TASK 4. Distributed Execution of matrix multiplication
Big Data

Alberto Guedes

December 2025

Contents

Abstract

This report presents the design, implementation, and evaluation of a distributed matrix multiplication using a block-based **MapReduce** approach implemented in both **Python** and **Java**. Matrices are partitioned into fixed-size blocks; mappers emit keyed block data, shuffling groups compatible blocks, and reducers compute block products to assemble the final result. Benchmarking has been done across matrix sizes $N \in \{256, 512, 1024\}$ and block sizes $b \in \{16, 32, 64\}$, reporting phase-level timings: generation, blocking, map, shuffle, reduce, and total. Results show Java significantly outperforms Python, with speedups growing with matrix size and for larger block sizes due to reduced communication overhead.

Repository: GitHub Repository

1 Methodology

1.1 Overview

Computation of $C = A \times B$ via **block matrix multiplication**. Partition A and B into $b \times b$ blocks so that

$$C_{ij} = \sum_k A_{ik} B_{kj}, \quad \text{where } A_{ik}, B_{kj}, C_{ij} \in \mathbb{R}^{b \times b}.$$

The **Map** stage emits, for every block of A and B, a key identifying destination block (i, j) and a value containing block data: (k, A_{ik}) or (k, B_{kj}). The **Shuffle** stage groups values by key (i, j). The **Reduce** stage iterates k to multiply compatible pairs and accumulates into C_{ij}.

1.2 Design Decisions

- **Block-based**: Reduces communication and leverages cache locality versus element-wise schemes.
- **Fixed block sizes**: Evaluate $b \in \{16, 32, 64\}$ to study computation vs. communication trade-offs.
- **Phase timings**: Recorded `generation`, `blocking`, `map`, `shuffle` (`map_shuffle` in CSV), `reduce`, and `total` for fine-grained analysis.
- **Concurrency**: Java uses `ExecutorService` with a thread pool; Python uses `multiprocessing.Pool` to avoid the GIL and parallelize mappers.

1.3 Java Implementation

The Java implementation resides under `java/src/` with core classes:

- `MatrixGenerator`: Creates dense random matrices.
- `BlockUtils`: Splits/join matrices into/from blocks.
- `Mapper`: Emits `MapOutput` keyed by `BlockIndex` with `BlockValue` for A and B blocks.
- `Reducer`: Multiplies matched A_{ik} and B_{kj} blocks, accumulates C_{ij}.
- `Driver`: Orchestrates benchmarking loops over N and b, runs mappers in parallel, shuffles, reduces, and writes CSV.

- `CSVUtils`: Appends rows to `benchmarks/results.csv` with a header-once policy.

A fixed-size pool has been used: `Executors.newFixedThreadPool(workers)`. Mapper outputs are collected to a synchronized list and timed (`map`); grouping by key is performed in-process and timed (`shuffle`). Reduce computes each `BlockIndex` result block. The driver prints per-run metrics and appends a CSV row using US decimal formatting.

1.4 Python Implementation

The Python implementation resides under `python/src/`:

- `matrix_generator.py`, `block_utils.py`, `mapper.py`, `reducer.py`: Functional equivalents of Java components.
- `driver.py`: Benchmarks over N and b, uses `multiprocessing.Pool.starmap` to run mappers, shuffles with a helper, reduces, and writes CSV with six-decimal precision.

Python records the same phases. Differences arise from process-based parallelism, data serialization overhead, and interpreter cost.

1.5 Benchmark Configuration

Evaluate on matrix sizes $N \in \{256, 512, 1024\}$ and block sizes $b \in \{16, 32, 64\}$ have been done, using all available CPU cores as workers. Each combination emits one CSV row per language with phase timings and total.

2 Results

2.1 Summary

Table 1 reports total time (seconds) for $b = 64$ across languages and sizes, extracted from `benchmarks/results.csv`.

Table 1: Total execution time vs. matrix size for block size $b = 64$

oprule Language	Matrix Size	Block Size	Total (s)
Java	256	64	0.010894
Python	256	64	1.633792
Java	512	64	0.080268
Python	512	64	11.614642
Java	1024	64	0.570978
Python	1024	64	90.137581

2.2 Observations

- **Language gap**: Java consistently outperforms Python. For $(N, b) = (1024, 64)$, Java total $\approx 0.57\,\mathrm{s}$ vs. Python $\approx 90.14\,\mathrm{s}$ ($\sim 158\times$ speedup).
- **Scaling with** N: Total time grows with matrix size; growth is far steeper for Python due to higher overhead in mapping and reduction.
- **Effect of** b: Larger blocks reduce map/shuffle overhead and improve totals (e.g., at $N = 512$, Java total drops from 0.167 at $b = 16$ to 0.080 at $b = 64$).
- **Phase breakdown**: For small b, communication (map/shuffle) is relatively heavier; for larger b, reduce dominates as block multiplication becomes costlier but fewer groups exist.

