Parallel Matrix Multiplication

Luis Perera Pérez
Grado en Ciencia e Ingeniería de Datos

Academic year 2024/25

Abstract

This paper addresses the challenge of efficiently multiplying large matrices in parallel. The study focuses on the implementation of a parallel version of matrix multiplication using multi-threading techniques. We evaluate the performance of the parallel algorithm through a series of benchmarks and compare it to the traditional sequential algorithm. The results demonstrate significant performance improvements with parallelization, particularly for large matrix sizes. The paper also explores potential future improvements in the algorithm.

Contents

1 Introduction

Matrix multiplication is a fundamental operation in many fields, including scientific computing, machine learning, computer graphics, and data analysis. Traditionally, matrix multiplication is performed sequentially, which can be inefficient for large matrices. With the advent of multi-core processors, parallel computing techniques offer an opportunity to significantly improve the performance of matrix multiplication by leveraging multiple CPU cores. This paper focuses on parallelizing the matrix multiplication process and benchmarking its performance.

2 Problem Statement

The challenge addressed in this paper is the inefficiency of performing matrix multiplication on large matrices using a sequential approach. As matrix sizes grow, the computational cost increases, leading to longer processing times. This paper seeks to explore how parallelism, specifically multi-threading, can be leveraged to speed up the matrix multiplication process. Additionally, we will investigate the efficiency of the parallel approach and how it scales with different matrix sizes and thread counts.

3 Methodology

This study evaluates the performance of parallel matrix multiplication using a multi-threaded approach and compares it with the traditional sequential algorithm. We tested both algorithms on matrices ranging from 10×10 to 2000×2000 and varied the number of threads (1, 2, and 4) to assess scalability.

The parallel algorithm uses a multi-threaded approach where matrix rows are divided among the threads, ensuring load balancing and concurrent execution. Each thread computes a portion of the result, and the final product is obtained by combining the individual results. This approach is expected to significantly reduce execution time for larger matrices by leveraging multiple cores of the processor.

The sequential algorithm computes the result element by element, processing one matrix element at a time. This method does not utilize multiple threads, and its execution time grows significantly as the size of the matrix increases.

Performance was measured by:

- **Execution Time:** The time taken for matrix multiplication to complete, indicating the speedup achieved through parallelization.
- **Memory Usage:** The amount of memory allocated during execution, reflecting the overhead introduced by multi-threading.

The comparison of these two algorithms provides insight into the trade-offs between speed and memory consumption as the number of threads increases, as well as the scalability of the parallel approach for large matrix sizes.

4 Experiments

Experiments were conducted on matrices of sizes 10×10, 100×100, 500×500, 1000×1000 and 2000×2000 using Java. The performance of the parallel matrix multiplication algorithm was evaluated with varying numbers of threads: 1, 2, and 4 threads. The goal was to assess the scalability of the parallel approach with increasing matrix sizes.

