Distributed Matrix Multiplication with Apache Spark: Design, Implementation, and Evaluation in Java and Python

Diego Muñoz Torres
Universidad de Las Palmas de Gran Canaria

Abstract

This report presents the design, implementation, and evaluation of distributed matrix multiplication using Apache Spark, implemented in both Java and Python. The objective is to study how Spark behaves when executing a compute-intensive, data-heavy kernel and to quantify the overhead introduced by distribution mechanisms such as broadcast and shuffle. Two distributed strategies are implemented. The first is a row-partitioned approach that broadcasts matrix B to all workers and assigns disjoint row ranges of A to partitions so that each partition can compute its output rows independently. The second is a block-partitioned approach that splits matrices into fixed-size blocks and relies on a shuffle-based join and reduction to assemble the final result.

To contextualize the distributed modes, each language also includes a single-machine baseline (`basic`) and a shared-memory parallel mode (`parallel`). All distributed runs are executed in Spark local mode (`local[*]`), which uses one machine but preserves Spark's execution model (jobs, stages, tasks, serialization, and shuffle) and therefore provides meaningful measurements of framework overhead and data movement. Experiments are performed for sizes 512×512 and 1024×1024 using `float32`. The evaluation reports wall-clock time, speedups relative to the baseline, and Spark-derived metrics including shuffle bytes, executor runtime, garbage collection time, spills, number of tasks and stages, and broadcast size.

Results confirm the expected trade-off: the row-based strategy eliminates shuffle at the cost of broadcasting B (1 MiB for 512^2 and 4 MiB for 1024^2), whereas the block-based strategy generates substantial shuffle (about 6.3 MiB for 512^2 and about 42 MiB for 1024^2 with block size 128) due to the block join and reduction pattern. In Java, Spark execution becomes competitive with the shared-memory parallel baseline at 1024^2 in local mode. In Python, the local NumPy baseline is extremely fast for these sizes, so Spark overhead dominates. The report explains these outcomes and discusses how the conclusions would evolve on a real multi-node cluster where distribution is motivated by memory capacity and network scaling.

Keywords: distributed computing, Apache Spark, matrix multiplication, broadcast, shuffle, scalability, block algorithm

1 Introduction

Matrix multiplication is a core primitive in scientific computing, machine learning, graphics, and large-scale data processing. The dense product $C = AB$ for $A \in R^{m \times n}$ and $B \in R^{n \times p}$ requires $\Theta(mnp)$ multiply-add operations and produces an output of size $m \times p$. For square matrices ($m = n = p = N$), computation scales as $\Theta(N^3)$ while memory scales as $\Theta(N^2)$. This asymmetry is a key reason why matrix multiplication becomes difficult to scale: even when memory is still available, runtime can increase sharply with moderate growth in N.

Single-node environments can be highly efficient when they leverage optimized kernels (cache blocking, vectorization, tuned BLAS libraries). However, the distributed setting becomes necessary once matrices exceed the memory capacity of one machine or when execution time must be reduced by scaling across nodes. Distributed systems introduce overhead from scheduling, serialization, and data movement. The relevant question is not whether distributed execution is "free" (it is not), but under what conditions its overhead is justified and which design choices minimize unavoidable costs.

This project addresses those design choices explicitly by implementing and comparing two Spark strategies. The first strategy intentionally minimizes shuffle by broadcasting B and partitioning work by rows of A. The second strategy follows a classic block decomposition that avoids a full broadcast but requires shuffle to bring compatible blocks together and to reduce partial results. These strategies are intentionally different: one optimizes for minimal data exchange, while the other makes data exchange explicit and exposes how shuffle scales with block count.

The implementations emit a scalar checksum per run. The checksum acts as a low-cost correctness signal and avoids storing the entire output matrix C, which would otherwise dominate memory and I/O at large sizes. The checksum is not expected to match across modes because inputs are generated using mode-specific deterministic seeds and partition-dependent generation patterns; it is used to detect internal inconsistencies within a mode rather than to enforce cross-mode identity.

