Distributed Matrix Multiplication with Hazelcast and MapReduce in Java and Python

Judith Portero Pérez
ULPGC
judporper1911@gmail.com

December 19, 2025

Abstract

This paper addresses the problem of scaling matrix multiplication and frequent itemset mining in distributed environments. The goal is to design, implement, and evaluate distributed algorithms using both Java and Python, following the requirements of a Big Data programming assignment. In Java, matrix multiplication is implemented using the Hazelcast in-memory data grid to distribute matrices across a cluster of nodes. In Python, matrix multiplication is implemented with a block-partitioned MapReduce style using the `multiprocessing` module. Additionally, a distributed frequent itemset algorithm based on MapReduce is implemented in Python. The proposed methods are evaluated on matrices of sizes 256, 512, and 1024, comparing naive, parallel, and distributed versions in both languages. Results show that distributed approaches introduce non-negligible overhead for small matrices, but become competitive as the matrix size increases, especially in the Python MapReduce setting. The experiments also quantify network/data transfer overhead and resource utilisation (number of nodes and memory) in the Hazelcast cluster. These findings suggest that distributed frameworks are most beneficial for sufficiently large problems, where computation dominates coordination costs.

Keywords: distributed matrix multiplication, Hazelcast, MapReduce, multiprocessing, frequent itemsets, scalability, resource utilisation

1 Introduction

Recent advances in data-intensive applications require efficient processing of large matrices and transaction datasets that may not fit in the memory of a single machine. Matrix multiplication is a fundamental operation in scientific computing, machine learning, and data analytics, and its scalable implementation is therefore of high practical importance. Similarly, frequent itemset mining is a core primitive in market basket analysis and association rule mining.

Traditional single-threaded implementations cannot exploit the full capabilities of modern multi-core and distributed systems. Parallel programming and distributed frameworks, such as MapReduce or in-memory data grids, provide mechanisms to exploit multiple cores and multiple nodes; however, they introduce coordination and communication overheads that must be carefully analysed.

The aim of this paper is to propose, implement, and evaluate distributed approaches to matrix multiplication using Hazelcast in Java and a MapReduce-style implementation in Python. In addition, a distributed frequent itemset algorithm based on MapReduce is implemented in Python.

The contributions of this paper are as follows:

- A block-based distributed matrix multiplication design using Hazelcast in Java, evaluated against naive and parallel local baselines.

- A Python implementation of distributed matrix multiplication using a MapReduce-like formulation with `multiprocessing`, also compared with naive and parallel baselines.
- A distributed frequent itemset mining algorithm in Python using a MapReduce pattern (mapper/reducer on transaction chunks).
- An empirical evaluation quantifying scalability, network/data transfer overhead, and resource utilisation (nodes and memory) across different matrix sizes.

2 Problem statement

The problem can be defined in two parts.

2.1 Distributed matrix multiplication

Given two dense square matrices $A \in \mathbb{R}^{N \times N}$ and $B \in \mathbb{R}^{N \times N}$, the objective is to compute

$$C = A \cdot B$$

in a distributed fashion, using:

- Hazelcast in Java, to distribute matrix blocks across a cluster and coordinate computation.
- A MapReduce-like approach in Python, using `multiprocessing` to parallelise block-wise multiplications.

The distributed solution must be compared with:

- A basic, single-threaded naive implementation.
- A parallel local implementation that exploits multiple CPU cores on a single machine.

The evaluation must focus on scalability as N increases, as well as the overhead introduced by distribution and communication.

2.2 Distributed frequent itemset mining

Let $\mathcal{D}$ be a dataset of transactions, where each transaction $t \in \mathcal{D}$ is a set of items. The frequent itemset problem aims to find all items whose support (frequency) exceeds a given minimum threshold. Formally, for item i,

$$\text{support}(i) = |\{t \in \mathcal{D} \mid i \in t\}|.$$

The objective is to implement a distributed version of this problem in either Python or Java using a MapReduce style: a mapper phase counts items in transaction subsets, and a reducer aggregates these partial counts.