4.1 Parallel Matrix Multiplication Code

The following Java code implements the parallel matrix multiplication method, where the matrix rows are divided among available threads to distribute the computational load.

```
1  package org.example;
2
3  public class MatrixMultiplicationParallel {
4
5      // Parallel matrix multiplication method using multiple threads
6      public static double[][] multiplyMatricesParallel(double[][]
           matrixA, double[][] matrixB, int numThreads) {
7          int rows = matrixA.length;
8          int columns = matrixB[0].length;
9          double[][] resultMatrix = new double[rows][columns];
10
11         // Create an array of threads
12         Thread[] threads = new Thread[numThreads];
13         int rowsPerThread = rows / numThreads;
14
15         // Assign work to each thread
16         for (int i = 0; i < numThreads; i++) {
17             int startRow = i * rowsPerThread;
18             int endRow = (i == numThreads - 1) ? rows : startRow +
                   rowsPerThread;
19
20             // Initialize each thread with the work it should
                   perform
```

```
            threads[i] = new Thread(new MatrixMultiplier(matrixA,
                matrixB, resultMatrix, startRow, endRow));
            threads[i].start(); // Start each thread
        }

        // Wait for all threads to finish
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        return resultMatrix;
    }

    // Sequential matrix multiplication method
    public static double[][] multiplyMatricesSequential(double[][]
        matrixA, double[][] matrixB) {
        int rows = matrixA.length;
        int columns = matrixB[0].length;
        double[][] resultMatrix = new double[rows][columns];
        int columnsA = matrixA[0].length;

        // Perform multiplication row by row
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                double sum = 0;
                for (int k = 0; k < columnsA; k++) {
                    sum += matrixA[i][k] * matrixB[k][j];
                }
                resultMatrix[i][j] = sum;
            }
        }
        return resultMatrix;
    }

    // Inner class for running the multiplication task in parallel
    static class MatrixMultiplier implements Runnable {
        private final double[][] matrixA;
        private final double[][] matrixB;
        private final double[][] resultMatrix;
        private final int startRow;
        private final int endRow;

        // Constructor to initialize the matrices, result matrix
            and the range of rows
        public MatrixMultiplier(double[][] matrixA, double[][]
            matrixB, double[][] resultMatrix, int startRow, int
            endRow) {
            this.matrixA = matrixA;
            this.matrixB = matrixB;
            this.resultMatrix = resultMatrix;
            this.startRow = startRow;
            this.endRow = endRow;
        }
```

```
73
74          // Run method for the thread that does matrix
                multiplication for a given row range
75          @Override
76          public void run() {
77              int columnsB = matrixB[0].length;
78              int columnsA = matrixA[0].length;
79              for (int i = startRow; i < endRow; i++) {
80                  for (int j = 0; j < columnsB; j++) {
81                      double sum = 0;
82                      for (int k = 0; k < columnsA; k++) {
83                          sum += matrixA[i][k] * matrixB[k][j];
84                      }
85                      resultMatrix[i][j] = sum;
86                  }
87              }
88          }
89      }
90  }
```

4.2 Benchmark Setup

The performance of the parallel matrix multiplication was benchmarked using the Java Microbenchmarking Harness (JMH). Below is the JMH benchmark code used to test the parallel matrix multiplication algorithm.

4.2.1 Matrix Multiplication Benchmark Code

The following JMH benchmark was used to evaluate the performance of the parallel matrix multiplication algorithm:

```
1   package org.example;
2
3   import org.openjdk.jmh.annotations.*;
4   import java.util.concurrent.TimeUnit;
5   import java.util.Random;
6
7   @BenchmarkMode(Mode.AverageTime)
8   @OutputTimeUnit(TimeUnit.MILLISECONDS)
9   @State(Scope.Thread)
10  public class MatrixMultiplicationBenchmark {
11
12      private double[][] matrixA;
13      private double[][] matrixB;
14      private int numThreads = 4;  // Number of threads to use in the
            parallel method
15      private int n = 1000;         // Size of the matrices (n x n)
16
17      @Setup(Level.Trial)
18      public void setUp() {
```

```
        // Generate random matrices
        matrixA = generateRandomMatrix(n);
        matrixB = generateRandomMatrix(n);
    }

    // Benchmark for parallel matrix multiplication
    @Benchmark
    public double[][] testParallelMatrixMultiply() {
        return MatrixMultiplicationParallel.
            multiplyMatricesParallel(matrixA, matrixB, numThreads);
    }

    // Benchmark for sequential matrix multiplication
    @Benchmark
    public double[][] testSequentialMatrixMultiply() {
        return MatrixMultiplicationParallel.
            multiplyMatricesSequential(matrixA, matrixB);
    }

    // Method to generate a random matrix of size n x n
    private double[][] generateRandomMatrix(int n) {
        Random rand = new Random();
        double[][] matrix = new double[n][n];

        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                matrix[i][j] = rand.nextDouble() * 10;  // Random
                    numbers between 0 and 10
            }
        }
        return matrix;
    }
}
```

4.3 How to Run the Code from GitHub

To replicate the experiments, the code for matrix multiplication in Java is available in the following GitHub repository: `https://github.com/LuisPereraPerez/BigData`.

Follow these steps to download and execute the code:

- **Clone the repository:** Clone the repository by running the following command:

  ```
  git clone https://github.com/LuisPereraPerez/BigData
  ```

- **Requirements:**
  - Ensure that JDK 8 or higher is installed on your system.

  - Install the JMH (Java Microbenchmark Harness) plugin in IntelliJ IDEA or your preferred IDE.

