Matrix Multiplication

Jakub Król

Individual Assignment
BIG DATA
Grado en Ciencia e Ingeniería de Datos
Universidad de Las Palmas de Gran Canaria

Contents

Abstract

This report explores the performance of matrix multiplication using distributed computing techniques, specifically leveraging Hazelcast for parallelizing tasks across multiple machines. The goal was to assess how distributed systems could improve the efficiency of matrix multiplication, especially for larger matrices, in comparison to traditional multi-threading approaches.

The experiments were conducted using matrices of various sizes (from 10x10 to 512x512) with different thread counts, and the results showed that while distributed computing has the potential to provide significant speedup, practical challenges hindered the expected performance. Issues with distributed communication, particularly with Docker containers running Hazelcast nodes, prevented successful scaling beyond a certain point, causing the distributed approach to underperform compared to non-distributed implementations.

While the previous non-distributed approach achieved better results with larger matrices, the distributed matrix multiplication approach faced performance limitations due to hardware constraints and network communication overhead. This report also discusses the impact of matrix size, thread count, and the hardware limitations on execution time and memory usage.

Finally, suggestions for future work include resolving technical issues with the distributed setup, optimizing the matrix multiplication algorithm for better performance, and exploring alternative parallelization techniques that can better utilize available hardware.

1 Introduction

Matrix multiplication is a fundamental operation in many scientific and engineering applications. As matrices grow in size, the computational cost increases exponentially. Traditional matrix multiplication algorithms perform the operation sequentially, which becomes infeasible for large matrices. Parallel computing approaches attempt to address this problem by dividing the work across multiple processing units, speeding up the computation. In a distributed system, the matrix multiplication task can be split across multiple nodes, allowing the system to handle extremely large matrices that might not fit into the memory of a single machine.

In this report, we explore the use of distributed computing for matrix multiplication using Hazelcast, a distributed computing framework that provides tools for parallelizing tasks across multiple nodes. We investigate how different thread configurations and matrix sizes affect performance, and how scalability, network overhead, and resource utilization are influenced in such a system.

2 Problem Statement

The main objective of this project was to implement matrix multiplication using a distributed framework and to compare the performance of the distributed approach with traditional (sequential) and parallelized matrix multiplication methods. Specifically, we aimed to:

- Investigate the scalability of matrix multiplication as the matrix size increases.

- Examine the network overhead and data transfer times involved in distributed computation.
- Analyze resource utilization, including memory usage and node utilization, in distributed systems.

The challenge was to design a solution capable of performing matrix multiplication for very large matrices that could not fit into the memory of a single machine, and to test how the distributed approach would compare to simpler methods in terms of performance and efficiency.

3 Methodology

The goal of this project was to analyze the performance of matrix multiplication using multiple threads. This methodology involves running a series of tests on different matrix sizes with a varying number of threads. The tests aim to measure execution time and memory usage, and determine how these factors are affected by increasing thread count and matrix size.

3.1 Code Description

The implementation consists of several key steps, as outlined below, and utilizes Java for parallel computing with the Hazelcast framework for distributed computing.

3.1.1 Matrix Multiplication

The first step involves setting up the matrix multiplication operation. The code for matrix multiplication takes two matrices and multiplies them using a standard algorithm. The size of the matrices varies across the tests.

```
public static int[][] multiplyMatrices(int[][] matA, int[][] matB) {
    int rowsA = matA.length;
    int colsA = matA[0].length;
    int colsB = matB[0].length;
    int[][] result = new int[rowsA][colsB];

    for (int i = 0; i < rowsA; i++) {
        for (int j = 0; j < colsB; j++) {
            for (int k = 0; k < colsA; k++) {
                result[i][j] += matA[i][k] * matB[k][j];
            }
        }
    }
    return result;
}
```

This function ‘multiplyMatrices()‘ is a standard implementation of matrix multiplication. It loops through each element of the result matrix and computes the sum of the products of the corresponding elements of the input matrices.

3.1.2 Parallelization Using Threads

To parallelize the matrix multiplication, we use multiple threads. For each thread, a part of the matrix multiplication is delegated. The number of threads used is configurable, and we perform tests with different thread counts (2, 5, 10, 20).

