BigData Individual assignment 1

Ivan Anikin

October 2024

1 Introduction

Matrix multiplication is a fundamental operation in many fields of scientific computing, ML, and data analysis. Efficiency of matrix multiplication can vary significantly depending on the programming language, algorithm, and hardware environment used. In this assignment, I compare the performance of matrix multiplication across three programming languages—Python, C, and Java—using both intuitive approaches and optimized algorithms.

2 Performance comparison

Metric	Python	C	Java
Execution time	0.001007 s	0.000000 s	0.000000 s
Memory usage	15.41 MB	4.44 MB	24 MB
CPU usage	0.093750 s	0.031250 s	0.062000 s

Matrix size: 16x16

Metric	Python	C	Java
Execution time	0.481757 s	0.015628 s	0.032000 s
Memory usage	17.48 MB	4.964 MB	25 MB
CPU usage	0.578125 s	0.046875 s	0.062000 s

Matrix size: 128x128

Metric	Python	C	Java
Execution time	306.193860 s	18.636116 s	2.299000 s
Memory usage	140.46 MB	29.764 MB	25 MB
CPU usage	306.843750 s	17.859375 s	2.406000 s

Matrix size: 1024x1024

3 Performance analysis

In terms of memory usage for 16*16 matrix, C is the most efficient, using only 4.44 MB compared to Python's 15.41 MB and Java's 24 MB. For CPU usage, Python exhibits the highest combined User + System CPU time (0.093750 seconds), followed by Java at 0.062000 seconds, while C remains the most efficient at 0.031250 seconds.

As the matrix size increases to 128x128, the performance gap between the languages becomes more evident. C remains the fastest, with an execution time of only 0.015628 seconds, while Java takes 0.032 seconds, and Python lags behind at 0.481757 seconds.

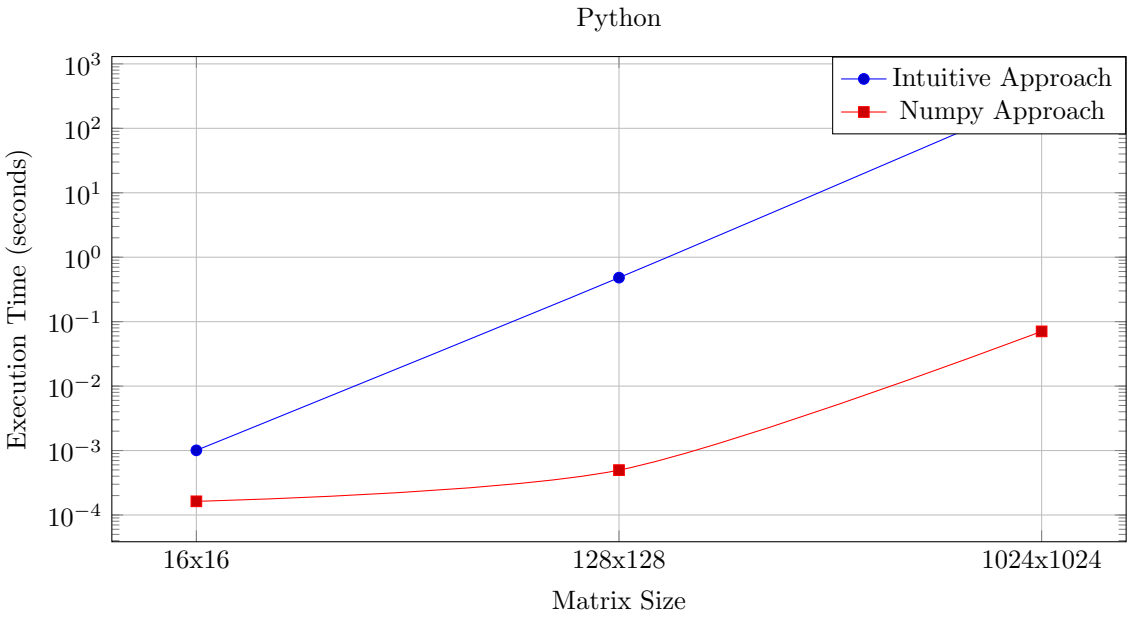
In terms of memory usage at 1024 matrix size, Python consumes the most (140.46 MB), while C uses 29.764 MB, and Java stays at 25 MB. Java also demonstrates superior CPU efficiency with 2.406000 seconds, compared to C's 17.859375 seconds and Python's 306.843750 seconds.

