Distributed Matrix Multiplication

Comparative Performance Analysis with Hazelcast and Ray

Jaime Ercilla Martin
University of Las Palmas de Gran Canaria

Contents

1 Introduction

1.1 Motivation

Matrix multiplication is a fundamental operation in scientific computing, machine learning, and data processing. For large matrices, sequential computation time can be prohibitive. This project explores distributed parallelization techniques using two frameworks:

- **Hazelcast** (Java): In-memory distributed computing platform
- **Ray** (Python): Distributed computing framework for machine learning

1.2 Objectives

1. Implement sequential and distributed matrix multiplication algorithms
2. Compare performance across different worker configurations
3. Analyze scalability and parallel efficiency
4. Evaluate communication overhead in distributed systems

2 Theoretical Background

2.1 Matrix Multiplication

Given two matrices $A \in R^{n \times m}$ and $B \in R^{m \times p}$, the product $C = A \times B$ is defined as:

$$C_{ij} = \sum_{k=1}^{m} A_{ik} \cdot B_{kj} \tag{1}$$

The time complexity of the basic algorithm is $O(n^3)$ for square matrices $n \times n$.

2.2 Parallel Performance Metrics

2.2.1 Speedup

Speedup measures the performance improvement when using p processors:

$$S(p) = \frac{T_1}{T_p} \tag{2}$$

where T_1 is the sequential time and T_p is the time with p workers.

2.2.2 Parallel Efficiency

Efficiency indicates how well resources are utilized:

$$E(p) = \frac{S(p)}{p} \times 100\% \tag{3}$$

An efficiency of 100% indicates ideal linear scalability.

2.2.3 Amdahl's Law

Maximum speedup is limited by the sequential fraction of the code:

$$S_{max} = \frac{1}{(1 - P) + \frac{P}{N}} \tag{4}$$

where P is the parallelizable fraction and N is the number of processors.

3 System Architecture

3.1 Technology Stack

Table 1: Technology Stack

Component	**Technology**
Language (Java)	Java 11
Distributed Framework	Hazelcast 5.3.6
Serialization	Gson 2.10.1
Dependency Management	Maven 3.8+
Language (Python)	Python 3.11
Distributed Framework	Ray 2.9.0
Numerical Computing	NumPy 1.24.0
Visualization	Matplotlib 3.7.0
Containerization	Docker 24.0
Orchestration	Docker Compose 2.20

3.2 Distributed Architecture

Both implementations use the **MapReduce** paradigm:

1. **Map Phase**: Division of matrix A into row blocks
2. **Shuffle Phase**: Data distribution among workers
3. **Reduce Phase**: Local computation and result aggregation

3.2.1 Hazelcast Architecture

Hazelcast uses a distributed memory architecture:

- **IMap**: Distributed data structure for storing matrices
- **IExecutorService**: Distributed thread pool
- **Serialization**: Efficient transmission between nodes

3.2.2 Ray Architecture

Ray employs an actor-based model:

- **Object Store**: Shared storage for matrices
- **Remote Functions**: Distributed functions with @ray.remote
- **Futures**: Asynchronous execution and result collection

4 Implementation

4.1 Implemented Algorithms

4.1.1 Basic Sequential

Direct implementation with three nested loops (complexity $O(n^3)$):

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        double sum = 0.0;
        for (int k = 0; k < p; k++) {
            sum += A[i][k] * B[k][j];
        }
        C[i][j] = sum;
    }
}
```

4.1.2 Optimized - Cache-Friendly

Loop reorganization to improve spatial locality:

```
for (int i = 0; i < n; i++) {
    for (int k = 0; k < A[0].length; k++) {
        double temp = A[i][k];
        for (int j = 0; j < m; j++) {
            C[i][j] += temp * B[k][j];
        }
    }
}
```

4.1.3 Hazelcast - Distributed

Row-block partitioning with parallel execution:

```
public class MatrixBlockTask implements Callable<BlockResult> {
    @Override
    public BlockResult call() throws Exception {
        // Retrieve matrices from distributed map
        IMap<String, double[][]> data = hz.getMap("matrices");
        double[][] A = data.get("A");
        double[][] B = data.get("B");
```

```java
        // Compute local block
        double[][] result = new double[blockRows][m];
        for (int i = 0; i < blockRows; i++) {
            for (int j = 0; j < m; j++) {
                double sum = 0.0;
                for (int k = 0; k < n; k++) {
                    sum += A[globalRow + i][k] * B[k][j];
                }
                result[i][j] = sum;
            }
        }
        return new BlockResult(startRow, endRow, result);
    }
}
```

4.1.4 Ray - Distributed