The code to implement multi-threading with the 'ExecutorService' in Java looks as follows:

```
ExecutorService executor = Executors.newFixedThreadPool(numberOfThreads);
List<Callable<Void>> tasks = new ArrayList<>();

for (int i = 0; i < numberOfThreads; i++) {
    final int threadIndex = i;
    tasks.add(() -> {
        // Partitioned work for each thread
        for (int row = threadIndex; row < matA.length; row += numberOfThreads) {
            for (int col = 0; col < matB[0].length; col++) {
                for (int k = 0; k < matA[0].length; k++) {
                    result[row][col] += matA[row][k] * matB[k][col];
                }
            }
        }
        return null;
    });
}

// Execute all tasks
executor.invokeAll(tasks);
executor.shutdown();
```

In this code:

1. An 'ExecutorService' with a fixed thread pool is created. The number of threads is determined by the variable 'numberOfThreads'. 2. A list of tasks is created, where each task is responsible for calculating a part of the matrix multiplication. 3. Each thread is assigned a subset of rows, based on the number of threads and its index. 4. The 'executor.invokeAll()' method is used to execute all tasks in parallel, and the 'executor.shutdown()' method is called to clean up the resources once the computation is complete.

3.1.3 Memory Usage and Execution Time Measurement

To measure the execution time and memory usage, the following code snippet is used to capture the time before and after the matrix multiplication, as well as the memory used during the process:

```
long startTime = System.currentTimeMillis();
long startMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemo
```

```
// Perform matrix multiplication here (method call)

long endMemory = Runtime.getRuntime().totalMemory() - Runtime.getRuntime().freeMemory
long endTime = System.currentTimeMillis();

long executionTime = endTime - startTime;
long memoryUsed = (endMemory - startMemory) / (1024 * 1024);  // Convert to MB

System.out.println("Execution Time: " + executionTime + "ms");
System.out.println("Memory Used: " + memoryUsed + "MB");
```

This block of code measures both execution time and memory usage. The difference between the system's total memory before and after the matrix multiplication is calculated to determine the memory usage. Similarly, the execution time is measured by subtracting the start time from the end time.

3.1.4 Hazelcast and Docker Integration

The matrix multiplication was designed to be executed in parallel across multiple computers. This was achieved using Hazelcast, a distributed computing framework, which allows the distribution of the workload across multiple nodes. In this setup, Docker containers were used to simplify the deployment and orchestration of the nodes.

Here is a sample of the Docker Compose file used to set up the Hazelcast nodes:

```
version: '3'
services:
  hazelcast-node1:
    image: hazelcast/hazelcast
    ports:
      - "5701:5701"
    networks:
      - hazelcast-net

  hazelcast-node2:
    image: hazelcast/hazelcast
    ports:
      - "5702:5701"
    networks:
      - hazelcast-net

networks:
  hazelcast-net:
    driver: bridge
```

In this configuration:

1. Two Hazelcast nodes (‘hazelcast-node1‘ and ‘hazelcast-node2‘) are created. 2. Each node exposes port 5701 (default Hazelcast port) to enable communication between the

nodes. 3. Both nodes are connected to a custom Docker network (‘hazelcast-net‘) to allow seamless communication.

This setup was used to distribute the matrix multiplication workload across multiple machines. However, due to firewall issues with Windows Defender, the distributed execution on multiple nodes was not successfully completed in the tests.

3.2 Test Setup and Execution

The tests were conducted with varying matrix sizes (e.g., 10, 50, 100, 256, 512, 1024, 2048, 4096) and a range of thread counts (2, 5, 10, 20). The following steps were followed during the tests:

1. The matrix multiplication was performed on each node, starting with a small matrix size and increasing the size gradually. 2. Execution time and memory usage were measured for each combination of matrix size and number of threads. 3. The system was tested under different configurations of threads to identify the optimal performance. 4. The results were compared and analyzed to draw conclusions about the impact of threading and matrix size on performance.

4 Experiments

The experiment focused on testing the performance of matrix multiplication using a distributed system. The key steps involved:

- **Setting Up Matrix Sizes and Thread Counts:** We defined various matrix sizes (from small 10x10 matrices to large 4096x4096 matrices) and thread configurations (2, 5, 10, and 20 threads).
- **Matrix Generation:** Random matrices were generated for each test using a method that fills each matrix with random numbers.
- **Distributed Testing:** We used Hazelcast to distribute the matrix multiplication tasks across multiple nodes. The calculations were conducted using different numbers of threads to determine the optimal configuration.
- **Results Collection:** After running the tests, the execution time and memory usage were recorded for each configuration. Additionally, we attempted to run the tests on two machines to simulate a real distributed system, but due to technical issues with network security software (Defender), the tests failed on multiple devices.