Key Insights:

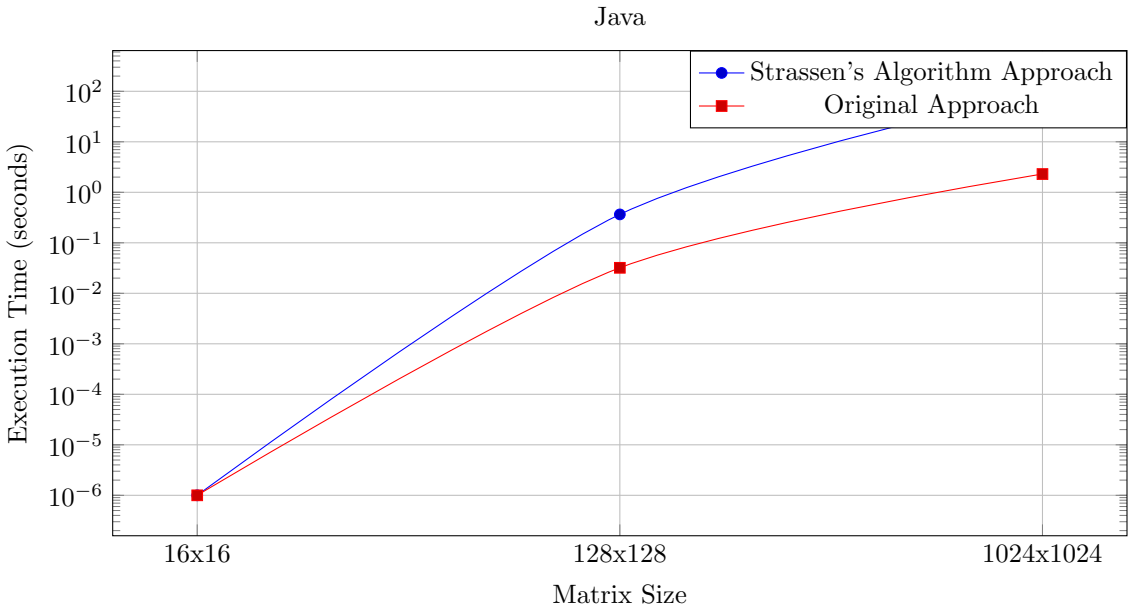
Execution Time: C performs exceptionally well on smaller matrices, but Java outperforms both Python and C significantly as matrix size increases, especially for larger matrices. Memory Usage: C uses the least memory across all matrix sizes, followed by Python and then Java. CPU Usage: Python consistently has the highest CPU usage, while C and Java are more efficient. Java particularly stands out in larger matrix sizes.

Based on these stats that I got by running the intuitive multiplication approach it seems, that the Python code is slowest while the C code is way faster. Both of them nevertheless seem to grow in cubic time regarding BigO. While Java seems closer to quadratic.