2.3 Visualization

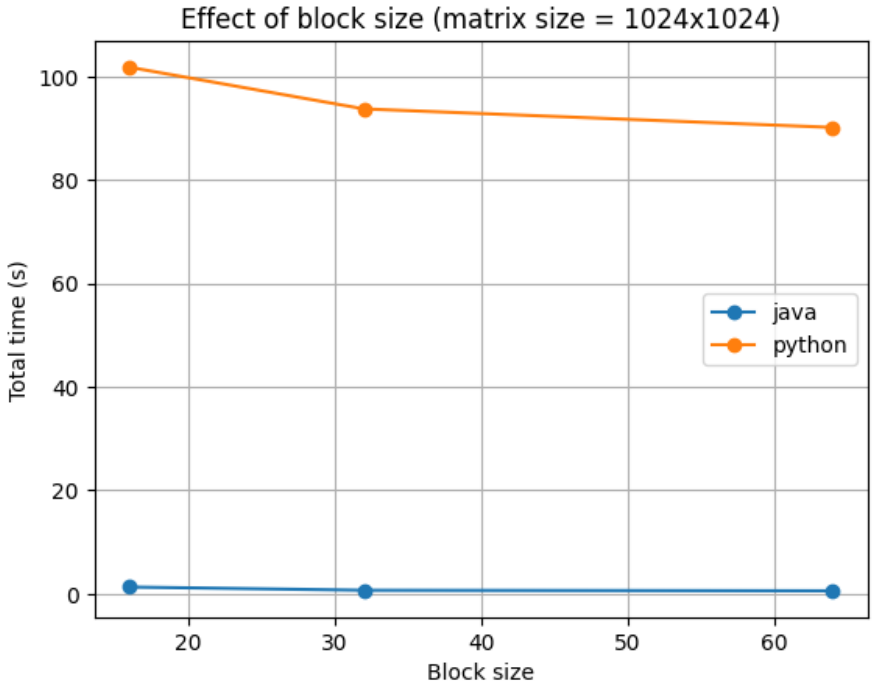

Figure 1: Effect of block size for N = 1024.

Figure 1 (produced by `benchmark_results.ipynb`) visualizes total time based on block size for N = 1024.

Figure 2 (produced by `benchmark_results.ipynb`) visualizes total execution time based on matrix size.

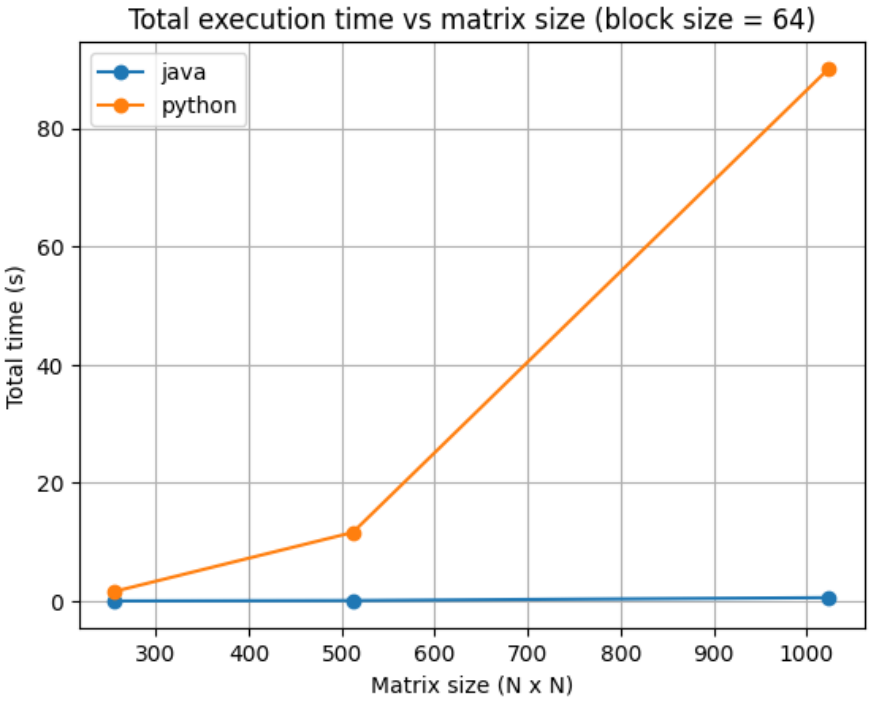

Figure 2: Total execution time based on N for b = 64.