- **Compilation:** Run the following command to compile the project and its dependencies:

```
mvn clean install
```

- **Execution:** Once the compilation is complete, run the program with the following command:

```
java -jar target/benchmarks-1.0.jar -prof gc
```

- **Varying Matrix Size and Number of Threads:** You can change the matrix size by adjusting the variable n in the code (for example, 10, 100, 500, 1000, or 2000). To modify the number of threads used in the parallel matrix multiplication, adjust the 'numThreads' variable in the code. The number of threads can be set to 1, 2, or 4 to test the scalability of the parallel implementation.

5 Results

This section presents the execution time, memory usage, and garbage collection (GC) counts for the different matrix multiplication algorithms implemented. Each metric is essential for evaluating performance and is described in detail below:

- **Execution Time (ms/op):** Represents the average time, in milliseconds per operation, that each algorithm takes to complete the matrix multiplication. A lower execution time indicates faster performance.
- **Memory Usage (MB/sec):** Refers to the rate at which memory is allocated during execution, measured in megabytes per second. Higher memory usage rates can signal increased memory demands, which may affect performance if the usage becomes excessive.

5.1 Execution Time and Memory Usage for 10×10 Matrices

The performance results for 10×10 matrix multiplication show that the sequential approach (1 thread) achieves an execution time of 1.727 ± 0.225 μs/op with

Table 1: Performance Results for 10×10 Matrix Multiplication Approaches

Algorithm	Execution Time (μs/op)	Memory Usage (MB/sec)
Sequential (1 Thread)	1.727 ± 0.225	534.716 ± 68.294
Parallel (2 Threads)	317.421 ± 16.108	6.362 ± 0.328
Parallel (4 Threads)	506.896 ± 12.001	6.119 ± 0.144

a memory usage of 534.716 ± 68.294 MB/sec. The parallel approach with 2 threads significantly increases execution time to 317.421 ± 16.108 μs/op, while reducing memory usage to 6.362 ± 0.328 MB/sec. With 4 threads, the execution time further increases to 506.896 ± 12.001 μs/op, with a slight decrease in memory usage to 6.119 ± 0.144 MB/sec. This indicates that, for small matrix sizes, the overhead of parallelism outweighs the benefits of multi-threading.

5.2 Execution Time and Memory Usage for 100×100 Matrices

Table 2: Performance Results for 100×100 Matrix Multiplication Approaches

Algorithm	Execution Time (ms/op)	Memory Usage (MB/sec)
Sequential (1 Thread)	1.209 ± 0.098	61.606 ± 5.013
Parallel (2 Threads)	1.018 ± 0.063	74.249 ± 4.610
Parallel (4 Threads)	0.877 ± 0.248	87.734 ± 22.626

The performance results for 100×100 matrix multiplication demonstrate that the sequential approach (1 thread) takes 1.209 ± 0.098 ms/op with a memory usage of 61.606 ± 5.013 MB/sec. The parallel approach with 2 threads improves execution time to 1.018 ± 0.063 ms/op, while increasing memory usage to 74.249 ± 4.610 MB/sec. With 4 threads, the execution time further decreases to 0.877 ± 0.248 ms/op, and memory usage increases again to 87.734 ± 22.626 MB/sec. These results show that parallelism reduces execution time, though memory usage continues to grow with more threads.

5.3 Execution Time and Memory Usage for 500×500 Matrices

The performance results for 500×500 matrix multiplication show a significant improvement in execution time with parallelism. The sequential approach (1 thread) has an execution time of 211.900 ± 8.116 ms/op, with a memory usage of 8.620 ± 0.326 MB/sec. Using 2 threads, the execution time reduces to 88.392

Table 3: Performance Results for 500×500 Matrix Multiplication Approaches

Algorithm	Execution Time (ms/op)	Memory Usage (MB/sec)
Sequential (1 Thread)	211.900 ± 8.116	8.620 ± 0.326
Parallel (2 Threads)	88.392 ± 15.600	17.089 ± 60.071
Parallel (4 Threads)	57.217 ± 2.981	31.942 ± 1.674

± 15.600 ms/op, but memory usage increases to 17.089 ± 60.071 MB/sec. The best performance is achieved with 4 threads, where the execution time decreases to 57.217 ± 2.981 ms/op, though memory usage rises to 31.942 ± 1.674 MB/sec. These results highlight the efficiency gains in execution time through parallelism, at the cost of increased memory usage.