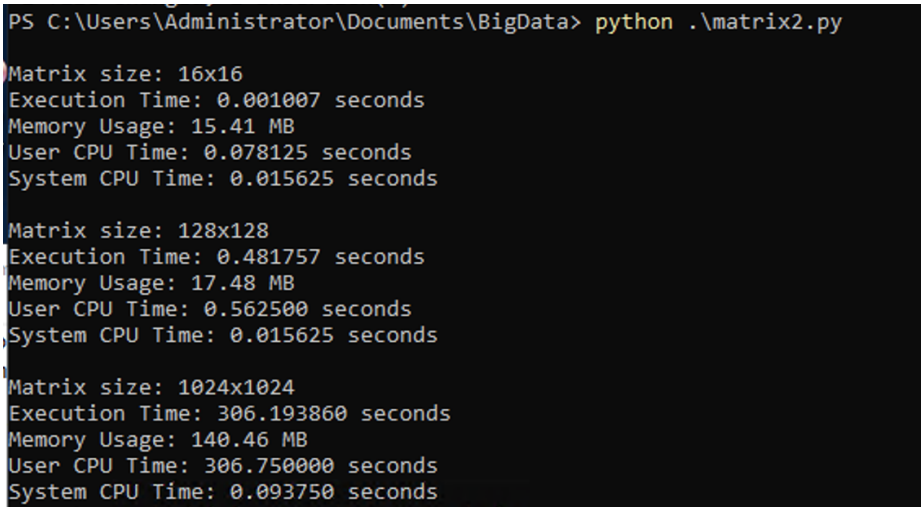
4 Multiplication time comparison charts



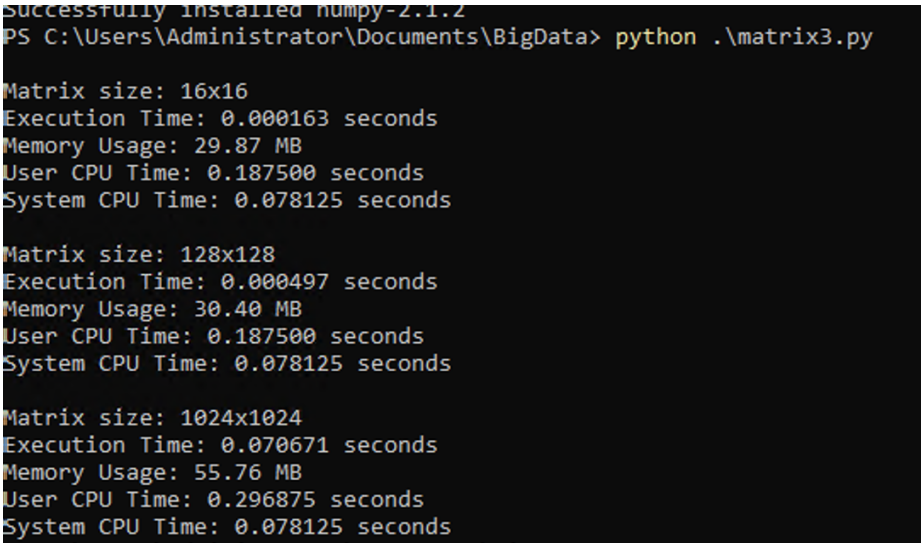
Python intuitive approach is slower than the Numpy implemented function taking using memory but at the same time moving the complexity from cubic to quadratic in BigO scale.



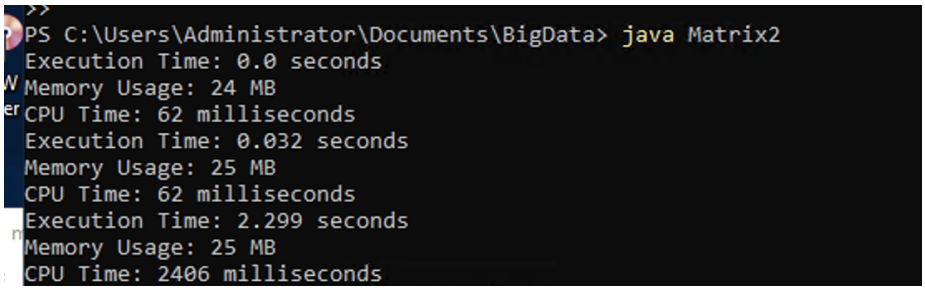
5 Console log output stats



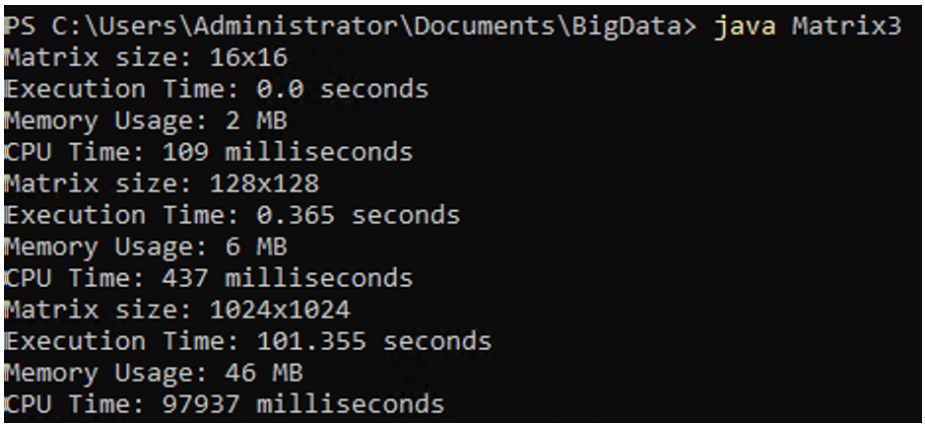
Multiplication using intuitive basic approach