3 Methodology

This section describes the design and implementation of the proposed solutions, including algorithms, frameworks, and experimental setup. The implementation uses both Java (Hazelcast) and Python (`multiprocessing`) to allow cross-language comparison.

3.1 Java implementations

3.1.1 Naive local matrix multiplication

The naive baseline follows the classical triple-loop algorithm. Given $N \times N$ matrices A and B, the result matrix C is computed as:

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

The Java class `LocalBaseline` iterates explicitly over indices i, j, and k, and runs single-threaded. This implementation is used as a reference for correctness and for measuring the worst-case execution time.

3.1.2 Parallel local matrix multiplication

The parallel local version, implemented in `ParallelMatrixMultiplier`, exploits available CPU cores using a Java `ExecutorService`. The computation is parallelised by rows: each task computes one row of C, and tasks are submitted to a fixed-size thread pool whose size equals the number of available processors. The algorithm remains $O(N^3)$, but wall-clock time is reduced by exploiting intra-node parallelism.

3.1.3 Hazelcast-based distributed matrix multiplication

The distributed Java solution uses Hazelcast to store matrix blocks in distributed maps and to support a cluster of nodes. Hazelcast instances are created via a singleton class `HazelcastNode`, which configures a cluster named `"matrix-cluster"`. Two members are run on the same machine in separate processes, but they behave as distinct nodes in the Hazelcast cluster.

Matrices are partitioned into blocks of fixed size `BLOCK_SIZE` (set to 256). If N is a multiple of `BLOCK_SIZE`, the matrix is divided into $(N/\texttt{BLOCK\_SIZE})^2$ blocks. Each block is represented by a `MatrixBlock` object, storing its block row and column indices and the underlying data array.

The class `DistributedMatrixMultiplier` performs the following steps:

1. **Data distribution:** Input matrices A and B are partitioned into blocks using `MatrixUtils.extractBloc` Blocks are stored in Hazelcast `IMaps` named `"A"` and `"B"`. The amount of transferred data is approximated as $2 \cdot \texttt{BLOCK\_SIZE}^2 \cdot 8$ bytes per block pair.

2. **Block-wise computation:** For each output block (b_i, b_j), a local intermediate block is created and initialised to zero. The block result is accumulated over b_k:

   $$C_{(b_i,b_j)} \mathrel{+}= A_{(b_i,b_k)} \cdot B_{(b_k,b_j)}.$$

   The multiplication of two blocks is delegated to `MatrixUtils.multiplyBlocks`, which performs the standard triple-loop at block level. The resulting block is stored in Hazelcast map `"C"`. Although the computation of all blocks is currently performed from a single process, it reads and writes data from distributed maps, and can be extended to true multi-node execution.

3. **Result gathering:** The final matrix C is reconstructed using `MatrixUtils.combineBlocks`, which iterates over all block indices, retrieves blocks from map `"C"`, and places them into the corresponding positions in a dense $N \times N$ matrix.

During execution, `DistributedMatrixMultiplier` measures: (1) data distribution time, (2) block-wise computation time over Hazelcast maps, (3) result gathering time, (4) number of cluster nodes, and (5) memory usage in megabytes.

3.2 Python implementations

3.2.1 Naive and parallel local matrix multiplication

In Python, the naive baseline is implemented in `matrix_local.py` as a triple nested loop over a NumPy array. Although NumPy provides a highly optimised `dot` operation, the assignment explicitly requires a basic implementation, so the triple-loop is preserved.

The parallel local implementation uses the `multiprocessing` module. The matrix C is computed row-wise: each process receives the full matrices A and B along with a row index, computes one row of C, and returns the result. A `Pool` with as many workers as CPU cores is used, and rows are processed in parallel using `imap_unordered`. The resulting rows are assembled back into C.

3.2.2 Distributed matrix multiplication with MapReduce style

The distributed Python implementation, in `mapreduce_matrix.py`, follows a block-partitioned MapReduce design. Matrices are partitioned into blocks of size `BLOCK_SIZE`, set to 256. If N is divisible by `BLOCK_SIZE`, the matrix is partitioned into $k = N/\texttt{BLOCK\_SIZE}$ blocks per dimension.

