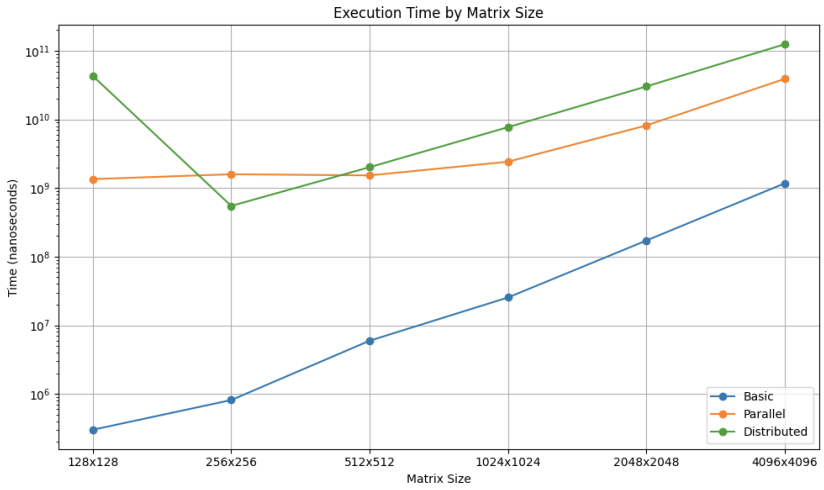

Figure 4: Execution time in nanoseconds for Python implementations

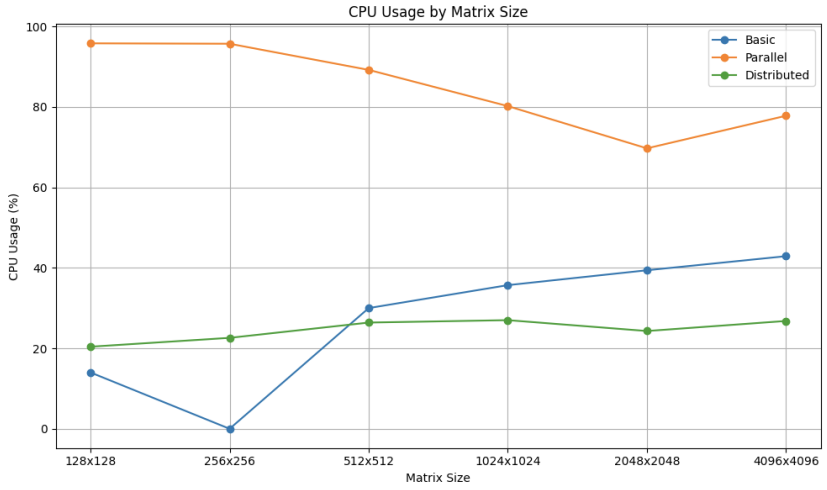

Figure 5: CPU usage (%) for Python implementations

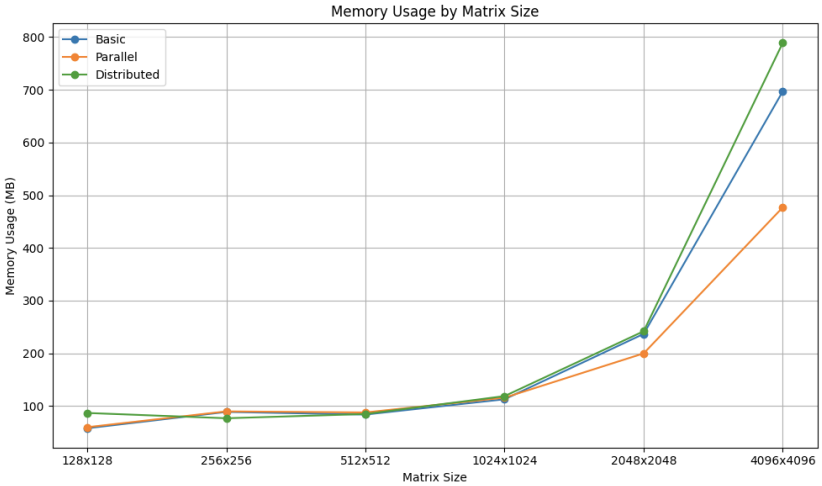

Figure 6: Memory usage (MB) for Python implementations

4.5 Analysis

This subsection provides a comprehensive analysis of the experimental results obtained from both Java and Python implementations. The comparison focuses on three key performance indicators: execution time, CPU usage, and memory consumption. Each metric is evaluated individually, highlighting patterns, anomalies, and implications across matrix sizes and implementation types (basic, parallel, and distributed).