2 Problem Statement

Given two dense matrices $A \in R^{m \times n}$ and $B \in R^{n \times p}$, compute $C = AB$ where

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

The objective of the project is to execute this computation under Spark and to study the performance implications of two distinct distributed decompositions.

The evaluation focuses on three aspects. First, scalability is studied by increasing the problem size from 512 to 1024 in the square case; this represents an 8× increase in arithmetic work. Second, overhead is characterized by separating broadcast costs from shuffle costs and by reporting the total shuffle read/write volume and its growth. Third, resource utilization is summarized by Spark-reported execution structure (stages and tasks) and by aggregated task metrics such as executor runtime, JVM garbage collection time, and spill bytes.

All distributed runs are executed in `local[*]` mode. This choice does not remove Spark overhead; it only removes the need for multiple physical machines. The measurements therefore still capture the job/stage/task pipeline, shuffle mechanics, and serialization, while remaining safe for development on a laptop. In the discussion, these values are interpreted as *transfer volume and overhead indicators*. On a real cluster, the same transfer volumes would be real network traffic and would typically carry a larger penalty.

3 Methodology

3.1 Configuration and Reporting

Both the Java and Python programs are driven by a configuration that specifies the modes to execute (`basic`, `parallel`, `spark-rows`, `spark-blocks`), the matrix cases (here 512^3 and 1024^3), and the execution parameters that should remain consistent across languages. The core parameters are the element type (`float32`), the shared-memory parallel configuration (4 threads with a row-chunk size of 128), and Spark parameters (`local[*]` master, 8 partitions, block size 128 for the block mode, and explicit local directories for Spark temporary data and event logs).

Each run produces a JSON record with the elapsed time, checksum, a success flag, and an `extra` object holding mode-specific metadata. For Spark modes, the `extra` object stores the most relevant overhead signals: shuffle read/write bytes, number of tasks and stages, executor runtime, GC time, and spill bytes. This standardized output allows direct comparison across modes and across languages without relying on external profilers.

3.2 Java Code Structure

The Java project is organized as a Maven application around a `Runner` interface. A single entry point loads the configuration, iterates over all cases and modes, executes the corresponding runner, and finally writes a JSON file containing both configuration and results. The intent of this structure is to keep the "experiment driver" independent of the multiplication kernels, so that adding a new mode does not affect reporting or configuration parsing.

To make the architecture easy to inspect, the main components can be summarized as follows:

- `Runner` (interface): common contract for executing one mode for a given (m, n, p) case and returning a `Result` object.
- `BasicRunner`: single-thread dense multiplication on `float[]` arrays, acting as the computational baseline.
- `ParallelRunner`: shared-memory parallel multiplication using a fixed thread pool, with rows assigned in chunks of `parallelChunkRows=128`.
- `SparkRowsRunner`: Spark RDD execution where each partition computes a subset of output rows after broadcasting matrix B.
- `SparkBlocksRunner`: Spark block algorithm with `blockSize` partitioning, a join-like grouping step (`cogroup`) and a `reduceByKey` accumulation of partial blocks.
- `Utils` (support code): deterministic random generation helpers, block operations (multiply/add), and small utilities used by multiple runners to keep kernels consistent.

The `BasicRunner` implements a single-thread dense multiplication on `float[]` arrays. This baseline is intentionally simple and deterministic so that its runtime reflects raw computation plus local memory access without distributed overhead. The `ParallelRunner` builds on the same numerical kernel but partitions the output by row ranges, using a fixed thread pool and a chunk size of 128 rows. This design is chosen to minimize synchronization: each worker writes to a disjoint region of the output matrix, so the numeric kernel itself requires no locks, and the only coordination cost is task management.