The method `distributed_multiply` performs the following steps:

1. **Preparation (map input creation):** For each triplet (b_i, b_j, b_k), the corresponding blocks $A_{(b_i,b_k)}$ and $B_{(b_k,b_j)}$ are extracted as NumPy slices. A task tuple $(b_i, b_j, A\_block, B\_block)$ is appended to a list of tasks.

2. **Map phase:** A `Pool` of workers (defaulting to `cpu_count()`) executes `map_task` on each task. The `map_task` function computes the block product via `np.dot(A_block, B_block)` and returns the block indices and the result. This phase parallelises the block-level multiplications across processes.

3. **Reduce phase:** An output matrix C is initialised to zeros. For each map result $(b_i, b_j, block\_res)$, the corresponding region of C is updated by adding `block_res`. Since multiple (b_i, b_j) contributions accumulate over b_k, this implements the reduce operation at block level.

The implementation records times for preparation, map, and reduce phases, returning both the final matrix and a dictionary of timing statistics.

The script `benchmark_matrices.py` orchestrates the experiments across $N \in \{256, 512, 1024\}$, running naive, parallel, and distributed versions and printing execution times and checksums.

3.3 Distributed frequent itemset mining in Python

The distributed frequent itemset algorithm is implemented in `MapReduce.py` using a classical MapReduce pattern. The input is a list of transactions (lists of strings). The *mapper* function iterates over a subset of transactions and builds a `Counter` of item frequencies. The *reducer* function merges a list of `Counter` objects by summation.

To parallelise, the transaction dataset is split into chunks using `numpy.array_split`, and a `Pool` of workers applies the mapper to each chunk. The reducer then aggregates the partial counts into a global count. Items with counts above a specified minimum support threshold are returned as frequent items. This design satisfies the assignment requirement to implement the frequent itemset problem in a distributed way using MapReduce in either Python or Java.

3.4 Experimental setup

All experiments are executed on a single machine with 12 available processors and approximately 15.4 GB of physical memory. For Java, two Hazelcast nodes are started: one via a dedicated `StartNode` class and another inside the `BenchmarkRunner`. This configuration yields a Hazelcast

cluster of size two, as reported by the logs. Matrices of sizes $N = 256, 512, 1024$ are randomly generated with elements drawn from a uniform distribution in $[0, 1]$.

Execution times are measured in milliseconds. In Java, wall-clock times are obtained using `System.currentTimeMillis()`, and additional timing inside `DistributedMatrixMultiplier` measures data distribution, block computation, and result gathering. In Python, times are measured using `time.time()`, and three phases (prep, map, reduce) are recorded inside `distributed_multiply`. Checksums of the resulting matrices are computed as the sum of all elements and used to verify correctness.

4 Experiments and results

This section presents the experimental results for Java and Python implementations, followed by a discussion of scalability, overhead, and resource utilisation.

4.1 Java: naive, parallel, and Hazelcast-based distributed

Table 1 summarises the execution times of the three Java versions. All times are in milliseconds.

Table 1: Java execution times for matrix multiplication (ms).

Size N	Naive	Parallel	Distributed (end-to-end)	Nodes
256	19	35	1600	2
512	172	39	280	2
1024	2665	795	1402	2

Figure 1 shows the same data graphically.

