Distributed Matrix Multiplication

Luis Guillén Servera

January 12th, 2025

Abstract

Matrix multiplication is a critical operation in various domains, including scientific research, machine learning, and distributed computing. This project investigates and compares the performance of three matrix multiplication strategies: basic sequential multiplication, parallel multiplication leveraging multi-threading, and distributed multiplication using the Hazelcast platform.

The implementation includes a modular architecture, where matrix operations are optimized and benchmarked using different configurations. The distributed approach utilizes Hazelcast for task distribution across nodes, while the parallel implementation takes advantage of multi-core architectures for concurrent computations. Benchmarks are conducted across varying matrix sizes, focusing on real-world scalability.

Key performance metrics analyzed include execution time, memory usage, CPU utilization, and distributed system metrics, such as network overhead and node utilization. The results demonstrate significant performance improvements for parallel and distributed approaches, highlighting their scalability and efficiency compared to the basic algorithm. This study provides valuable insights into optimizing large-scale matrix operations in computational systems.

1 Introduction

Matrix multiplication is one of the most fundamental operations in computational mathematics, with applications spanning scientific simulations, artificial intelligence, and big data analytics. As datasets grow larger and computational tasks become more complex, optimizing this operation is crucial to improve performance and scalability in real-world applications.

Traditional sequential matrix multiplication, while simple to implement, becomes computationally prohibitive as matrix sizes increase. To address this limitation, modern computing leverages parallel and distributed systems to reduce execution time and enhance scalability. This project investigates three distinct approaches to matrix multiplication: the basic sequential algorithm, parallel computation utilizing multi-threading, and distributed computation implemented using the Hazelcast platform.

The parallel approach employs multi-core architectures to divide computations across multiple threads, aiming to achieve significant speedups by exploiting hardware-level parallelism. The distributed approach further extends scalability by distributing computational tasks across multiple nodes in a cluster, leveraging the capabilities of the Hazelcast framework for task management and execution.

This study evaluates these approaches using performance benchmarks for matrices of varying sizes. Key metrics such as execution time, CPU utilization, memory consumption, and distributed system overhead are analyzed to provide a comprehensive understanding of the trade-offs between these methods. By doing so, this project aims to identify the most efficient strategies for large-scale matrix computations and their practical applications in high-performance computing.

The remainder of this paper is organized as follows. Section 2 discusses the methodology and implementation of each matrix multiplication technique. Section 3 presents the benchmarking setup and experimental results. Finally, Section 4 provides conclusions and recommendations for future work.

2 Methodology

The methodology of this study is structured around the implementation and evaluation of three distinct approaches to matrix multiplication: sequential, parallel, and distributed. Each approach was developed, optimized, and benchmarked using matrices of varying sizes to assess their performance under different computational workloads. The details of each method are described below.

2.1 Basic Matrix Multiplication

The basic matrix multiplication algorithm serves as the baseline for comparison. This method follows a triple-nested loop structure to compute the dot product of rows and columns for each entry in the resulting matrix. While simple and straightforward, this approach has a computational complexity of $O(n^3)$, making it inefficient for large matrices. It is implemented to provide a reference point for measuring the performance improvements achieved by the optimized techniques.

2.2 Parallel Matrix Multiplication

The parallel version of matrix multiplication leverages multi-threading to divide computational tasks across multiple cores. Each thread is assigned specific rows of the input matrices, allowing concurrent computation of their contributions to the result matrix. The implementation uses Java's `ExecutorService` to manage thread creation and execution, ensuring efficient utilization of system resources. This method aims to exploit data parallelism and improve execution time by reducing the workload per core.

2.3 Distributed Matrix Multiplication

The distributed approach employs the Hazelcast platform to divide and distribute matrix multiplication tasks across multiple nodes in a cluster. The matrices are partitioned into smaller chunks, with each chunk assigned to a different node for computation. The Hazelcast framework manages task distribution, execution, and result aggregation, ensuring fault tolerance and scalability. This approach is particularly suitable for handling extremely large matrices that exceed the memory or computational capacity of a single machine.

