TASK 4. Distributed Matrix Multiplication Big Data

Denys Kavkalo Gumeniuk

Grado en Ciencia e Ingeniería de Datos

Universidad de Las Palmas de Gran Canaria

July 5, 2025

Abstract

This paper presents the implementation and evaluation of distributed matrix multiplication using Hazelcast, a distributed computing framework. The study compares this distributed approach, implemented in both Java and Python, with local strategies including basic, parallel, and vectorized multiplication methods. Experiments are conducted with matrices of increasing sizes (from 10×10 up to 3000×3000), analyzing metrics such as execution time per operation, upload time, memory usage, transfer rate, and memory allocation rate. The results demonstrate the benefits and limitations of distributed computation, showing how it enables the processing of large matrices and highlighting the trade-offs between computational speed, memory consumption, and data transfer costs in distributed environments.

1 Background

Matrix multiplication is a fundamental computational operation used extensively in scientific computing, data analysis, and machine learning. Traditional algorithms, such as the $O(n^3)$ nested loop method, are effective for small matrices but face scalability issues as matrix sizes increase. On a single machine, parallel or vectorized implementations can enhance performance by utilizing multiple threads or CPU instructions, but they remain constrained by memory and processing power.

When matrices become too large to fit in memory or when further performance gains are needed, distributed computing provides a scalable alternative. By dividing the workload among several nodes, distributed frameworks such as Hazelcast allow for parallel computation across a cluster, enabling the processing of large matrices that would be infeasible on a single system.

However, distributed matrix multiplication introduces additional complexities, such as network communication overhead, data upload latency, and memory management across nodes. These trade-offs must be carefully evaluated to determine whether distribution leads to practical performance improvements.

This paper explores these aspects by comparing distributed matrix multiplication in Java and Python with local methods including basic, parallel, and vectorized strategies. Metrics such as execution time, upload time, memory usage, and data transfer rates are used to assess performance and scalability.

For access to the source code visit the GitHub repository:
https://github.com/DenysKavkalo/Matrix-Multiplication.

2 Problem Statement

Matrix multiplication is a computationally intensive task, especially when dealing with very large matrices that exceed the memory capacity of a single machine. Traditional and parallel implementations on single nodes are limited in their ability to efficiently handle such large-scale problems due to hardware constraints.

This work proposes the design, implementation, and evaluation of a distributed matrix multiplication solution using a distributed computing framework. The system is developed in both Python and Java to highlight its versatility and to compare performance across different programming environments.

The main challenges addressed include:

- Efficiently distributing the matrix multiplication workload across multiple nodes.
- Managing matrices that do not fit into the memory of a single machine.
- Minimizing network overhead and data transfer times, which are inherent to distributed environments.
- Evaluating the scalability of the solution as matrix size increases.
- Comparing distributed performance against basic, parallel, and vectorized single-node implementations.
- Measuring resource utilization, including memory usage and number of nodes or workers involved.

The ultimate goal is to demonstrate how distributed systems can overcome the limitations of local computation to efficiently process large matrix operations, while analyzing the trade-offs in communication cost and resource allocation that come with distributed computation.

3 Methodology

In previous work (TASK 3. Vectorized and Parallel Matrix Multiplication), several implementations of matrix multiplication were developed exclusively in Java:

- A basic $O(n^3)$ algorithm implementing the standard nested loop multiplication.
- A vectorized algorithm exploiting SIMD instructions to optimize row and column operations.
- A parallel algorithm using multi-threading to distribute computations across multiple CPU cores.

These Java implementations provide a comprehensive baseline for sequential and parallel performance on a single machine. Building upon these results, the current work focuses on the design, implementation, and benchmarking of distributed matrix multiplication across 4 Hazelcast nodes using both Java and Python. The distributed system is deployed using Docker Compose, with each Hazelcast node running as a separate container connected via a dedicated bridge network (`hazelcast-net`). The matrix multiplication client, also containerized, connects to the Hazelcast cluster through this network to partition and execute the computation across the nodes.

