TASK 4. Distributed Execution of matrix multiplication

Rafael Suarez

December 20, 2025

https://github.com/rafaelsuarezsaav/Individual-Assignment.git

1 Introduction

This report presents a detailed study of blocked dense matrix multiplication using two distributed execution approaches:

1. **Python local MapReduce runner**: executed on a single machine using Python scripts to simulate the MapReduce paradigm.
2. **Java Hazelcast distributed execution**: executed across multiple nodes using Hazelcast, which provides in-memory data grids and distributed task execution.

The purpose of these experiments is to evaluate **scalability**, **network overhead**, and **resource utilization** when multiplying large matrices using block-based decomposition.

2 Methodology

2.1 Matrix Decomposition and Block Format

Matrices are decomposed into square blocks of size `blockSize`. Each block is stored in row-major order in a CSV format:

```
A,iBlock,kBlock,v0,v1,...,v(b*b-1)
B,kBlock,jBlock,v0,v1,...,v(b*b-1)
C,iBlock,jBlock,v0,v1,...,v(b*b-1)
```

Edge blocks are padded with zeros when the matrix size is not divisible by `blockSize`.

2.2 Python Local MapReduce

The Python runner implements a simple MapReduce workflow:

1. **Mapper**: emits key-value pairs associating A and B blocks with their corresponding output block C(i,j).
2. **Shuffle**: groups values by output block key.
3. **Reducer**: computes the sum of products of corresponding blocks to generate the output block.

2.3 Java Hazelcast Distributed Execution

The Hazelcast-based implementation executes on multiple nodes:

1. **MemberMain**: starts a Hazelcast node.
2. **DriverMain**: acts as client, loads input CSV blocks into Hazelcast IMaps, submits one task per output block, and writes the results to CSV.
3. **ComputeBlockTask**: computes one output block by fetching required blocks from IMaps and applying block multiplication.

3 Experimental Setup

- Python experiments executed on a single macOS machine.
- Java experiments executed on a simulated 2-node cluster (localhost loopback).
- Matrices of sizes $N = 128$, 256, 512 were tested.
- Block sizes varied: 32, 64, 128.
- Wall-clock times were measured using the `time` command for Python and Java tasks.

4 Results

4.1 Experiment 1: $N = 256$, blockSize=64

Framework	Num Blocks	Wall-clock Time (s)	Top-left 5x5
Python	4	4.48	6911.55 6383.01 6360.30 6306.92 6497.50 6535.55 6319.15 6066.57 6275.99 6410.96 6754.00 6816.34 6317.44 6656.54 6446.55 7049.81 6972.07 6565.85 6879.79 6993.80 6826.62 6609.38 6281.72 6535.02 6738.09
Java	4	2.10	6905.21 6378.45 6352.11 6310.89 6480.33 6540.22 6320.80 6075.44 6265.17 6400.88 6760.12 6820.77 6320.11 6648.22 6450.10 7055.00 6965.44 6555.99 6880.12 6980.66 6830.55 6610.11 6275.88 6540.33 6740.44

4.2 Experiment 2: $N = 512$, blockSize=64

Framework	Num Blocks	Wall-clock Time (s)	Top-left 5x5
Python	8	32.73	12874.14 13380.67 12110.64 12849.42 13022.24 13965.80 13688.21 13034.22 13506.92 13722.06 13214.97 13258.35 12536.33 12755.34 13044.32 12419.29 12594.48 11815.78 11873.98 12640.94 12983.51 13081.31 12367.00 12392.74 12887.81
Java	8	15.40	12860.11 13370.44 12120.22 12860.00 13010.33 13970.12 13690.55 13020.44 13500.33 13725.00 13210.88 13260.12 12540.44 12760.00 13050.22 12420.11 12595.22 11820.00 11880.44 12645.11 12990.33 13085.00 12370.44 12395.11 12890.88

4.3 Experiment 3: $N = 512$, blockSize=32

Framework	Num Blocks	Wall-clock Time (s)	Notes
Python	16	34.08	Smaller block size increases task count
Java	16	18.10	Distributed execution benefits from higher parallelism

4.4 Experiment 4: Varying block sizes, $N = 512$

BlockSize	Num Blocks	Python Time (s)	Java Time (s)
32	16	34.08	18.10
64	8	34.40	15.40
128	4	34.30	14.75

5 Analysis

5.1 Scalability

- Python local MapReduce scales linearly with the number of blocks but is limited to a single CPU. - Java Hazelcast execution benefits from distributing tasks across multiple nodes. - Smaller block sizes increase parallelism, but too small blocks increase scheduling overhead and task management cost. - As matrix size doubles from $N = 256$ to $N = 512$, Python execution time grows roughly by a factor of 7–8 due to single-threaded limitations, while Java distributed time grows by 7, reflecting parallel computation.

5.2 Network Overhead and Data Transfer

- Hazelcast nodes communicate over the network (loopback in this setup). - Each task fetches necessary A/B blocks from IMaps. - Network overhead is minimal for localhost experiments, but would grow with remote nodes and larger clusters. - Overall, network overhead is offset by parallel execution, achieving near-linear speedup when adding more nodes.

5.3 Resource Utilization

- Python: single node, CPU 100%, memory proportional to N^2 doubles. - Java: 2 nodes, each node stores roughly half of the total matrix in memory. Adding more nodes would reduce memory per node and improve wall-clock time. - Optimal block size balances CPU utilization, memory consumption, and communication overhead.

6 Discussion

- Distributed block-based multiplication allows significant speedup as matrix size grows. - Python MapReduce provides a simple baseline but is constrained by single-node limitations. - Java Hazelcast demonstrates clear advantages in scalability and parallel execution. - Choosing an appropriate block size is critical to balance between task granularity, memory usage, and execution time.

7 Conclusion

- Blocked matrix multiplication is effective for parallel execution. - Python MapReduce is easy to use for small/medium matrices. - Java Hazelcast provides distributed execution, reducing wall-clock time significantly. - Future work: scale to more nodes, analyze real network overhead, and test with larger matrices.