For Spark, Java provides two distinct runners with different overhead goals. `SparkRowsRunner` creates an RDD of row ranges and broadcasts matrix B so that each partition can compute its assigned subset of rows locally. The algorithm is arranged so Spark does not need to shuffle intermediate numerical data: partitions compute independently and only a small checksum-like value is reduced at the end. In contrast, `SparkBlocksRunner` decomposes both matrices into $b \times b$ blocks. Blocks are keyed by the inner dimension index and combined using `cogroup`; partial C blocks are computed and then summed via `reduceByKey`. This pattern intentionally creates a shuffle stage and is representative of the canonical distributed block multiplication when a full broadcast is not viable.

Spark metrics in Java are collected through a custom `SparkListener` that reads `TaskMetrics` on task completion and aggregates values across tasks. The resulting totals represent cluster-wide sums rather than wall-clock time; for example, executor runtime accumulates CPU time across concurrent tasks and can therefore exceed elapsed time. This is expected and useful: it provides a coarse measure of total scheduled work and overhead.

3.3 Python Code Structure

The Python project follows the same conceptual separation between configuration, runners, and reporting. The `utils.py` module defines configuration and results dataclasses, provides directory management and safety checks, and implements the JSON reporter that serializes outputs in a consistent format. This module centralizes "infrastructure" responsibilities so that runner code remains focused on the computation.

The Python architecture is intentionally parallel to the Java one, while also reflecting idiomatic Python conventions:

- `utils.py`: dataclasses `Case` and `Result`, JSON read/write helpers, safety checks (disk space, output directories), and Spark session/helper functions.
- `runners/basic.py`: baseline single-machine multiplication using NumPy (`C = A @ B`) with `float32`.
- `runners/parallel.py`: shared-memory parallel orchestration using `ThreadPoolExecutor` over row chunks, with each chunk computed via NumPy kernels.
- `runners/spark_rows.py`: Spark row-based method with broadcast-B and one partition per row-range group.
- `runners/spark_blocks.py`: Spark block-based method with grouping and reduction of partial blocks (shuffle-heavy by design).

In Python, the `basic` mode uses NumPy for matrix multiplication with float32. NumPy dispatches to optimized native kernels, so the Python baseline represents a strong single-node reference rather than a naive interpreted triple loop. The `parallel` mode partitions the output by row chunks and uses a `ThreadPoolExecutor` to compute those chunks; each chunk still performs its numerical work in NumPy, which means the achievable speedup depends on whether the underlying BLAS is already parallel and on the cost of Python-level orchestration. This is precisely why, in practice.

The Spark modes mirror the Java designs. The row-based Spark runner broadcasts B and partitions work by rows of A, aiming for negligible shuffle. The block-based Spark runner partitions both matrices into blocks and uses a group/join pattern to bring compatible blocks together, then reduces partial blocks into final output blocks. As in Java, the block mode is included because it models the general distributed strategy when broadcasting B is not feasible, even though it is not the best choice for moderate sizes in single-machine local mode.

Spark metrics in Python are collected by querying the Spark UI REST endpoints. This approach is pragmatic in local experimentation: it avoids implementing a custom listener in Python while still providing the key quantities needed for analysis (stages, tasks, shuffle bytes, executor runtime, GC time, spill bytes, and result-size estimates). The reported values are directly comparable to the Java metrics at the level of interpretation required here, even though the collection mechanisms differ.

4 Distributed Designs

4.1 Row-Based Strategy (Broadcast-B)

The row-based strategy assigns each Spark partition a disjoint set of output rows. Matrix B is broadcast once so that every executor can access it without requesting it through shuffle. Each partition then generates the required rows of A deterministically, multiplies by the broadcast B, and accumulates a local checksum that represents the contribution of the computed output rows. A final reduction combines partition checksums into a global checksum.

The overhead profile of this design is predictable. Broadcasting B requires sending $n \cdot p$ elements to each executor; with `float32`, the payload is $4np$ bytes. For the evaluated square cases, this corresponds to 1 MiB at 512^2 and 4 MiB at 1024^2. Because partitions do not exchange intermediate numerical results, shuffle is expected to be near zero, and the Spark job structure should be simple (typically one stage plus a small reduction).