Given that the sequential and parallel baseline implementations from TASK 3 are only available in Java, the comparison between local and distributed performance (TASK 3 vs. TASK 4) is conducted exclusively in the Java environment. This allows for a direct evaluation of how distributed execution improves upon traditional single-node approaches.

Subsequently, a cross-language comparison is performed between the Java and Python implementations of distributed matrix multiplication. Both clients connect to equivalent Hazelcast cluster deployments, enabling an analysis of performance, scalability, and implementation differences across languages within a consistent distributed framework.

Notably, the deployment architecture differs between the two implementations due to language-specific Hazelcast client behavior. In the Java setup, custom Docker containers are used to run four Hazelcast nodes along with a dedicated client container that coordinates the distributed multiplication. This setup provides fine-grained control over node behavior and client execution.

In contrast, the Python implementation uses the official Hazelcast Docker image to deploy the four cluster nodes, while worker containers—each identified by a unique environment variable—connect to the cluster to perform computations. This reflects the current operation model of the Hazelcast Python client, which interacts with the cluster differently than the Java client. Despite these architectural differences, both environments ensure fair benchmarking and allow for meaningful cross-language performance comparisons within a unified distributed framework.

3.1 Java Distributed Implementation

The distributed system developed in this project is primarily implemented in **Java** and structured as a **Maven** project. Its goal is to efficiently perform matrix multiplication at scale by distributing the computational load across multiple nodes using **Hazelcast**, and managing the execution environment with **Docker Compose** for reproducibility and scalability.

The system architecture includes several Hazelcast nodes responsible for computation, and a dedicated client node that coordinates task distribution and result collection. All components are containerized and launched using Docker Compose, eliminating the need for complex manual configurations.

The core of the implementation resides in the `MatrixBenchmark` class, located in the `com.example.benchmark` package. This class is responsible for connecting to the Hazelcast cluster, generating random matrices, partitioning the multiplication tasks, dispatching them to remote workers via `IExecutorService`, and collecting the partial results to reconstruct the final matrix. The benchmark supports various matrix sizes and logs key performance metrics during execution.

Each distributed task is represented by the `MatrixMultiplicationTask` class, which implements `Callable` and is `Serializable`, a requirement for transmitting tasks over the network. Each task instance carries a subset of matrix A, the full matrix B, and index information defining the row range to be processed. The result of each task is a submatrix, returned to the client for aggregation.

Performance metrics are recorded using the `MetricsLogger` class. This component writes detailed benchmarking data to a CSV file, including matrix size, total execution time, task dispatching and computation times, network transfer volume, and system resource usage (CPU and memory) before and after execution. These logs are saved in the shared `results/` directory, allowing external access without container inspection.

Additionally, the `Utils` class provides utility functions for generating random matrices, estimating memory usage, and retrieving the Java process's CPU utilization. These tools enable more precise monitoring of system behavior during benchmarks.

Deployment is orchestrated via `docker-compose.yml`, which defines four main services: three Hazelcast nodes (`hazelcast-node1`, `hazelcast-node2`, and `hazelcast-node3`, with the option of adding a fourth), and a client container (`matrix-client`). All nodes use the same `Dockerfile.hazelcast-node`, which launches a standard Hazelcast instance. The client is built using `Dockerfile.matrix-client`, which compiles the Java project and runs `MatrixBenchmark` upon startup. Service dependencies are managed by Docker Compose to ensure proper startup order. A shared volume `./results:/app/results` ensures benchmark results persist outside the containers.

Client connection settings to the Hazelcast cluster are configured in `hazelcast-client.xml` (located in `src/main/resources`), which defines discovery parameters, port configurations, and error handling strategies. This ensures the client reliably connects to all active cluster nodes over the Docker network.

3.2 Python Distributed Implementation

The distributed system was also re-implemented in **Python** to enable language-agnostic benchmarking and to demonstrate the flexibility of the Hazelcast-based architecture. This implementation mirrors the core functionality of the Java version—performing distributed matrix multiplication across a Hazelcast cluster—while leveraging Python tools and libraries, particularly **NumPy**, for numerical computation.