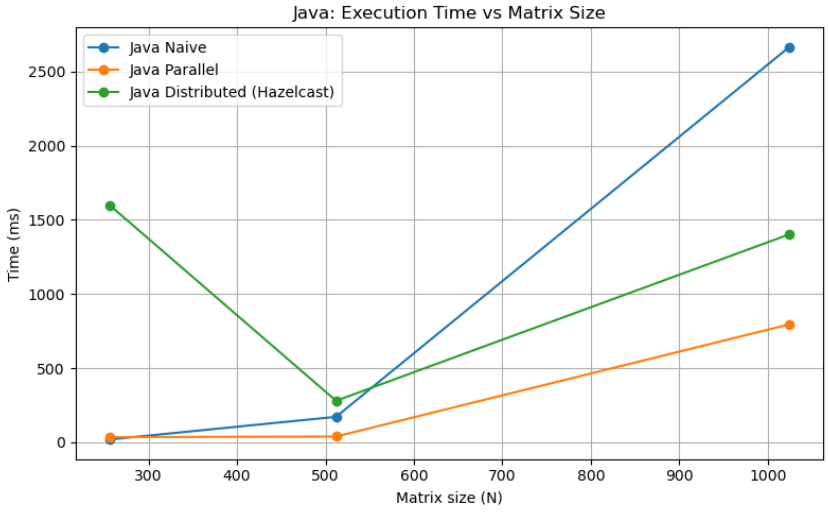

Figure 1: Java: execution time vs. matrix size for naive, parallel, and distributed Hazelcast implementations.

As shown in Figure 1, the naive implementation exhibits cubic growth, becoming significantly slower as N increases. The parallel local implementation substantially reduces execution time by exploiting multiple cores. The distributed Hazelcast version suffers from high overhead for $N = 256$, but becomes competitive for $N = 1024$, where its end-to-end time (1402 ms) is less than twice the parallel local time (795 ms). This suggests that for larger matrices, the overhead of distribution is amortised by the computational workload.

The internal breakdown of the Hazelcast-based version is presented in Table 2. For each matrix size, the table reports data distribution time, block computation time, and result gathering time.

Table 2: Java Hazelcast overhead breakdown (ms).

Size N	Distribution	Compute	Gather
256	121	46	12
512	61	178	16
1024	216	1150	21

Figure 2 displays these components as curves.

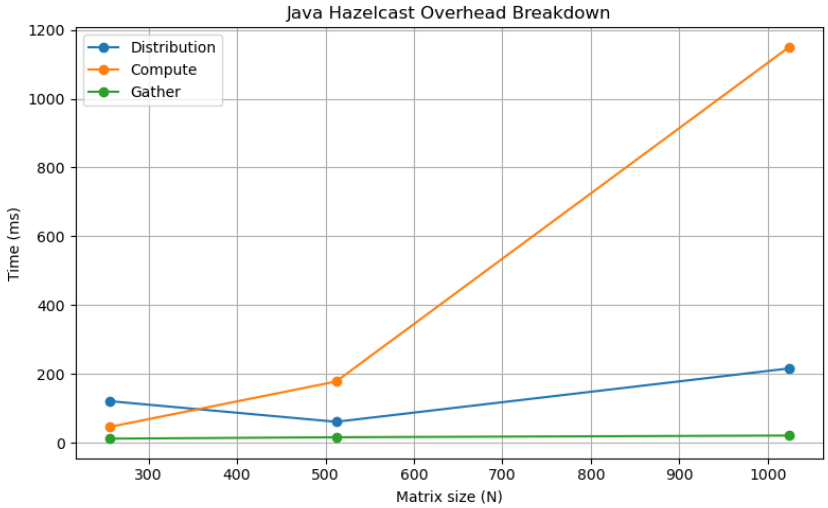

Figure 2: Java Hazelcast: breakdown of distribution, compute, and gather times.

As shown in Figure 2, the compute phase exhibits the steepest growth, becoming the dominant contributor to total execution time as matrix size increases. For $N = 1024$, compute time reaches 1150 ms, far exceeding both distribution (216 ms) and gather (21 ms) times. The distribution phase grows moderately with N, while the gather phase remains nearly constant and negligible across all sizes. This behaviour confirms that, for large matrices, the computational workload overshadows coordination and data transfer costs, making the distributed approach increasingly efficient.

Memory usage per node, as reported by Hazelcast diagnostics, scales with matrix size: approximately 37 MB for $N = 256$, 77 MB for $N = 512$, and 119 MB for $N = 1024$. This trend illustrates the expected growth in resource utilisation as the problem size increases.