4.2 Block-Based Strategy (Shuffle Join + Reduce)

The block-based strategy uses a fixed block size b and decomposes matrices into $b \times b$ blocks. For $N \times N$ matrices, the number of blocks per dimension is $N_b = N/b$. With $b = 128$, $N_b = 4$ for $N = 512$ and $N_b = 8$ for $N = 1024$. Each block contains b^2 elements; with `float32`, $128^2 \cdot 4 \approx 64$ KiB per block payload.

Computation follows the standard decomposition $C_{ij} = \sum_{k=1}^{N_b} A_{ik} B_{kj}$ at block level. The algorithm groups blocks by the inner index k so that blocks that participate in the same sum are co-located. It then computes all partial block products for each k and emits them keyed by (i, j). Finally, a reduction aggregates partial blocks over k to produce each final output block. This design necessarily generates shuffle because partial results and/or grouped inputs must be routed to the reducers responsible for each output key.

The expected growth of intermediate data is the main reason this strategy is interesting to measure. Even though A and B each contain N_b^2 blocks, the number of partial block products is N_b^3. With $N = 512$ ($N_b = 4$), that is 64 partial blocks; with $N = 1024$ ($N_b = 8$), it becomes 512 partial blocks. This rapid growth explains why shuffle increases sharply as the block grid becomes finer, and it aligns with the measured increase from roughly 6.3 MiB to roughly 42 MiB in shuffle read/write volume between 512^2 and 1024^2.

5 Experimental Setup

All experiments use the same conceptual settings in both languages: matrices of size 512×512 and 1024×1024, element type `float32`, four threads for the shared-memory parallel mode with a row-chunk size of 128, and Spark local execution with eight partitions. The block mode uses block size 128. Spark local directories are configured explicitly to keep temporary data in a controlled location, and a free-disk threshold is enforced to prevent unsafe runs when shuffle grows.

The assignment motivation involves matrices that are too large for one machine. On a laptop, executing extremely large dense matrices is limited by RAM and by the fact that shuffle can rapidly spill to disk. The selected sizes are thus a safe compromise: they are large enough to produce meaningful Spark overhead and shuffle signals, while still fitting in-memory (as confirmed by zero spill bytes in all runs). The analysis then uses the measured broadcast and shuffle volumes to reason about what would happen at larger scales and on multi-node clusters.

6 Results

6.1 Raw Execution Times

Table 1 summarizes Java wall-clock times for each mode.

Table 1: Java execution time (seconds) and speedup vs. `basic`.

Size	Basic	Parallel	Spark-Rows	Spark-Blocks	Speedup (Parallel)
512^2	0.3417	0.1132	0.4087	0.5172	3.02
1024^2	1.9098	0.5322	0.5514	0.5477	3.59

For 512^2, both Spark modes are slower than the basic baseline, which indicates that Spark scheduling and serialization overhead exceed the benefit of task-level parallelism at this scale. At 1024^2, Spark-Rows and Spark-Blocks become much more competitive, approaching the shared-memory parallel time. This reflects two effects: the computation per task is larger (overhead is amortized), and the local Spark scheduler can exploit multiple cores via concurrent tasks.

Table 2 summarizes Python wall-clock times. Here, the baseline represents optimized local computation via NumPy.

Table 2: Python execution time (seconds) and speedup vs. `basic`.

Size	Basic	Parallel	Spark-Rows	Spark-Blocks	Speedup (Parallel)
512^2	0.00957	0.00794	2.0600	3.2991	1.21
1024^2	0.03960	0.03204	1.5432	3.2257	1.24

The Python `parallel` mode provides a modest improvement, which is consistent with a scenario where the underlying numeric kernel is already highly optimized and where additional Python-level threading mainly affects orchestration. In contrast, Spark is orders of magnitude slower than the NumPy baseline at both sizes, meaning the tested problems are still far from the regime where Spark is needed for capacity. Interestingly, Spark-Rows becomes faster at 1024^2 than at 512^2, which supports the interpretation that fixed overhead dominates at smaller sizes and becomes less visible once each task performs more work.