The system architecture comprises four Hazelcast nodes (`hazelcast1` through `hazelcast4`) and four stateless Python worker containers (`worker0` through `worker3`). All services are defined and orchestrated through `docker-compose.yml`, which configures a dedicated Docker network (`hazelcast-net`) to ensure seamless communication between containers. Workers dynamically connect to the Hazelcast cluster at runtime and continuously poll a distributed map for task assignments.

The matrix multiplication process is coordinated by a script running outside the Docker network—typically `coordinator.py` or `benchmark_distributed.py`. This script initializes two random matrices A and B, partitions matrix A into row blocks based on the number of workers, and uploads both matrices to the Hazelcast distributed map. Each worker retrieves its specific subtask from a designated key (`task_{id}`), performs the matrix block multiplication using NumPy, and stores the partial result back into Hazelcast under a corresponding key (`result_{id}`).

Worker logic is implemented in the `worker.py` script. Upon startup, each container initializes a Hazelcast client using its `WORKER_ID`, which is passed via environment variables. The worker then enters a polling loop, waiting for the assignment of a task. Once a task becomes available, it computes the result and uploads it back to the distributed map, after which it resumes waiting for the next task.

Performance evaluation is carried out by the `benchmark_distributed.py` script. For matrix sizes ranging from 10×10 to 3000×3000, this script performs the complete distributed computation while logging a variety of metrics: total execution time, matrix upload time, computation time, total bytes transferred, data transfer rate, CPU and memory usage (before and after), and the number of active workers. The results are stored in the `benchmark_distributed.csv` file for later analysis.

This Python implementation offers a flexible and extensible alternative to the Java-based system. It supports scaling to additional nodes or workers by simply modifying the Docker Compose configuration. Like the Java version, it leverages Hazelcast's distributed data structures and Docker's orchestration capabilities to create a reproducible, containerized distributed computing environment.

4 Environment and Execution

Both the Java and Python implementations rely on containerized environments orchestrated through Docker. Therefore, it is essential to have Docker Desktop running prior to execution. All experiments described in this report were conducted using **Docker Desktop 4.42.1 (196648)** on the following system:

- **Operating System:** Microsoft Windows 11 Home 64-bit
- **Laptop Model:** ASUS TUF Dash F15
- **CPU:** Intel Core i7-12650H (12th Gen, 10 cores: 6 P-cores + 4 E-cores, up to 4.7 GHz)
- **RAM:** 16 GB DDR5-4800 (dual-channel)
- **Storage:** 512 GB SSD M.2 NVMe PCIe 3.0
- **GPU:** NVIDIA GeForce RTX 3050 Ti Laptop GPU (4 GB GDDR6, up to 75W with Dynamic Boost)

4.1 Java Project Workflow

For the Java-based project, the recommended development environment is **IntelliJ IDEA**, configured with **Oracle OpenJDK 24.0.1** as the SDK. Once the project is opened in IntelliJ, the following steps should be performed:

1. Open the built-in terminal in IntelliJ or a system terminal located at the root of the project.
2. Run the Maven build process with:

   ```
   mvn clean package
   ```

3. Once the build completes, deploy the distributed system using Docker Compose:

   ```
   docker-compose up --build
   ```

This command builds and launches four Hazelcast node containers along with a client container responsible for executing the distributed matrix multiplication. Upon completion, a CSV file named `benchmark_results.csv` will be generated automatically in the `results/` directory located at the root of the project.

This file contains the benchmarking metrics captured during the execution, such as execution time, data transfer time, and other performance indicators relevant to the distributed computation.