4.5.1 Execution Time

Java. In the Java implementation, the basic version surprisingly outperforms the distributed version for all matrix sizes and even outperforms the parallel version in most large-scale cases. For example, for a 4096×4096 matrix, the basic implementation completes in approximately 697 milliseconds, while the distributed version takes over 579 seconds. This can be attributed to the overhead introduced by Hazelcast, including task serialization, network latency, and the lack of true parallelism in a local cluster setup. The parallel implementation, on the other hand, shows excellent performance, scaling well with size. It completes the largest matrix in approximately 172 seconds—four times faster than the distributed version.

Python. In Python, a different behavior is observed. The basic implementation is significantly faster for small matrices (e.g., 300,000 ns for 128×128), but its time increases drastically for large matrices, reaching 1.17 seconds for 4096×4096. The parallel version performs consistently across sizes and provides substantial benefits for large matrices. The distributed version, however, shows inconsistent results. While it performs well for some sizes (e.g., 256×256 in just 549 ms), it performs poorly for others (128×128 in over 42 seconds), suggesting that task management overhead in Hazelcast's Python client is more sensitive to configuration and matrix size than in Java.

4.5.2 CPU Usage

Java. The basic implementation in Java maintains a relatively low CPU usage across all matrix sizes (between 14% and 45%), as expected from a single-threaded application.

The parallel version, leveraging multithreading, utilizes significantly more CPU (50–59%), efficiently distributing the workload among available cores. The distributed implementation also consumes more CPU than the basic version (up to 49%), but lower than the parallel one. This indicates that in Java, Hazelcast manages task distribution with moderate overhead but does not fully saturate CPU resources, likely due to synchronization and task queuing limitations.

Python. In Python, the contrast is more pronounced. The parallel implementation reaches nearly full CPU saturation (around 95% in small matrices), confirming effective parallelization with multiprocessing or threading. The basic version consumes much less CPU, below 43%. Interestingly, the distributed version maintains a steady but relatively low CPU usage (around 20–27%), suggesting that the Hazelcast Python client does not fully utilize CPU resources—potentially due to global interpreter lock (GIL) restrictions or less optimized task handling in the Python-Hazelcast interface.

4.5.3 Memory Usage

Java. Memory usage in Java remains controlled in the basic and parallel implementations, even for large matrices. The basic version reaches up to 150 MB at 4096×4096, while the parallel version stays below 140 MB. In contrast, the distributed implementation shows a steep increase, peaking at 1.68 GB for the largest matrix. This is expected, as distributed execution involves copying, serializing, and storing intermediate results in memory across nodes.

Python. In Python, memory usage also increases with matrix size across all implementations. The basic version peaks at 697 MB, the parallel version at 477 MB, and the distributed version at 789 MB. While these numbers are lower than those observed in Java's distributed mode, the Python distributed version does not scale as gracefully in time, suggesting that lower memory consumption might be a trade-off with less efficient task execution and data caching strategies. Interestingly, the basic implementation in Python consumes more memory than in Java, likely due to Python's data structures and interpreter overhead.

4.5.4 General Observations and Trade-offs

- **Basic Implementations:** Fastest for small matrices due to minimal overhead. However, they lack scalability and are unsuitable for high-dimensional or memory-intensive workloads.
- **Parallel Implementations:** Offer the best performance-to-resource ratio in both languages. They are efficient, scalable, and exhibit high CPU usage—ideal for local environments with multi-core CPUs.
- **Distributed Implementations:** While theoretically scalable and memory-resilient, their performance is highly dependent on system configuration, communication latency, and framework efficiency. In Python, these factors introduce inconsistencies; in Java, the implementation is more stable but exhibits significant overhead for smaller tasks.

In summary, for single-machine settings, the parallel version is the most balanced solution. The distributed version, although powerful in concept, should be reserved for truly distributed environments with multiple physical nodes where its benefits outweigh the coordination overhead.