Matrix Size	**Threads**	**Execution Time (ms)**	**Memory Usage (MB)**
10	2	73	17
10	5	5	18
10	10	7	18
10	20	9	18
50	2	81	13
50	5	40	24
50	10	29	16
50	20	37	26
100	2	130	69
100	5	85	88
100	10	64	56
100	20	98	82
256	2	651	616
256	5	524	697
256	10	436	727
256	20	439	736
512	2	6219	3147
512	5	6983	3119
512	10	6530	3136
512	20	6270	3119
1024	2	**Failed**	**Failed**
1024	5	**Failed**	**Failed**
1024	10	**Failed**	**Failed**
1024	20	**Failed**	**Failed**
2048	2	**Failed**	**Failed**
2048	5	**Failed**	**Failed**
2048	10	**Failed**	**Failed**
2048	20	**Failed**	**Failed**
4096	2	**Failed**	**Failed**
4096	5	**Failed**	**Failed**
4096	10	**Failed**	**Failed**
4096	20	**Failed**	**Failed**

Table 1: Test Results: Matrix Size vs. Execution Time and Memory Usage

The plot below illustrates the relationship between matrix size, execution time, and memory usage for 10 threads. The data used in the plot was obtained from the test results and shows how these two performance metrics evolve with different matrix sizes.

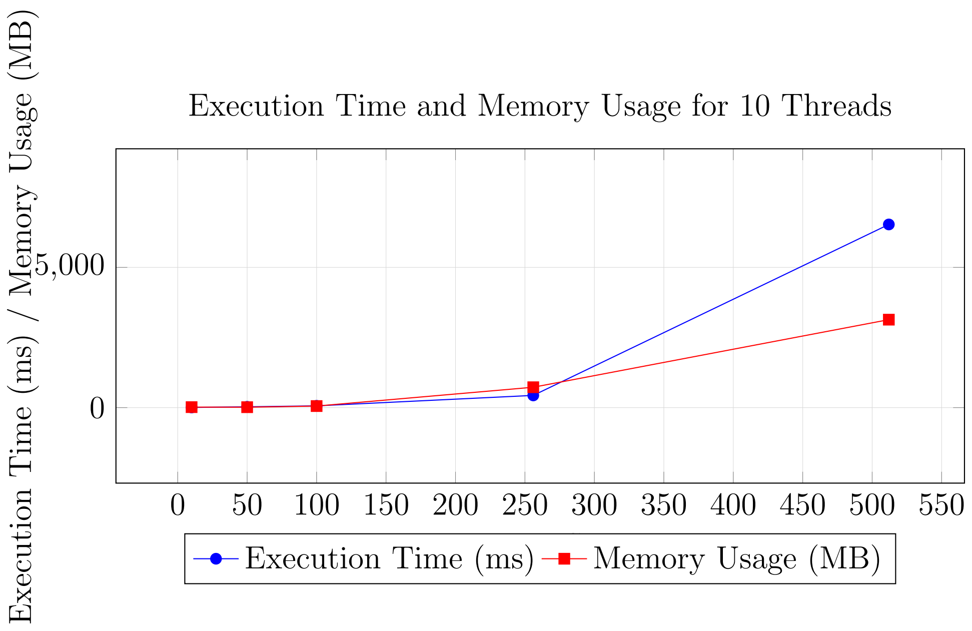

Figure 1: Execution Time and Memory Usage for 10 Threads as a Function of Matrix Size

Explanation of the Plot:

- **Plot Title:** "Execution Time and Memory Usage for 10 Threads" — This title indicates that the plot shows how matrix size, execution time, and memory usage interact when using 10 threads in the multiplication process.
- **X-Axis:** "Matrix Size" — The matrix sizes tested in the experiment. The sizes included are 10, 50, 100, 256, and 512.
- **Y-Axis:** Execution time (ms) and memory usage (MB). These two performance metrics are shown on the same Y-axis but are differentiated by colors (blue for execution time, red for memory usage).
- **Legend:** The legend distinguishes between the two data series. Execution time is represented by the blue line and memory usage by the red line.