4.2 Python Project Workflow

For the Python-based project, the recommended development environment is **PyCharm**, configured with a virtual environment using **Python 3.9**. Once the project is properly set up in PyCharm, the following steps should be followed:

1. Open a terminal within the activated Python virtual environment, ensuring that the current working directory is the root of the project.
2. Install the required dependencies by executing:

   ```
   pip install -r requirements.txt
   ```

3. Launch the Hazelcast cluster and worker containers using:

   ```
   docker-compose up --build
   ```

4. Keep this terminal running, as it maintains the execution of the Hazelcast nodes and worker processes.
5. Open a second terminal (also with the virtual environment activated and in the project root directory), and run the benchmark script:

   ```
   python benchmark_distributed.py
   ```

Upon completion, a CSV file named `benchmark_distributed.csv` will be generated in the root directory of the project. This file contains performance metrics gathered during the distributed matrix multiplication, including execution time and communication overhead.

It is recommended to minimize background processes during execution, as the Python implementation tends to be less efficient compared to its Java counterpart under similar conditions.

Important Notes

- Ensure that **only one Docker Compose stack** is running at a time. Running both the Java and Python Hazelcast clusters simultaneously may cause **port conflicts** or unstable behavior in Docker Desktop due to overlapping container configurations.

- Before switching between the Java and Python projects, make sure to fully stop and remove any active containers related to the previous project. This can be done by closing the corresponding terminals or by running:

  ```
  docker-compose down
  ```

  inside the project's root directory.

- Failure to isolate the Docker environments may lead to incorrect benchmark results or runtime errors, particularly due to conflicting usage of ports such as `5701-5704`.

5 Experiments

The experimental section is divided into two main parts. The first part corresponds to **Task 3**, where local matrix multiplication methods in Java are evaluated using benchmarking tools. The second part focuses on the performance of a **distributed matrix multiplication system**, implemented both in Java and Python using Docker and Hazelcast to simulate distributed computation in a single-host environment.

5.1 Local Matrix Multiplication Benchmarks in Java (Task 3)

This task evaluates and compares the performance of three different matrix multiplication strategies in Java: Basic, Parallel, and Vectorized. The experiments were conducted using three matrix sizes (10×10, 100×100, 1000×1000) and different thread counts (1, 2, and 4) to analyze scalability and efficiency. To ensure precise measurement, the Java Microbenchmark Harness (JMH) framework was used. Although JMH introduces some benchmarking overhead, it provides reliable execution time and memory allocation metrics, which are essential for microbenchmarking.

The **Basic method** uses a standard triple-loop implementation with cubic time complexity, serving as a baseline. The **Parallel method** employs multithreading by distributing the computation of matrix rows across multiple threads to reduce execution time. The **Optimized method** applies loop unrolling to reduce control overhead. Finally, the **Vectorized method** leverages Java's parallel streams (IntStream.parallel) to process rows concurrently using modern Java parallelism features. Each configuration was executed multiple times to measure execution time (ms/op) and memory allocation rate (MB/s), allowing performance comparisons across methods, matrix sizes, and thread counts.

5.2 Distributed Matrix Multiplication in Java

The second part of the experiments evaluates a distributed matrix multiplication system implemented in Java using Hazelcast and Docker containers. This setup simulates a distributed architecture by deploying multiple worker nodes in isolated containers, coordinated by a Hazelcast client.

The benchmarking logic is encapsulated in the `MatrixBenchmark` class, which performs multiplications for increasing matrix sizes: 10, 100, 500, 1000, and 3000. The client uses Hazelcast's `IExecutorService` to dispatch matrix multiplication tasks to distributed worker nodes. Each worker is responsible for computing a specific subset of rows of the result matrix, determined by partitioning the matrix evenly across available nodes.

During each benchmark run, several metrics are collected:

- Memory usage before and after execution (in MB), to estimate RAM usage of the coordinator.
- CPU load before and after, to gauge system resource utilization.
- Upload time, representing the latency in task distribution.
- Computation time, reflecting the duration required by workers to complete their assigned tasks.
- Total execution time, the sum of upload and computation times.
- Network transfer volume and simulated transfer rate, approximating communication costs.

All performance data is logged into a CSV file for further analysis. Although the setup is hosted on a single machine, Docker isolation enables a realistic approximation of a distributed system, which could be easily scaled to a multi-host environment in the future.