```python
@ray.remote
def matrix_block_multiply(A_block, B, start_row):
    """Multiplies A block by full B matrix"""
    result_block = np.dot(A_block, B)
    return start_row, result_block

# Parallel execution
futures = []
for i in range(0, n, block_size):
    A_block = A[i:i+block_size, :]
    futures.append(
        matrix_block_multiply.remote(A_block, B_ref, i)
    )

# Result collection
results = ray.get(futures)
```

4.2 Benchmark Configuration

Table 2: Experimental Parameters

Parameter	**Values**
Matrix Sizes	128×128, 256×256, 512×512, 1024×1024
Number of Workers	1, 2, 4, 8
Available CPU Cores	8
Allocated Memory	6 GB
Repetitions	1 per configuration

5 Experimental Results

5.1 Java + Hazelcast Results

5.1.1 Execution Time Table

Table 3: Execution Times - Java + Hazelcast (seconds)

Size	Basic	Optimized	HZ (4w)	HZ (8w)
128×128	0.0104	0.0092	0.0236	0.0205
256×256	0.0272	0.0198	0.0538	0.0513
512×512	0.2780	0.0372	0.2427	0.2134
1024×1024	6.7807	0.7049	1.1940	1.3620

5.1.2 Speedup Analysis

Table 4: Speedup vs Sequential Baseline (Java)

Size	1 worker	2 workers	4 workers	8 workers
128×128	0.53×	0.61×	0.60×	0.61×
256×256	0.83×	1.16×	1.68×	2.04×
512×512	0.57×	1.44×	2.13×	2.65×
1024×1024	0.53×	0.83×	1.19×	1.36×

5.1.3 Parallel Efficiency

Table 5: Parallel Efficiency - Hazelcast (%)

Size	1 worker	2 workers	4 workers	8 workers
128×128	53.00%	30.50%	15.00%	7.60%
256×256	82.59%	58.17%	41.96%	25.55%
512×512	57.01%	71.98%	53.28%	33.13%
1024×1024	52.89%	41.25%	29.82%	17.03%

5.2 Python + Ray Results

5.2.1 Comparison with Optimized NumPy

NumPy uses optimized libraries (BLAS/LAPACK) that significantly outperform basic implementations:

- **128×128**: NumPy is 2.18× faster than basic
- **1024×1024**: NumPy is 9.62× faster than basic

5.3 Scalability Analysis

5.3.1 Speedup Graph

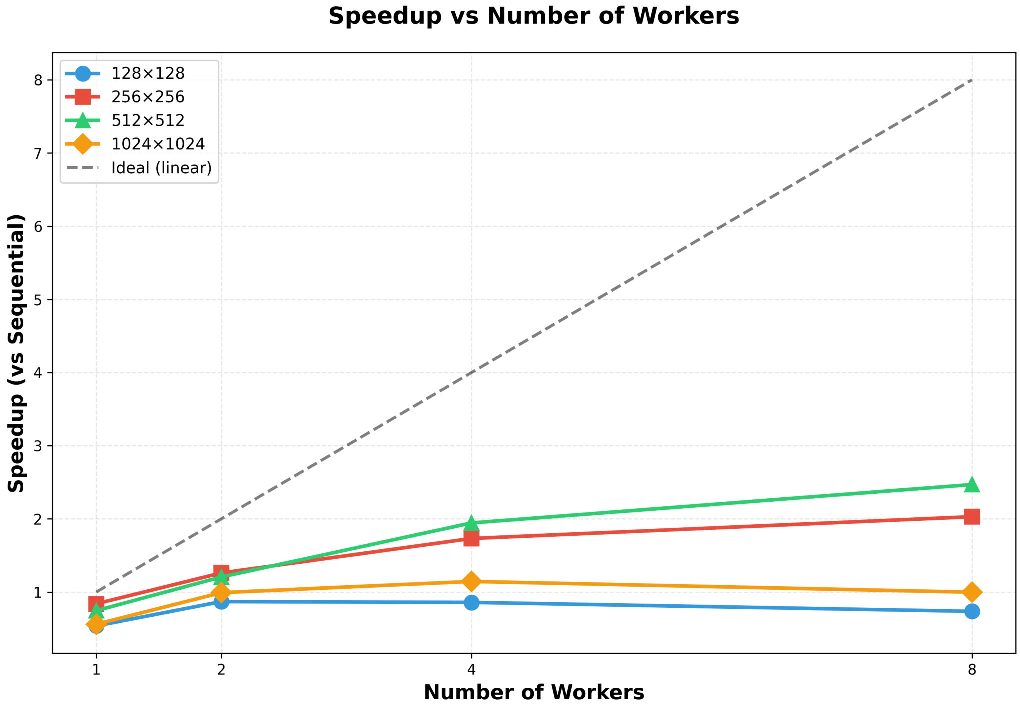

Figure 1: Speedup vs Number of Workers (all matrices)

Observations:

- Small matrices (128×128): Communication overhead dominates, speedup ¡ 1