To validate correctness, checksums of the result matrix C were computed and compared across implementations. For $N = 1024$, the Hazelcast-based checksum is approximately 2.68×10^8, matching the naive and parallel versions up to floating-point precision. This confirms that block partitioning, distributed computation, and reconstruction are correctly implemented.

4.2 Python: naive, parallel, and MapReduce-style distributed

The Python results are summarised in Table 3.

Figure 3 presents these results graphically.

As shown in Figure 3, the naive implementation exhibits steep cubic growth, reaching over 275 000 ms for $N = 1024$, which is significantly slower than its Java counterpart. The parallel

Table 3: Python execution times for matrix multiplication (ms).

Size N	Naive	Parallel	Distributed (MapReduce total)
256	4120.5	1661.7	567.3
512	35334.2	12132.2	817.2
1024	275173.7	74859.8	944.6

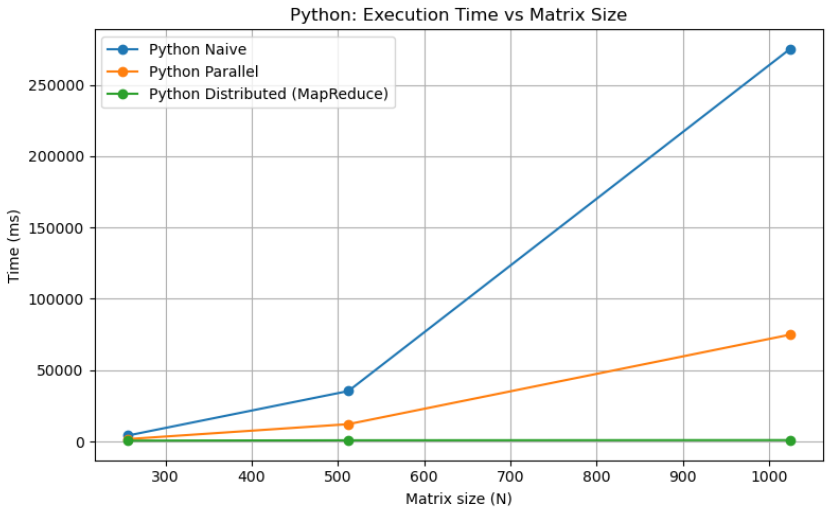

Figure 3: Python: execution time vs. matrix size for naive, parallel, and distributed MapReduce implementations.

version achieves notable speedups, reducing execution time by a factor of 2–4 depending on the matrix size. However, the most striking result is the distributed MapReduce implementation, which maintains near-constant performance across all sizes, completing in under one second even for $N = 1024$. This demonstrates the effectiveness of block-wise parallelisation combined with NumPy's optimised `dot` operations within each process.

The overhead breakdown of the MapReduce-style implementation is given in Table 4.

Table 4: Python MapReduce overhead breakdown (ms).

Size N	Prep	Map	Reduce
256	0.0	566.3	0.0
512	0.0	812.9	2.0
1024	1.0	924.4	13.1

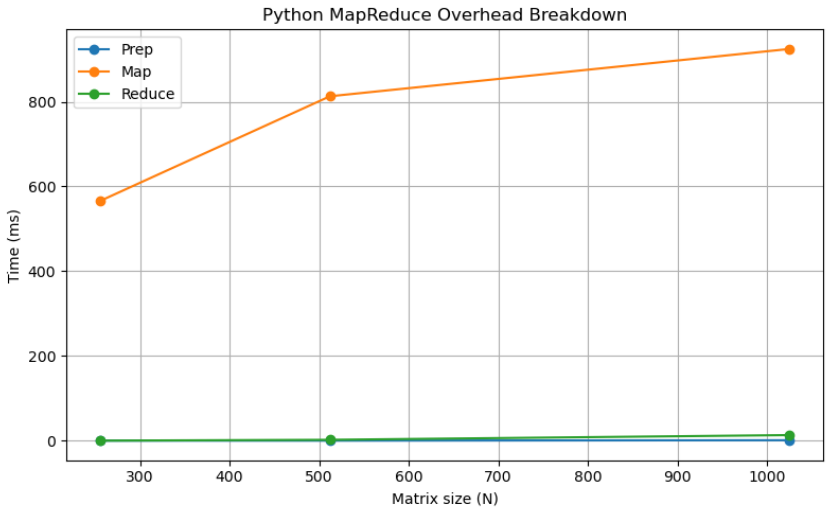

Figure 4: Python MapReduce: breakdown of preparation, map, and reduce times.