Unlike the local benchmarking scenario in Task 3, the distributed environment poses challenges that make traditional microbenchmarking frameworks like JMH unsuitable. JMH is designed for fine-grained benchmarking within a single JVM and cannot accurately account for inter-node communication latency, task dispatch overhead, or Docker-induced isolation effects. Instead, the distributed system requires custom instrumentation to capture higher-level metrics such as network transfer times, coordination overhead, and container-level resource usage, which are critical for evaluating real-world distributed performance.

5.3 Distributed Matrix Multiplication in Python

Complementing the Java-based distributed system, a Python implementation was also benchmarked under the same simulated conditions using Docker and Hazelcast. The objective was to replicate the distributed execution logic while using the Python ecosystem and assess comparative performance and resource usage.

The main benchmarking script, `benchmark_distributed.py`, executes matrix multiplications for varying sizes and measures key metrics: memory usage and CPU load (collected via `psutil`), upload time (via `time.perf_counter()`), and the size and rate of simulated network transfers.

Computation time is measured from the moment workers start processing tasks to the time they return results. Total execution time, combining upload and compute durations, reflects the overall efficiency of the distributed operation. As with the Java version, all results are saved in a CSV file.

This Python-based setup mirrors the Java system's architecture and benchmarking philosophy while leveraging Python's flexibility. Docker containerization ensures reproducibility and component isolation, making the system a reliable tool for evaluating distributed matrix processing in a controlled, single-host environment.

6 Results

Table 1: Basic Multiplication in Java from TASK 3

Metric	**Threads**	**10x10**	**100x100**	**1000x1000**
Execution Time (ms/op)	1	0.010227	0.223589	1194.134230
	2	0.010229	0.224028	1240.426365
	4	0.010227	0.219594	1215.684212
Memory Allocation Rate (MB/s)	1	11.434512	43.223105	470.004230
	2	11.416561	33.744033	413.012053
	4	11.477078	34.901871	425.739102

Table 2: Parallel Multiplication in Java from TASK 3

Metric	**Threads**	**10x10**	**100x100**	**1000x1000**
Execution Time (ms/op)	1	0.006193	0.115076	1167.394122
	2	0.006071	0.116512	1204.151875
	4	0.006163	0.118478	1211.285778
Memory Allocation Rate (MB/s)	1	10.578634	43.312708	463.445201
	2	10.534269	34.274664	412.552143
	4	10.642417	34.785693	423.478328

Table 3: Vectorized Multiplication in Java from TASK 3

Metric	**Threads**	**10x10**	**100x100**	**1000x1000**
Execution Time (ms/op)	1	0.011349	0.223958	1194.725304
	2	0.011610	0.225400	1240.333427
	4	0.011850	0.219620	1215.655720
Memory Allocation Rate (MB/s)	1	10.738054	43.107010	470.024453
	2	10.749760	33.755960	413.097552
	4	10.779758	34.853804	425.735993

Table 4: Distributed Multiplication in Java + Hazelcast from TASK 4

Metric	**10x10**	**100x100**	**500x500**	**1000x1000**	**3000x3000**
Total Time (ms)	165	64	176	1789	157766
Upload Time (ms)	35	15	79	203	2056
Execution Time (ms/op)	0.1650	0.000064	0.00000141	0.00000179	0.00000584
Transfer Rate (MB/s)	0.04347	10.1052	48.2263	75.1294	66.7911
Memory Used (MB)	1	3	9	79	721
Memory Allocation Rate (MB/s)	6.06	46.88	51.14	44.16	4.57
Number of Workers	4	4	4	4	4

Table 5: Distributed Multiplication in Python + Hazelcast from TASK 4

Metric	**10x10**	**100x100**	**500x500**	**1000x1000**	**3000x3000**
Total Time (ms)	2935.57	35410.39	32142.38	41217.84	96701.42
Upload Time (ms)	8.88	15225.17	16194.30	18683.21	50899.74
Execution Time (ms/op)	29.36	3.54	0.13	0.041	0.0107
Transfer Rate (MB/s)	0.1718	0.0100	0.2356	0.8167	2.6980
Memory Used (MB)	3.81	1.90	22.54	26.85	519.35
Memory Allocation Rate (MB/s)	1.30	0.054	0.70	0.65	5.37
Number of Workers	4	4	4	4	4