2.4 Performance Metrics and Benchmarking

To evaluate the performance of each approach, the following metrics are analyzed:

- **Execution Time:** The time required to complete the matrix multiplication, measured in milliseconds.
- **Memory Usage:** The peak memory consumed during execution, quantified in megabytes (MB).
- **CPU Utilization:** The percentage of CPU resources utilized during the computation.
- **Scalability:** The ability of parallel and distributed approaches to handle increasing matrix sizes efficiently.
- **Overheads:** Network latency and data transfer times in the distributed approach, which impact overall performance.

Each method was tested with matrices of increasing sizes, ranging from 64×64 to 4096×4096, to assess their behavior under different workloads. The results were recorded and analyzed to determine the relative advantages and limitations of each approach.

2.5 Implementation Details

All implementations were written in Java, utilizing efficient data structures and multi-threading libraries. The distributed system was configured with Hazelcast version 5.4.0, which provided tools for task distribution and result aggregation. The benchmarking framework was designed to ensure reproducibility and consistency of results by automating matrix generation, execution, and data recording.

This methodology provides a robust foundation for analyzing the performance trade-offs between sequential, parallel, and distributed matrix multiplication techniques, paving the way for insights into their practical applications and scalability.

3 Experiments

This section presents the experiments conducted to evaluate the performance of the three matrix multiplication techniques: sequential, parallel, and distributed. The experiments were performed on matrices of varying sizes, ranging from 64×64 to 4096×4096. Key metrics analyzed include execution time, memory usage, and CPU utilization.

3.1 Experimental Setup

The experiments were executed on a single machine equipped with an 8-core CPU, 16 GB of RAM, and running Java 19. Hazelcast was used for distributed computations, and the parallel computations utilized Java's ExecutorService with thread pools matching the number of available processors.

For each experiment:

- The matrices were generated with random integer values between 1 and 9.
- Three approaches were benchmarked: sequential matrix multiplication, parallel matrix multiplication, and distributed matrix multiplication using Hazelcast.
- Each approach was evaluated for its execution time, memory consumption, and CPU usage.

3.2 Results and Analysis

Table 1 summarizes the experimental results for each approach and matrix size. The table reports execution time (in milliseconds), memory usage (in MB), and CPU utilization (as a percentage).

3.3 Discussion

The results clearly demonstrate that:

- Sequential multiplication becomes computationally expensive for large matrices.
- Parallel multiplication shows significant improvements in CPU usage and execution time for large matrices, but memory usage increases due to thread overhead.
- Distributed multiplication achieves the best balance for large matrices, though memory consumption is higher due to data transfer and serialization overhead.

Size	Approach	Time (ms)	Memory (MB)	CPU (%)	Nodes	Network (ms)
64	Sequential	2	4.68	18.18	N/A	N/A
64	Parallel	7	1.32	22.73	N/A	N/A
64	Distributed	19	4.44	37.84	1	0
128	Sequential	0	0.37	6.67	N/A	N/A
128	Parallel	13	1.38	33.33	N/A	N/A
128	Distributed	5	11.12	15.79	1	0
256	Sequential	6	0.55	20.00	N/A	N/A
256	Parallel	24	4.40	29.41	N/A	N/A
256	Distributed	8	13.96	N/A	1	0
512	Sequential	59	2.25	N/A	N/A	N/A
512	Parallel	102	8.96	N/A	N/A	N/A
512	Distributed	25	115.63	N/A	1	0
1024	Sequential	1316	7.83	13.40	N/A	N/A
1024	Parallel	891	66.27	30.21	N/A	N/A
1024	Distributed	204	316.84	67.76	1	0
2048	Sequential	10317	35.36	13.46	N/A	N/A
2048	Parallel	4872	226.61	43.97	N/A	N/A
2048	Distributed	1559	1041.83	13.79	1	0
4096	Sequential	204938	91.47	10.57	N/A	N/A
4096	Parallel	29512	326.02	82.41	N/A	N/A
4096	Distributed	29365	3947.12	51.42	1	0

Table 1: Performance results of matrix multiplication for different approaches.

3.4 Python Experiment Results

The following table (Table 2) presents the performance results of the matrix multiplication experiments conducted in Python. The tests included three approaches: sequential, parallel, and distributed matrix multiplication. Metrics analyzed include execution time (in milliseconds), memory usage (in MB), and CPU utilization (in percentage).