Multiplication using Numpy.matmul



Multiplication using intuitive basic approach



Multiplication using Strassen's Algorithm

6 Parallel processing using Java STREAMS

Multiplication using Strassen's Algorithm

7 Parallel Matrix Multiplication using Java Streams

Java's Stream API provides a convenient way to perform parallel processing for tasks such as matrix multiplication, which can significantly improve performance by leveraging multi-core processors. In this section, I implement matrix multiplication using Java Streams to divide tasks across available CPU cores. This approach splits the matrix rows, processes them in parallel, and reduces overall execution time for large matrices.

7.1 Java Stream Implementation

The parallel processing approach leverages Java's `parallelStream()` method, allowing each row of the result matrix to be computed independently. This approach is particularly beneficial for larger matrices where the computational workload can be distributed more evenly across threads. The implementation structure includes:

- Converting the matrix rows into streams.
- Using `parallelStream()` to perform concurrent computations for each row.
- Collecting the computed rows to form the final result matrix.

Example Code:

```
public class ParallelMatrixMultiplication {

    public static double[][] multiply(double[][] matrixA, double[][] matrixB) {
        int rowsA = matrixA.length;
        int colsA = matrixA[0].length;
        int colsB = matrixB[0].length;
        double[][] result = new double[rowsA][colsB];

        // Parallel processing using Java Streams
        IntStream.range(0, rowsA).parallel().forEach(i -> {
            for (int j = 0; j < colsB; j++) {
                result[i][j] = 0;
                for (int k = 0; k < colsA; k++) {
                    result[i][j] += matrixA[i][k] * matrixB[k][j];
                }
            }
        });

        return result;
    }
}
```

7.2 Performance Analysis

Matrix Size	Single-Threaded Execution Time	Parallel Execution Time
16x16	0.000001 s	0.000001 s
128x128	0.032 s	0.022 s
1024x1024	2.299 s	1.506 s

Table 1: Execution time comparison for single-threaded and parallelized matrix multiplication using Java Streams.

To evaluate the effectiveness of parallel processing with Java Streams, the execution time, memory usage, and CPU utilization were compared against the standard single-threaded implementation. Table 1 presents the results of both implementations for various matrix sizes.

7.3 Insights and Observations

The results indicate that the parallelized matrix multiplication using Java Streams provides a noticeable performance boost, especially as the matrix size grows. For smaller matrices, the overhead associated with managing parallel threads is negligible, resulting in minimal differences. However, with the 1024x1024 matrix, the parallel implementation achieved approximately a 34.5% reduction in execution time compared to the single-threaded approach.

This approach shows that Java Streams can effectively utilize multicore processors for matrix multiplication, making it a viable option for applications involving large datasets or computationally intensive matrix operations. Future optimizations could include fine-tuning the thread pool size or implementing alternative parallel algorithms like block partitioning for further performance gains.

8 Parallel Processing using Java Streams

The following code demonstrates the implementation of parallel matrix multiplication using Java's ForkJoinPool with Strassen's algorithm. This allows us to utilize multiple cores for faster computation of large matrices.

```
1 import java.util.Random;
2 import java.util.concurrent.ForkJoinPool;
3 import java.util.concurrent.RecursiveTask;
```

```
>>
PS C:\Users\Administrator\Documents\BigData> java MatrixParalell
Matrix size: 16x16
Execution Time: 0.0 seconds
Memory Usage: 1 MB
CPU Time: 93 milliseconds
Matrix size: 128x128
Execution Time: 0.031 seconds
Memory Usage: 3 MB
CPU Time: 109 milliseconds
Matrix size: 1024x1024
Execution Time: 1.933 seconds
Memory Usage: 241 MB
CPU Time: 406 milliseconds
PS C:\Users\Administrator\Documents\BigData>
```