6.1 Comparison and Observations

The three multiplication methods—Basic, Parallel, and Vectorized—were tested across different matrix sizes (10x10, 100x100, and 1000x1000) and the corresponding execution times, memory allocation rates, and the number of threads used. Below is a comparative analysis of the results:

6.1.1 Comparison: Local vs Distributed Multiplication in Java

This section presents a comparison between the performance of **local matrix multiplication** in Java (Tables 1, 2, and 3) and the **distributed multiplication using Java + Hazelcast** (Table 4). The analysis focuses on two primary metrics: **Execution Time (ms/op)** and **Memory Allocation Rate (MB/s)**.

Execution Time (ms/op)

For small matrices like `10x10`, local parallel implementations are significantly faster. For example, the best local time achieved is approximately `0.0061 ms/op` (parallel), whereas the distributed version takes `0.1650 ms/op`, making it about **27 times slower**. This performance loss is attributed to overheads associated with task distribution, data serialization, and network communication, which dominate when matrix sizes are small.

In the case of medium-sized matrices like `100x100`, an unusual result appears: the distributed version reports an execution time per operation of only `0.000064 ms/op`, which is surprisingly low. This indicates a potential inconsistency or miscalculation, likely due to dividing total time by the number of matrix elements (n^2) without considering actual computation time per thread or per task.

For larger matrices such as `1000x1000` and `3000x3000`, the distributed execution times are extremely small when expressed per operation (on the order of nanoseconds), which again suggests an inconsistency in the metric. A revision of the calculation method is recommended, possibly shifting from per-element to per-block timing, to better reflect actual performance.

Memory Allocation Rate (MB/s)

In terms of memory efficiency, local implementations outperform distributed ones in most cases. For `10x10` matrices, vectorized local multiplication reaches a memory allocation rate of up to `11.47 MB/s`, while the distributed version achieves only `6.06 MB/s`. For `100x100`, both methods are comparable: local (parallel) achieves `43.31 MB/s`, and distributed achieves `46.88 MB/s`, showing that distributed execution can be competitive for intermediate matrix sizes.

However, for large matrices like `1000x1000`, the local vectorized version allocates memory at `470 MB/s`, while the distributed method drops to just `44.16 MB/s`, indicating a **10x drop in efficiency**. This decline in distributed memory performance becomes even more pronounced with `3000x3000` matrices, where the rate falls to `4.57 MB/s`.

In contrast, local Java implementations—especially parallel and vectorized ones—are consistently faster and more memory-efficient across all matrix sizes tested. They avoid distribution overhead and benefit from optimized in-memory computation and thread management.

6.1.2 Comparison: Java vs Python Distributed Multiplication

The distributed matrix multiplication implementations in Java and Python running on Hazelcast show significant performance differences. Java consistently outperforms Python in execution time per operation across all tested matrix sizes. For example, at 10×10, Java completes operations in approximately `0.165 ms/op` compared to Python's `29.36 ms/op`, nearly two orders of magnitude slower. This gap narrows slightly for larger matrices (1000×1000 and 3000×3000), where Python is slower by a factor of 5–10x, but remains substantial.

This performance difference stems from multiple factors: Java's native compilation, efficient thread management, and tighter integration with Hazelcast reduce overheads in serialization, deserialization, and network communication. Python's interpreter overhead, less optimized threading, and higher serialization costs (especially in distributed communication) contribute to slower execution.

Memory allocation rates further highlight Java's advantage. At 1000×1000, Java achieves `44.16 MB/s` compared to Python's `0.65 MB/s`, showing a drastic difference in how efficiently each environment handles memory during distributed computation. Interestingly, at 3000×3000, Python improves to `5.37 MB/s`, but still lags behind Java's `4.57 MB/s`, despite Java's memory allocation rate being much lower than its local vectorized counterpart. This suggests that both implementations face scaling challenges, but Python's overhead remains more pronounced.