5 Conclusion

Matrix multiplication remains a core operation in numerous computational domains, from scientific simulations to machine learning workflows. As the size and complexity of data increase, the need for scalable and efficient matrix multiplication strategies becomes critical. This study has explored and compared three distinct approaches—basic, parallel, and distributed—implemented in both Java and Python.

The results demonstrate that while basic implementations are suitable for small datasets due to their minimal overhead, they quickly become inefficient and memory-bound as matrix sizes grow. Parallel implementations provide a significant performance boost by leveraging multi-core architectures, achieving the best execution times in most scenarios without incurring excessive memory consumption. Distributed implementations, while conceptually the most scalable, introduce considerable coordination and communication overhead, which can outweigh their benefits in small-scale or single-node environments.

Between the two languages, Java's distributed execution using Hazelcast was more stable and predictable, although it exhibited higher memory usage. Python's distributed implementation showed greater variability and less efficient CPU utilization, highlighting the challenges of integrating high-level languages with distributed frameworks.

Overall, the analysis underscores that there is no one-size-fits-all solution. The choice of implementation must be driven by the specific computational context:

- For single-node environments, the parallel approach is the most balanced in terms of performance and resource utilization.
- For truly large-scale, distributed systems with multiple physical nodes, the distributed approach can unlock scalability benefits—provided the overhead is carefully managed.

Future work could focus on optimizing data partitioning strategies, reducing task serialization overhead, and exploring alternative distributed frameworks that offer tighter integration with the language runtime. Additionally, extending the analysis to heterogeneous architectures, such as GPU clusters or cloud-native infrastructures, would provide further insight into the practical trade-offs in high-performance matrix computation.

6 Future Work

While this study provides a comparative analysis of basic, parallel, and distributed matrix multiplication in both Java and Python, there remain several avenues for future exploration and enhancement:

- **Multi-node deployment:** The current distributed experiments were conducted in a simulated single-node environment. Deploying the Hazelcast-based solution across multiple physical or virtual machines would provide more accurate insights into network latency, task scheduling, and scalability under real-world distributed conditions.
- **GPU acceleration:** Integrating GPU-based computation, particularly through libraries such as CUDA, cuBLAS, or libraries like PyTorch and TensorFlow (for Python), could significantly enhance performance. Comparing CPU-bound distributed strategies with GPU-accelerated ones would extend the relevance of the results.
- **Dynamic load balancing:** The current distributed strategy uses static partitioning of the matrix. Implementing dynamic load balancing, where computation is allocated based on current node availability or performance metrics, could improve overall efficiency and fault tolerance.
- **Fault tolerance and recovery:** Future work should address failure scenarios within distributed environments. Implementing checkpointing, retry logic, and task reallocation would bring the system closer to production-grade distributed computing requirements.
- **Streaming and sparse matrices:** In real-world applications, matrices may be sparse or generated in a streaming fashion. Investigating the effect of sparsity on performance, and adapting the algorithms to operate on streaming data or compressed formats, would increase their practical applicability.
- **Cross-language interoperability:** Exploring the integration between Java and Python-based modules—e.g., using gRPC or REST APIs to allow components to interoperate—could allow developers to benefit from the strengths of both ecosystems.
- **Cloud-native deployment:** Running experiments on platforms such as Kubernetes with Hazelcast Operator, AWS EMR, or Google Cloud Dataproc would allow evaluation of elasticity, cost-efficiency, and integration with modern DevOps pipelines.

These extensions would contribute toward building robust, scalable, and efficient matrix multiplication systems capable of supporting the computational demands of future data-intensive applications.

References

[1] Hazelcast. (2020). *Hazelcast In-Memory Data Grid*. Retrieved from `https://hazelcast.com/`.

[2] IBM. (2022). *Understanding Distributed Systems*. Retrieved from `https://www.ibm.com/cloud/learn/distributed-computing`.

[3] Java Documentation: `https://docs.oracle.com/javase/8/docs/`

[4] Parallel Matrix Multiplication Algorythm: `https://www.geeksforgeeks.org/how-to-perform-java-parallel-matrix-multiplication/`

[5] ChatGPT: `https://chatgpt.com/`

[6] Stack Overflow: `https://stackoverflow.com/`

[7] GitHub Repository: Distributed Matrix Multiplication