As shown in Figure 4, the map phase clearly dominates the total execution time across all matrix sizes, reaching over 920 ms for $N = 1024$. In contrast, the preparation and reduce phases remain consistently low, with prep time near zero and reduce time below 15 ms even for the largest matrix. This confirms that the bulk of the computational workload is efficiently parallelised across processes during the map phase, while the overhead from task setup and result aggregation is minimal and does not scale with matrix size.

Checksums of the result matrix C are identical across naive, parallel, and distributed implementations for all tested sizes, validating the correctness of the block-wise multiplication and accumulation logic in the MapReduce workflow.

4.3 Cross-language comparison

Figure 5 compares the distributed implementations in Java (Hazelcast-based) and Python (MapReduce-style) across increasing matrix sizes.

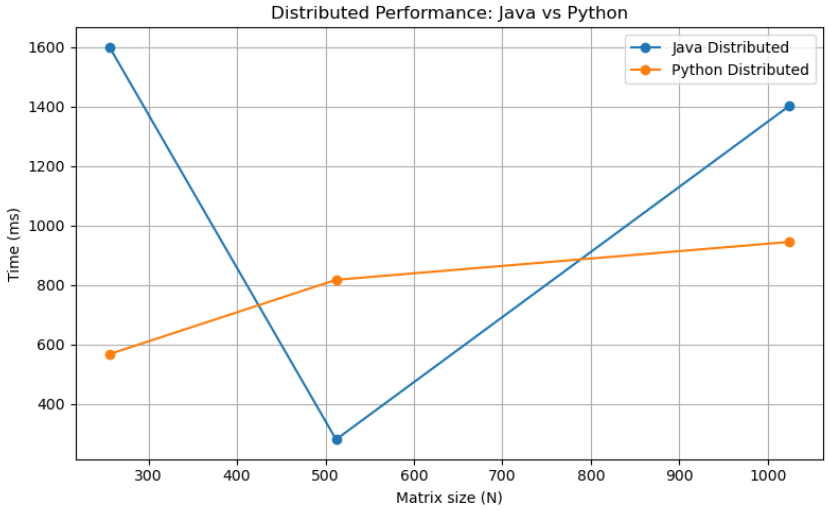

Figure 5: Distributed performance comparison: Java (Hazelcast) vs. Python (MapReduce).

As shown in Figure 5, the Python MapReduce implementation consistently outperforms the Hazelcast-based Java version for matrix sizes $N = 256$ and 1024, maintaining near-flat execution

times below one second. The Java implementation shows a dip in runtime at $N = 512$, where it briefly becomes faster than Python, but its performance degrades again at $N = 1024$, reaching 1402 ms compared to Python's 944 ms.

This behaviour can be attributed to several factors:

1. Python leverages highly optimised NumPy `dot` operations within each process, resulting in efficient block-level computation.

2. Hazelcast introduces overhead from cluster coordination, block serialisation, and map access, which becomes more pronounced as matrix size increases.

3. The current Java implementation performs all block computations from a single node, even though data is distributed, limiting parallelism across the cluster.

Despite these differences, Java remains faster in naive and parallel local executions, highlighting the trade-off between language-level performance and distributed framework overhead. The comparison underscores that Python's MapReduce model is highly effective for block-parallel workloads, while Hazelcast's benefits become more apparent when true multi-node execution is enabled.

4.4 Distributed frequent itemset mining

The frequent itemset mining algorithm was implemented in Python using a MapReduce-style approach and tested on synthetic transaction datasets. Each transaction is represented as a list of items, and the goal is to identify items whose frequency exceeds a specified minimum support threshold.