6.2 Spark Overhead and Structure

Tables 3 and 4 report the Spark metrics that best capture overhead: tasks, shuffle bytes, executor runtime, GC time, and spills.

Table 3: Java Spark metrics (selected). Bytes are in absolute units.

Size	Mode	Tasks	Shuffle Read	Shuffle Write	Exec Run (ms)	GC (ms)	Spill (disk)
512^2	Rows	8	0	0	1244	0	0
512^2	Blocks	32	6308688	6308688	2343	40	0
1024^2	Rows	8	0	0	3908	24	0
1024^2	Blocks	32	42016104	42016104	3721	20	0

Table 4: Python Spark metrics (selected). Bytes are in absolute units.

Size	Mode	Stages	Tasks	Shuffle Read	Shuffle Write	Exec Run (ms)	GC (ms)	Spill (disk)
512^2	Rows	1	8	0	0	11794	96	0
512^2	Blocks	3	32	6310336	6310336	31977	210	0
1024^2	Rows	1	8	0	0	11663	0	0
1024^2	Blocks	3	32	42037952	42037952	32026	516	0

The row-based design produces exactly zero shuffle read/write in both languages, which validates the design goal: broadcasting B enables independent row computation. The reported broadcast sizes (1 MiB at 512^2 and 4 MiB at 1024^2) match the theoretical payload of B stored in `float32`. The block-based design, by construction, introduces shuffle and multi-stage execution. The measured shuffle bytes scale sharply with matrix size, and the increase aligns with the growth in the number of partial blocks (N_b^3) when moving from a 4×4 block grid to an 8×8 grid.

A particularly important result is that spills are zero in all runs. This means that the observed overhead is not dominated by disk I/O; instead it reflects Spark's scheduling, serialization, shuffle mechanics, and in the block case the additional coordination required by grouping and reduction. This is desirable for controlled experimentation because it isolates the intrinsic costs of the distributed design.

7 Discussion

7.1 Why Scaling Deviates from the $8\times$ Rule

Doubling N in dense square multiplication increases arithmetic work by $8\times$, but the measured wall-clock scaling deviates from this ideal for reasons that are specific to both the execution model and the chosen baselines.

In Python, NumPy executes matrix multiplication through optimized native kernels that use cache-aware blocking and may exploit parallelism inside the BLAS backend. This explains why the baseline grows by roughly $4\times$ rather than $8\times$ when moving from 512^2 to 1024^2. The computation does increase, but hardware utilization improves and fixed overhead is amortized.

In Spark, the presence of fixed costs (context initialization, job planning, task scheduling, Python-to-JVM bridging in PySpark) means that small problems can be dominated by overhead. As the problem size grows, each task performs more work, so the overhead fraction decreases. This effect is visible in the Python Spark-Rows time, which becomes lower at 1024^2 than at 512^2 despite the larger matrix.

7.2 Broadcast vs Shuffle as a Design Choice

The results show that the two distributed strategies have fundamentally different cost profiles. The row-based strategy transforms the problem into independent work units after broadcasting B. This is attractive when B can fit in memory on each executor and when B can be reused across many tasks, because it avoids shuffle entirely and therefore avoids the most expensive category of distributed data movement in Spark.

The block-based strategy is the more general approach when broadcasting is not possible or when both matrices are too large to replicate. Its generality comes with an unavoidable cost: shuffle volume grows quickly because intermediate partial blocks must be routed and aggregated. The measured jump in shuffle volume from about 6.3 MiB to about 42 MiB between 512^2 and 1024^2 illustrates how fast this cost can rise even with a fixed block size.

7.3 Interpreting Java vs Python Outcomes

The most important cross-language insight is not a direct comparison of absolute times, because the baselines are not equivalent: Java uses explicit arrays and loops, while Python uses NumPy (native optimized). Instead, the meaningful comparison is that Spark overhead patterns are consistent across languages. In both, row-based multiplication eliminates shuffle and exposes broadcast as the primary data movement. In both, block-based multiplication introduces large shuffle traffic and multiple stages, and shuffle grows sharply with block grid size.