Upload Time and Transfer Rate Upload time, defined as the interval between sending input data and receiving acknowledgment of successful distribution to Hazelcast workers, is a critical metric that impacts distributed execution efficiency. Java consistently shows lower upload times than Python across all matrix sizes, implying more efficient data handling and network communication. For example, at smaller sizes, Python's upload time dominates total execution time, severely limiting speedup potential.

Transfer rate (MB/s), computed as the total bytes of input and output matrix data divided by upload time, further reflects the efficiency of data serialization and network transfer. Java maintains higher transfer rates than Python, indicating superior serialization mechanisms and less overhead in network I/O. However, as matrix sizes increase, both languages experience reduced transfer rates due to larger data volumes and possible network congestion in the Docker single-host environment.

Memory Usage Memory consumption is generally higher in Python, particularly for large matrices. At 3000×3000, Python's memory usage exceeds `519 MB` while Java's is about `721 MB`, yet Java completes computations significantly faster. This suggests that Python's memory management and garbage collection are less efficient in the distributed context, leading to higher runtime memory footprints without proportional performance gains.

These observations suggest that Java + Hazelcast is better suited for high-performance distributed matrix computations in this setup, due to more efficient execution, memory management, and communication. Python's simplicity and ease of development come at the cost of runtime overheads that become critical at scale.

6.2 Metric Calculation Methodology

The following metrics were calculated using custom instrumentation embedded in the Java and Python clients, combined with external tools such as JMH for local tests and internal timers (e.g., `System.nanoTime()`, `Runtime.getRuntime()`) for distributed runs. The methodology per metric is detailed below:

- **Execution Time (ms/op)**: For local tests, obtained from JMH output as average time per operation. For distributed versions, calculated as total execution time divided by the number of computed elements (n^2). This may underestimate real computation cost in the presence of communication delays or task batching.
- **Memory Allocation Rate (MB/s)**: Computed as the total allocated memory (heap used before and after the operation, in megabytes) divided by the operation time in seconds.
- **Transfer Rate (MB/s)**: Defined as the amount of matrix data (input + output) transferred during the job, divided by the measured upload time. Matrix sizes were used to approximate byte volume ($n^2 \times$ `sizeof(double)`).
- **Upload Time**: Measured as the time elapsed between sending the data from the client and receiving acknowledgment of successful distribution to Hazelcast workers.
- **Memory Used (MB)**: Snapshot of used heap memory just after execution, using `Runtime.getRuntime().totalMemory() - freeMemory()` for Java, and `psutil.Process.memory_info().rss` in Python.
- **Number of Workers**: Fixed to 4 for all distributed experiments to ensure fair comparison and resource consistency across executions.

Note: All distributed tests were executed on a single host running Docker containers to isolate network effects and ensure stable timing.

7 Conclusion

The comparative analysis between local and distributed matrix multiplication implementations highlights significant trade-offs. While local Java implementations—especially parallel and vectorized—achieve superior execution times and memory efficiency for all tested matrix sizes, distributed multiplication incurs overheads from data serialization, network communication, and task distribution. These overheads make distributed computation less efficient for small matrices, though it shows potential competitiveness at intermediate sizes. However, current metrics indicate inconsistencies for very large matrices, suggesting that further refinement in measurement methodology is needed to better capture the distributed performance profile.

When comparing distributed implementations in Java and Python using Hazelcast, Java consistently outperforms Python in execution time, memory allocation rate, upload time, and transfer rate. Java's advantages stem from its native compilation, optimized thread management, and tighter integration with Hazelcast, resulting in more efficient serialization and network handling. Python, while offering easier development, exhibits substantially higher runtime overhead and memory consumption, particularly as matrix sizes grow. Consequently, Java + Hazelcast proves to be the more effective choice for scalable distributed matrix multiplication in this setup, although both implementations face challenges in handling large data volumes efficiently.

8 References

[1] Denys Kavkalo Gumeniuk. *Matrix-Multiplication*. GitHub repository, 2025. Available at: `https://github.com/DenysKavkalo/Matrix-Multiplication`