5.4 Execution Time and Memory Usage for 1000×1000 Matrices

Table 4: Performance Results for 1000×1000 Matrix Multiplication Approaches

Algorithm	Execution Time (ms/op)	Memory Usage (MB/sec)
Sequential (1 Thread)	7,351.077 ± 36.523	1.006 ± 0.007
Parallel (2 Threads)	4,335.430 ± 88.392	1.698 ± 0.035
Parallel (4 Threads)	1,959.501 ± 285.666	3.741 ± 0.543

The performance results for 1000×1000 matrix multiplication demonstrate a clear benefit from parallel processing. The sequential approach (1 thread) has an execution time of 7,351.077 ± 36.523 ms/op, with a memory usage of 1.006 ± 0.007 MB/sec. With 2 threads, the execution time reduces to 4,335.430 ± 88.392 ms/op, and memory usage increases to 1.698 ± 0.035 MB/sec. Using 4 threads further reduces the execution time to 1,959.501 ± 285.666 ms/op, with a corresponding increase in memory usage to 3.741 ± 0.543 MB/sec. This highlights the substantial performance improvements in terms of execution time, although at the cost of increased memory consumption.

5.5 Execution Time and Memory Usage for 2000×2000 Matrices

The performance results for 2000×2000 matrix multiplication show significant improvements with parallelism. The sequential approach (1 thread) has an execution time of 106,733.126 ± 1,354.881 ms/op and a memory usage of 0.285 ± 0.004 MB/sec. With 2 threads, the execution time reduces to 54,910.998

Table 5: Performance Results for 2000×2000 Matrix Multiplication Approaches

Algorithm	Execution Time (ms/op)	Memory Usage (MB/sec)
Sequential (1 Thread)	106,733.126 ± 1,354.881	0.285 ± 0.004
Parallel (2 Threads)	54,910.998 ± 2,834.922	0.551 ± 0.029
Parallel (4 Threads)	30,120.052 ± 1,359.808	1.001 ± 0.039

± 2,834.922 ms/op, while memory usage increases to 0.551 ± 0.029 MB/sec. With 4 threads, the execution time further decreases to 30,120.052 ± 1,359.808 ms/op, and memory usage increases to 1.001 ± 0.039 MB/sec. These results show that parallel processing significantly reduces execution time at the cost of higher memory usage.

6 Conclusion

The performance analysis of the matrix multiplication approaches for various matrix sizes demonstrates the significant impact of parallelism on execution time, although memory usage increases with the number of threads.

For smaller matrices (10×10), the sequential approach performs adequately, with parallelism offering marginal improvements. However, as the matrix size increases (100×100 to 1000×1000), parallel execution shows substantial reductions in execution time, highlighting the benefits of utilizing multiple threads. The 2000×2000 matrix multiplication results further reinforce this trend, with the parallel (4 threads) approach offering a drastic reduction in execution time compared to the sequential method.

While parallelism leads to faster computation, it comes with an increase in memory usage, especially when moving from 1 to 4 threads. This increase in memory usage is expected, as parallel computation involves additional overhead such as thread management and memory allocation for each thread.

Overall, the results indicate that parallel matrix multiplication is highly beneficial for larger matrices, providing considerable performance gains in terms of speed. However, the trade-off between execution time and memory usage must be considered when choosing the optimal configuration. For smaller matrices, sequential computation remains competitive, but for larger problems, parallel execution offers a clear advantage, making it an essential approach for performance optimization in matrix multiplication tasks.

7 Future Work

Future work could focus on testing different parallelization strategies, such as OpenMP or SIMD instructions, to further improve the performance of matrix multiplication. Additionally, experimenting with different types of algorithms for matrix multiplication, such as Strassen's algorithm or divide-and-conquer approaches, may yield even more significant improvements. Finally, the impact of different hardware configurations and the number of threads can be studied to optimize the performance for specific use cases.

8 GitHub Repository

The code for matrix multiplication and benchmarking can be found in the following GitHub repository:

`https://github.com/LuisPereraPerez/BigData`