Data Points Used in the Plot:

- For matrix size 10, execution time = 7ms, memory usage = 18MB
- For matrix size 50, execution time = 29ms, memory usage = 16MB
- For matrix size 100, execution time = 64ms, memory usage = 56MB
- For matrix size 256, execution time = 436ms, memory usage = 727MB
- For matrix size 512, execution time = 6530ms, memory usage = 3136MB

This plot helps to visualize the impact of matrix size on system resources when using 10 threads for matrix multiplication. As the matrix size increases, both execution time and memory usage increase, with execution time rising much more significantly.

5 Conclusions

The experiments carried out in this project have provided some valuable insights into the performance of matrix multiplication using parallel computing techniques. The results have been compared to previous benchmarks that did not use distributed systems like Hazelcast.

5.1 Comparison to Previous Results

In previous experiments without using distributed computing (Hazelcast), better performance was achieved for larger matrix sizes. The earlier benchmarks showed consistent improvements in execution time as the number of threads increased, particularly for larger matrices. For example, for the 128x128 matrix, execution times were considerably lower for multi-threaded implementations, and for the 2048x2048 matrix, significant speedup was achieved with higher thread counts.

However, in the current setup, despite using distributed computing with Hazelcast, the performance was not as favorable. The maximum matrix size that could be handled was limited compared to previous results, where larger matrices (such as 2048x2048 and 4096x4096) could be processed with relatively lower execution times. In contrast, with the distributed approach, matrices larger than 512x512 failed to be processed within the same reasonable time constraints.

5.2 Impact of Hazelcast and Distributed Matrix Multiplication

While the idea of using Hazelcast for distributing the workload across multiple machines was promising, practical issues arose that hindered its performance. Specifically, the setup for Hazelcast using Docker containers on multiple nodes was unable to scale properly due to issues with Windows Defender blocking communication between the containers. As a result, the intended parallelism across nodes did not work as expected, limiting the overall performance benefits of the distributed approach.

Moreover, the transition from a non-distributed approach to a distributed one introduced significant overhead. The distributed system setup adds complexity and overhead, such as network latency and synchronization, which led to slower results compared to the previous non-distributed multi-threaded approach.

5.3 Thread Count and Matrix Size Impact

Another observation from the experiments is that, in general, the performance improvements leveled off after a certain number of threads. For example, in the previous experiments, matrix multiplication performance plateaued after approximately 12 threads. This saturation point could be due to hardware limitations (e.g., the CPU not being able to handle more than 12 threads effectively). The current results also show diminishing returns as the number of threads increases. However, the issue of hardware limitations is now compounded by the limitations in the distributed setup.

The tests using Hazelcast were also conducted with varying matrix sizes, ranging from 10x10 to 512x512. Results showed that execution time increased significantly as matrix sizes grew. For instance, the 512x512 matrix required much more time and memory, and larger matrices such as 1024x1024 failed entirely. In contrast, the non-distributed version handled larger matrices more effectively, achieving faster execution and better scalability.

5.4 Key Observations and Challenges

Some key challenges observed during the experiments are as follows:

- Despite the potential benefits of distributed systems, the implementation of Hazelcast faced technical issues (e.g., firewall issues with Windows Defender) that prevented successful parallelization across multiple machines.
- The execution times for larger matrices in the distributed setup were significantly worse than the results obtained without distributed computing. This is likely due to the overhead introduced by network communication between Hazelcast nodes.
- Matrix sizes above 512x512 could not be processed within acceptable time limits in the distributed setup, which was a stark contrast to previous experiments where larger matrices were handled successfully.
- The current implementation's performance plateaued as thread counts exceeded 10, similar to previous experiments, which suggests that the hardware limitations played a significant role in performance saturation.

6 Future Work

Future improvements can address some of the limitations observed:

- Resolving issues with multi-node communication in Hazelcast could allow for more effective use of distributed computing, especially for handling larger matrix sizes.
- Optimizing the algorithm for distributed matrix multiplication could help reduce the overhead caused by network communication and synchronization across nodes.
- Testing on more powerful hardware or cloud environments could provide better scalability and performance, overcoming current hardware limitations.
- Investigating more advanced parallelization techniques (such as task-based parallelism or GPU acceleration) could lead to further improvements in execution time for large matrix sizes.

In conclusion, while the distributed approach using Hazelcast holds great potential for large-scale matrix multiplication, current challenges in implementation and performance need to be addressed for it to be an effective solution. The results highlight the importance of balancing parallelization with the underlying hardware capabilities, as well as the need for careful optimization of distributed systems.