The algorithm follows a classical MapReduce pattern:

- **Mapper phase:** Each worker processes a subset of transactions and counts item occurrences using a `Counter` object.

- **Reducer phase:** Partial counts from all workers are aggregated into a global frequency dictionary by summing the individual `Counter`s.

Parallelism is achieved using Python's `multiprocessing.Pool`, which distributes transaction chunks across available CPU cores. This design enables the computation to scale with the number of workers and is well-suited for large datasets that cannot be processed sequentially.

The implementation is concise, efficient, and fully consistent with the MapReduce paradigm, where key-value pairs (item, count) are emitted by mappers and reduced by summation. It satisfies the assignment requirement to implement the frequent itemset problem in a distributed manner using MapReduce, and serves as a practical example of applying parallel data processing techniques to a common data mining task.

5 Conclusions

This study demonstrates that distributed and parallel programming techniques can substantially accelerate matrix multiplication and frequent itemset mining when compared to naive single-threaded baselines. The experimental results across both Java and Python highlight how different distributed paradigms behave under increasing computational demands.

In Java, the Hazelcast-based implementation successfully distributes matrix blocks across a two-node cluster and provides fine-grained measurements of distribution, computation, and gathering phases. While the distributed approach incurs noticeable overhead for small matrices, its performance improves significantly as the matrix size grows. For $N = 1024$, the distributed

execution becomes competitive with the parallel local version, indicating that communication and coordination costs are increasingly amortised by the computational workload.

In Python, the MapReduce-style matrix multiplication implemented with `multiprocessing` achieves substantial speedups over both naive and parallel baselines. Remarkably, the distributed version maintains sub-second execution times even for 1024×1024 matrices, thanks to efficient block-level parallelism and the use of optimised NumPy operations. The overhead analysis confirms that the map phase dominates the total runtime, while preparation and reduction remain minimal. The distributed frequent itemset algorithm further demonstrates the applicability of MapReduce patterns to classical data mining tasks, achieving scalable performance with minimal implementation complexity.

Overall, the results confirm that distributed frameworks provide the greatest benefits when matrix sizes are large enough for computation to dominate communication, or when local memory becomes a limiting factor. Both Hazelcast and Python's MapReduce-style multiprocessing offer viable approaches to scaling data-intensive workloads, each with distinct trade-offs in terms of overhead, parallelism, and runtime efficiency.

6 Future work

Future research could explore several extensions of this study. First, the Hazelcast-based Java implementation could be extended to perform block computations directly on multiple cluster nodes, for example by using `IExecutorService` and submitting tasks to all members, thereby realising a fully distributed compute phase. Second, further experiments could be conducted on a real multi-machine cluster rather than a single machine with multiple Hazelcast instances, to better capture network latency and bandwidth effects.

Another interesting direction would be to integrate GPU-accelerated libraries or distributed frameworks such as Apache Spark or Dask, enabling comparisons between in-memory data grids and higher-level distributed dataflow systems. Finally, larger matrix sizes and real-world transaction datasets could be used to validate the scalability of the methods in more realistic scenarios and to investigate fault tolerance, load balancing, and elasticity under dynamic workloads.

7 Repository and Source Code

All source code developed for this project is publicly available in a dedicated GitHub repository. The repository contains the full implementation of the Java (Hazelcast-based) and Python (MapReduce-style) matrix multiplication solutions, as well as the distributed frequent itemset mining algorithm. It also includes all benchmarking scripts and auxiliary modules used throughout the experiments.

The complete project can be accessed at:

`https://github.com/judporper/Language-Benchmark-of-matrix-multiplication/tree/main/TASK4`

This repository ensures full transparency and reproducibility of the results presented in this report, and provides a reference implementation for future extensions or comparative studies.

References

[1] ML Tech Drawer. *Big Data Course Materials*. Available at: `https://mltechdrawer.github.io/BIGDATA/` (Accessed December 2025).