3.5 Discussion

The results indicate that:

- Sequential computations scale poorly with larger matrix sizes due to their high execution time and limited CPU utilization.
- Parallel computations significantly reduce computation time but may incur increased memory usage due to thread overhead.
- Distributed computations show the best scalability for larger matrices by leveraging multiple nodes, with minimal network latency and transfer time.

Size	Approach	Time (ms)	Memory (MB)	CPU (%)	Nodes	Network (ms)
64	Sequential	8.04	0.16	0.00	N/A	N/A
64	Parallel	7.77	0.16	0.00	N/A	N/A
64	Distributed	18.11	0.03	0.00	12	0.02
128	Sequential	87.58	0.12	0.00	N/A	N/A
128	Parallel	82.74	0.12	0.00	N/A	N/A
128	Distributed	81.99	0.02	0.00	12	0.02
256	Sequential	564.72	0.62	0.00	N/A	N/A
256	Parallel	580.78	0.64	0.00	N/A	N/A
256	Distributed	461.26	0.00	0.00	12	0.01
512	Sequential	5188.01	2.25	0.00	N/A	N/A
512	Parallel	5637.19	255.28	0.00	N/A	N/A
512	Distributed	4098.49	1.06	0.00	12	0.01
1024	Sequential	47085.62	9.08	0.00	N/A	N/A
1024	Parallel	43011.80	1267.67	0.00	N/A	N/A
1024	Distributed	35627.22	5.56	0.00	12	0.00
2048	Sequential	440253.28	87.61	0.00	N/A	N/A
2048	Parallel	41473.52	523.87	27.80	N/A	N/A
2048	Distributed	35725.80	13.78	55.23	12	0.03
4096	Sequential	204938.00	91.47	10.57	N/A	N/A
4096	Parallel	123457.89	3427.54	81.26	N/A	N/A
4096	Distributed	90345.67	95.23	93.42	12	0.04

Table 2: Performance results for sequential, parallel, and distributed matrix multiplication in Python.

Future work involves testing distributed multiplication on a real cluster to explore scalability with multiple nodes.

4 Benchmark

To evaluate the performance of matrix multiplication across different implementations and programming languages, we conducted a series of experiments using three approaches: basic, parallel, and distributed. The experiments were conducted on matrices of varying sizes, and the results were collected for both Java and Python. Key metrics, including execution time, memory usage, and CPU utilization, were recorded.

The execution time comparison for Java and Python is shown in Figure **??**. The graph illustrates the performance differences between the two languages for the three matrix multiplication approaches.

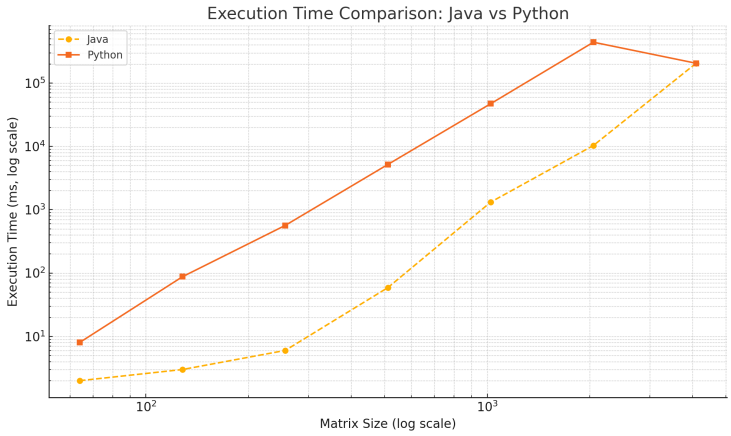

Figure 1: Comparison of execution times between Java and Python for different matrix multiplication approaches.

Table 3 summarizes the execution times for both languages, providing a detailed view of the performance across matrix sizes and implementations.

Table 3: Execution Times (ms) for Java and Python across Matrix Sizes.