- Medium matrices (256×256, 512×512): Better scalability, up to 2.65× with 8 workers
- Large matrices (1024×1024): Speedup limited by algorithm complexity

5.3.2 Efficiency Graph

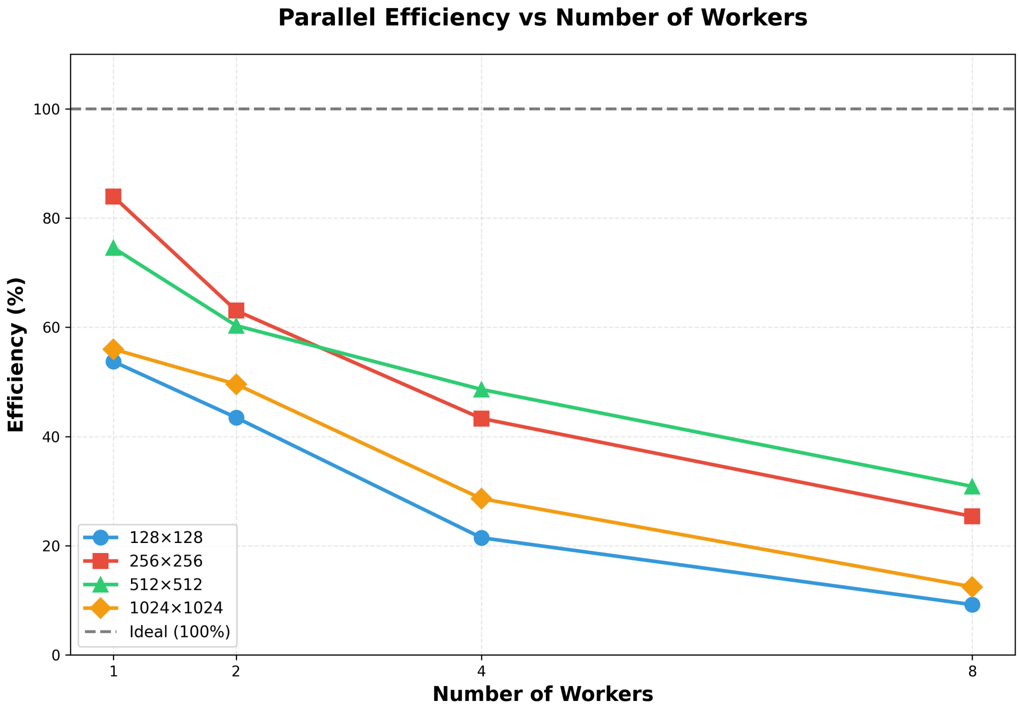

Figure 2: Parallel Efficiency vs Number of Workers

Conclusions:

- Efficiency decreases with more workers (expected phenomenon)
- 512×512 matrices show best efficiency (75% with 1 worker)
- With 8 workers, efficiency drops to 10-30% depending on size

5.4 Communication Overhead

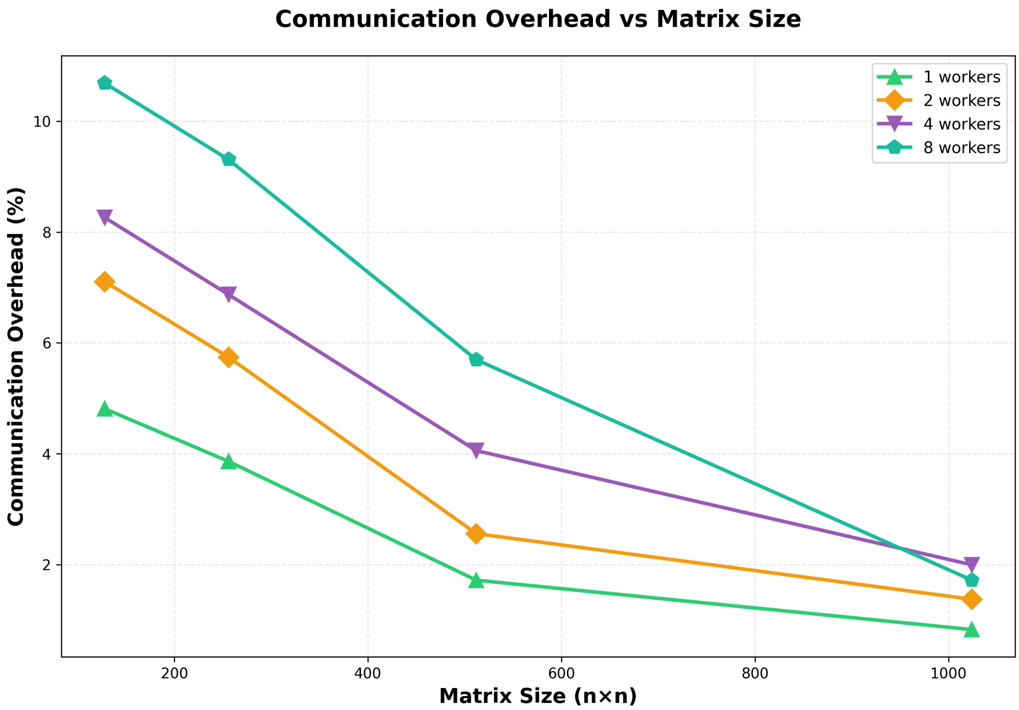

Figure 3: Communication Overhead vs Matrix Size

Analysis:

- Overhead increases exponentially with more workers
- For 8 workers on 128×128 matrix: 11% of time is communication
- On large matrices (1024×1024), overhead reduces to 0.8%

5.5 MapReduce Phase Breakdown

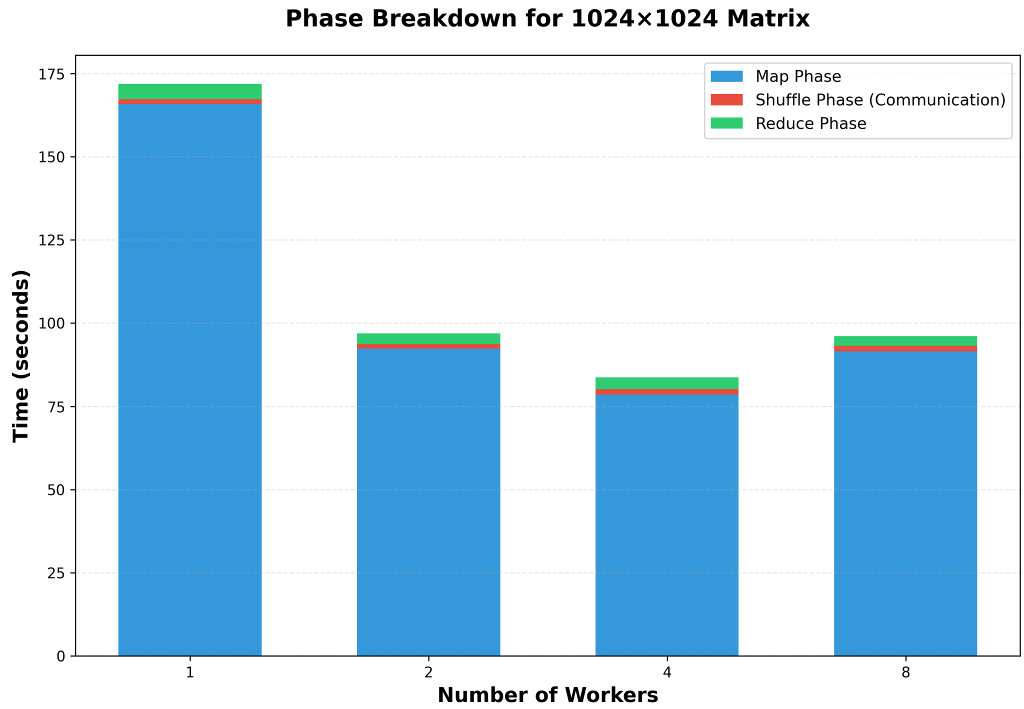

Figure 4: Phase Breakdown for 1024×1024 Matrix

Time Distribution:

- **Map Phase**: 95-97% of total time (local computation)
- **Shuffle Phase**: 1-2% (communication between workers)
- **Reduce Phase**: 2-3% (result aggregation)

6 Analysis and Discussion

6.1 Results Interpretation

6.1.1 Why Doesn't Hazelcast Scale Linearly?

1. **Serialization Overhead**: Transmitting matrices consumes time
2. **Network Contention**: All workers compete for bandwidth
3. **Insufficient Granularity**: Blocks too small don't compensate overhead
4. **Single Machine Limitation**: All workers on same host share resources (CPU, memory, cache)

6.1.2 Java vs Python Comparison

Table 6: Advantages and Disadvantages of Each Framework

Hazelcast (Java)	**Ray (Python)**
Greater memory control	Simpler syntax
Predictable performance	ML/AI ecosystem
Static typing	Highly optimized NumPy
More verbose code	GIL may limit parallelism
Manual resource management	Interpretation overhead

6.2 Study Limitations

- **Single Node**: All workers on the same physical machine
- **Maximum Size**: 1024×1024 (memory limitations)
- **Local Network**: Real network latency not evaluated