Figure 3 (produced by `benchmark_results.ipynb`) visualizes total execution time for each major process from the MapReduce framework.

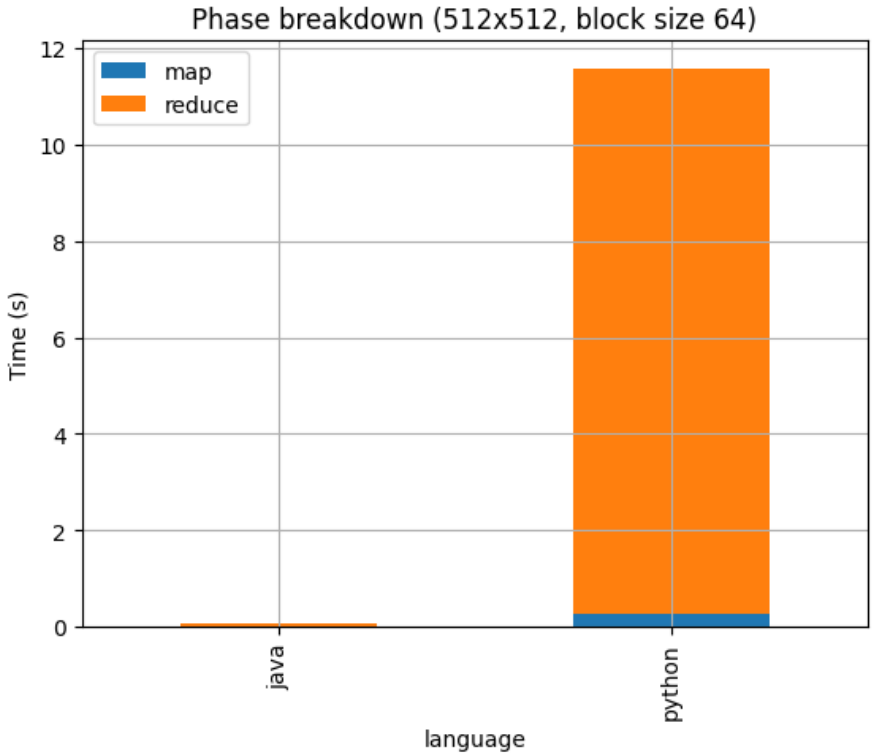

Figure 3: Phase breakdown between mapping and reducing.

3 Discussion

3.1 Why Java is Faster

- **Runtime and memory**: The JVM JIT, optimized memory layouts, and reduced serialization overhead outperform Python's interpreter and process-based IPC.
- **Parallelism model**: Java threads share memory; Python's `multiprocessing` requires pickling/unpickling, adding overhead in map/shuffle.
- **Numeric throughput**: Java's loops over primitive arrays avoid Python-level iteration costs and benefit from CPU caches.

3.2 Block Size Trade-offs

- **Small** b: More blocks increase scheduler, shuffle, and metadata overhead; better load balance but higher communication.

- **Large** b: Fewer, heavier tasks reduce communication and improve locality; reducers do more work per key.

3.3 Limitations and Improvements

- **Single-node MapReduce**: Experiments run on one machine; a distributed cluster would change shuffle costs and network patterns.
- **I/O and formats**: Using direct byte buffers or shared memory segments could reduce Python's serialization overhead.
- **BLAS acceleration**: Replacing block multiplication with optimized BLAS could substantially cut reduce times in both languages.
- **Adaptive** b: Selecting block size based on N and hardware cache sizes can improve performance.

4 Conclusion

Implementation of **block-based MapReduce** matrix multiplication in both **Python** and **Java**, instrumented per-phase timings, and evaluated across sizes $N \in \{256, 512, 1024\}$ and blocks $b \in \{16, 32, 64\}$ have been conducted. **Java** delivers markedly lower runtimes due to reduced overhead in mapping/shuffling and faster numeric computation, while **Python** exhibits substantial serialization and interpreter costs. Larger blocks generally improve performance by lowering communication. Future work could include multi-node deployments, BLAS-backed reducers, and adaptive block sizing.

Reproducibility

The results were obtained using a PC equipped with a 6-core CPU and 16GB DDR4 RAM at 3200mhz.

Repository

All code and artifacts are available at: GitHub Repository.

Analysis Notebook

Open `analysis.ipynb` and run the cells to reproduce plots and pivot tables. The notebook loads `benchmarks/results.csv` generated by both drivers.