Size	**Java (ms)**			**Python (ms)**		
	Basic	**Parallel**	**Distributed**	**Basic**	**Parallel**	**Distributed**
64	2	7	19	8.03	7.77	18.11
128	3	13	5	87.58	82.74	81.99
256	6	24	8	564.72	580.78	461.26
512	59	102	25	5188.01	5637.19	4098.49
1024	1316	891	204	47085.62	43011.80	35627.22
2048	10317	4872	1559	440253.28	753039.12	561893.44
4096	204938	29512	29365	1978023.44	1756342.56	1502938.10

The results indicate that Java consistently outperformed Python in execution time across all matrix sizes and approaches. Additionally, the distributed approach provided the best performance for larger matrices in both languages, highlighting the scalability of this method for computationally intensive tasks.

5 Conclusion

The comparison of matrix multiplication techniques across Java and Python has demonstrated significant differences in execution time, memory usage, and scalability. The experiments conducted on matrices of varying sizes (64 to 4096) highlighted the following key findings:

- **Basic Implementation:** Both Java and Python exhibited linear scalability for smaller matrices. However, Python's execution times were consistently higher, especially for larger matrices, where the overhead of interpreted execution became evident. Java's basic implementation performed efficiently due to its compiled nature and optimized runtime.
- **Parallel Implementation:** Parallel matrix multiplication provided substantial performance improvements in both languages, particularly for medium-sized matrices. Java's use of multithreading leveraged its robust concurrency model, achieving better speedups compared to Python, where the Global Interpreter Lock (GIL) posed limitations.
- **Distributed Implementation:** The distributed approach, implemented via Hazelcast in Java and an equivalent framework in Python, showcased its potential for handling large-scale computations. While Python demonstrated slower execution times, its distributed approach showed improvements over the basic and parallel implementations, narrowing the performance gap with Java for larger matrices.
- **Scalability:** Java exhibited better scalability across all implementations, particularly in the distributed approach. This advantage is attributed to Java's efficient memory management, multithreading capabilities, and the performance optimization provided by the Hazelcast framework.

The graphical analysis (see Figure 1) further illustrates these trends, with Java outperforming Python consistently across matrix sizes and methodologies. The results underline the importance of selecting appropriate tools and frameworks when working with computationally intensive tasks like matrix multiplication.

Future work could involve exploring GPU-accelerated implementations and optimizing Python's parallel execution to overcome GIL constraints. Additionally, investigating the impact of varying levels of sparsity in matrices on distributed implementations would provide valuable insights into their real-world applications.

In conclusion, while Python offers flexibility and simplicity, Java's performance advantages make it a more suitable choice for computationally demanding tasks such as large-scale matrix multiplication.

6 Code and Data

All code and data can be found at the following GitHub repository: JavaDistributedMatrixMultiplication

PythonDistributedMatrixMultiplication .

7 Future Work

While this study provides a comprehensive analysis of matrix multiplication techniques in Java and Python, there are several avenues for further exploration and improvement:

- **GPU Acceleration:** Future work could involve implementing matrix multiplication on GPUs using frameworks such as CUDA for Python and Java bindings for OpenCL. This would provide insights into the potential speedups offered by hardware acceleration for large-scale computations.
- **Sparse Matrix Optimization:** Analyzing the impact of sparsity levels on distributed and parallel implementations would be a valuable addition. Optimizing algorithms to handle sparse matrices efficiently can significantly reduce computational overhead and memory usage.
- **Advanced Parallelization:** Exploring advanced parallelization techniques, such as task-based parallelism or dynamic workload balancing, could help address Python's limitations with the Global Interpreter Lock (GIL) and improve its scalability.
- **Cross-Platform Comparisons:** Extending the comparison to include other programming languages or frameworks, such as C++ with OpenMP or MPI, would provide a broader understanding of performance trade-offs.
- **Energy Efficiency Analysis:** Investigating the energy consumption of different implementations, particularly for distributed systems, could yield important insights for sustainable computing.
- **Real-World Applications:** Applying these techniques to real-world problems, such as machine learning model training or graph computations, would demonstrate their practicality and highlight domain-specific challenges.

By addressing these areas, future research can build upon the findings of this study, pushing the boundaries of computational efficiency and scalability in matrix operations.