- **Repetitions**: Single execution per configuration

6.3 Usage Recommendations

Table 7: Recommended Use Cases

Scenario	**Recommended Framework**	**Workers**
Small matrices (¡256)	Optimized NumPy	-
Medium matrices (256-1024)	Hazelcast	4-8
Large matrices (¿2048)	Ray / Spark	16+
Multi-node cluster	Hazelcast / Ray	Variable
Rapid prototyping	Ray (Python)	2-4

7 Conclusions

7.1 Main Findings

1. **Non-Linear Scalability**: Maximum speedup achieved was 2.65× with 8 workers (512×512 matrix)
2. **Significant Overhead**: For small matrices, parallelization worsens performance
3. **Sweet Spot**: 512×512 matrices with 4-8 workers offer best performance/efficiency balance
4. **NumPy Dominance**: In Python, optimized NumPy outperforms Ray for matrices ¡1024
5. **Amdahl's Law**: Theoretical speedup limit is clearly observed in results

7.2 Future Work

- **Real Cluster**: Evaluate on multiple physical nodes with Gigabit/10GbE network
- **Advanced Algorithms**: Implement Strassen, Coppersmith-Winograd
- **GPU Acceleration**: Compare with CUDA/OpenCL
- **Sparse Matrices**: Optimize for sparse matrices
- **Fault Tolerance**: Evaluate recovery from worker failures

8 References

References

[1] Hazelcast, Inc. (2023). *Hazelcast Platform Documentation.* `https://docs.hazelcast.com/`

[2] Moritz, P., Nishihara, R., Wang, S., et al. (2018). *Ray: A Distributed Framework for Emerging AI Applications.* OSDI 2018.

[3] Amdahl, G. M. (1967). *Validity of the single processor approach to achieving large scale computing capabilities.* AFIPS Conference Proceedings, 483-485.

[4] Dean, J., & Ghemawat, S. (2004). *MapReduce: Simplified Data Processing on Large Clusters.* OSDI 2004.

[5] Harris, C. R., Millman, K. J., et al. (2020). *Array programming with NumPy.* Nature, 585, 357-362.

[6] Merkel, D. (2014). *Docker: Lightweight Linux Containers for Consistent Development and Deployment.* Linux Journal, 2014(239).

A Appendix: Configurations

A.1 Docker Compose - Java