In Java local mode, Spark can approach the shared-memory parallel baseline at 1024^2 because Spark tasks execute on multiple cores and the numerical kernel is implemented inside the JVM without crossing language boundaries. In Python, Spark remains far slower than NumPy because the single-node baseline is already near the machine's practical capability for these sizes, and the additional framework overhead is not compensated unless the workload is scaled beyond a single machine's capacity.

8 Conclusions

This work implemented distributed dense matrix multiplication in Spark using two complementary strategies and evaluated them in Java and Python alongside local baselines.

The row-based design behaves as intended: shuffle is eliminated and the dominant data transfer becomes the broadcast of B, whose size matches theoretical expectations for `float32`. The block-based design exposes the classical shuffle-heavy behavior of distributed block multiplication; its shuffle volume grows rapidly with matrix size and block grid resolution, and it requires multiple Spark stages.

On a single machine in `local[*]` mode, Spark is not competitive with highly optimized single-node numerical kernels for moderate sizes, particularly in Python where NumPy provides a strong baseline. Nevertheless, Spark metrics (task structure, shuffle volume, executor runtime, and spills) provide a clear and reproducible way to quantify distributed overhead and to reason about how these approaches would behave on a real cluster where scaling is motivated by memory capacity and by the need to distribute computation and storage.

9 Future Work

A natural extension is to run the same code on a multi-node cluster. This would turn shuffle bytes into real network traffic and would allow the study of how performance evolves with additional executors and distributed memory capacity. Another extension is to increase matrix sizes until spilling appears, because the transition from in-memory shuffle to spill-dominated execution is a key practical limit in Spark workloads. Finally, systematic tuning of block size and partition count would make it possible to quantify the trade-off between parallelism and shuffle overhead, and to identify configurations that maximize throughput for a given cluster and workload.

Repository and Reproducibility Notes

All source code and this paper are publicly available at:
`https://github.com/data-dmt/BigData/tree/main/Individual_Assignment/DistributedMatrixMultiplication`

Repository contents:

- `paper/` folder containing the LaTeX source of this report (main `.tex` file, bibliography if applicable) and the compiled PDF, plus any figures used in the document.
- `code/java/` folder containing the Java **Maven** project (`pom.xml`) for the DMM benchmark. Source code lives under `src/main/java/ulpgc/shared/` and includes `App.java` (entry point and orchestration), `Utils.java` (matrix generation/helpers), the `Runner` interface, and the four mode implementations: `BasicRunner`, `ParallelRunner`, `SparkRowsRunner`, and `SparkBlocksRunner`. The project writes run outputs to `results/` as JSON files (`dmm_results_java.json`) and uses `spark_tmp/` as the configured Spark local directory.
- `code/python/` folder containing the Python implementation of the same benchmark. The entry point is `app.py`; shared utilities and dataclasses are in `utils.py`; and the four modes are implemented in `runners/` (`basic.py`, `parallel.py`, `spark_rows.py`, `spark_blocks.py`). The program writes outputs to `results/` as JSON files (`dmm_results_py.json`) and uses `spark_tmp/` and `spark_eventlog/` for Spark local data and event logs.

References

[1] Apache Spark, "RDD Programming Guide," Spark Documentation, v3.5.x. Available: `https://downloads.apache.org/spark/docs/3.5.1/rdd-programming-guide.html`.

[2] Apache Spark, "Monitoring and Instrumentation," Spark Documentation, v3.5.x. Available: `https://downloads.apache.org/spark/docs/3.5.1/monitoring.html`.

[3] NumPy Developers, "NumPy Documentation: `matmul` and array operations." Available: `https://numpy.org/doc/`.

[4] OpenAI, "ChatGPT (GPT-5)," 2025. Available: `https://chat.openai.com`.