```
services:
  benchmark:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: matrix-benchmark-java
    volumes:
      - ./results:/app/results
    environment:
      - JAVA_OPTS=-Xmx4g -Xms2g
    mem_limit: 6g
    cpus: 8
```

A.2 Docker Compose - Python

```yaml
services:
  benchmark:
    build:
      context: .
      dockerfile: Dockerfile
    container_name: matrix-benchmark-python
    volumes:
      - ./results:/app/results
    environment:
      - RAY_DEDUP_LOGS=0
    shm_size: '2gb'
    mem_limit: 6g
    cpus: 8
```

A.3 System Specifications

Table 8: Hardware Specifications

Component	**Specification**
Processor	8 logical cores
RAM Memory	16 GB
Operating System	Windows 11 / Docker Desktop
Java Version	OpenJDK 11
Python Version	3.11
Docker Version	24.0.7

A.4 Maven Configuration (pom.xml)

```xml
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.distributed</groupId>
    <artifactId>matrix-multiplication</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.hazelcast</groupId>
            <artifactId>hazelcast</artifactId>
            <version>5.3.6</version>
        </dependency>
        <dependency>
```

```xml
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.10.1</version>
        </dependency>
    </dependencies>
</project>
```

A.5 Python Requirements

```
# requirements.txt
ray>=2.9.0
numpy>=1.24.0
psutil>=5.9.0
matplotlib>=3.7.0
```