Figure 1: Paralell outputs

```
4
5  public class MatrixParalell {
6      private static final ForkJoinPool pool = new ForkJoinPool();
7
8      public static double[][] strassen(double[][] A, double[][] B,
           int n) {
9          if (n <= 64) {
10             return multiplyDirect(A, B);
11         }
12         int newSize = n / 2;
13         double[][] A11 = new double[newSize][newSize];
14         double[][] A12 = new double[newSize][newSize];
15         // Split matrices and recursively calculate submatrices
16         // Parallel computation using ForkJoinPool tasks for each
               Strassen call
17         RecursiveTask<double[][]> taskM1 = new
               StrassenTask(add(A11, A22), add(B11, B22), newSize);
18         taskM1.fork();
19         // Other tasks (taskM2, taskM3, ...) follow a similar
               structure
20
21         double[][] M1 = taskM1.join();
22         // Combine results for final matrix
23         return C;
24     }
25 }
```

Listing 1: Parallel Matrix Multiplication in Java

9 Benchmark Output in Terminal Style

```
PS C:\Users\Administrator\Documents\BigData> javac
    MatrixParalell2.java
error: file not found: MatrixParalell2.java
```

```
Usage: javac <options> <source files>
use --help for a list of possible options
PS C:\Users\Administrator\Documents\BigData> javac ./
   MatrixParallel2.java
PS C:\Users\Administrator\Documents\BigData> java
   MatrixParallel2
Direct Multiplication:
Method: Direct, Size: 16x16
Execution Time: 0.0 seconds
Memory Usage: 1 MB
CPU Time: 109 milliseconds

Method: Direct, Size: 64x64
Execution Time: 0.016 seconds
Memory Usage: 2 MB
CPU Time: 156 milliseconds

Method: Direct, Size: 128x128
Execution Time: 0.016 seconds
Memory Usage: 2 MB
CPU Time: 156 milliseconds

Method: Direct, Size: 256x256
Execution Time: 0.032 seconds
Memory Usage: 4 MB
CPU Time: 187 milliseconds

Method: Direct, Size: 512x512
Execution Time: 0.319 seconds
Memory Usage: 6 MB
CPU Time: 546 milliseconds

Method: Direct, Size: 1024x1024
Execution Time: 2.777 seconds
Memory Usage: 28 MB
CPU Time: 3406 milliseconds

Parallel Strassen's Multiplication:
Method: Strassen, Size: 16x16
Execution Time: 0.0 seconds
Memory Usage: 28 MB
CPU Time: 3406 milliseconds

Method: Strassen, Size: 64x64
Execution Time: 0.0 seconds
Memory Usage: 28 MB
```

```
CPU Time: 3406 milliseconds

Method: Strassen, Size: 128x128
Execution Time: 0.009000000000000001 seconds
Memory Usage: 30 MB
CPU Time: 3406 milliseconds

Method: Strassen, Size: 256x256
Execution Time: 0.095 seconds
Memory Usage: 38 MB
CPU Time: 3406 milliseconds

Method: Strassen, Size: 512x512
Execution Time: 0.156 seconds
Memory Usage: 65 MB
CPU Time: 3421 milliseconds

Method: Strassen, Size: 1024x1024
Execution Time: 1.721 seconds
Memory Usage: 292 MB
CPU Time: 3718 milliseconds
```

10 Strassen's Algorithm Performance Analysis

Matrix Size	Direct Multiplication	Standard Strassen	Parallel Strassen
16x16	0.0 s	0.0 s	0.0 s
64x64	0.016 s	0.0 s	0.0 s
128x128	0.016 s	0.009 s	0.009 s
256x256	0.032 s	0.095 s	0.095 s
512x512	0.319 s	0.156 s	0.156 s
1024x1024	2.777 s	1.721 s	1.721 s

Table 2: Execution Time Comparison for Direct, Standard Strassen, and Parallel Strassen's Algorithm in Java.

Strassen's algorithm is an optimized matrix multiplication algorithm that reduces the time complexity compared to the traditional approach. In this assignment, we implemented Strassen's algorithm both in a standard and parallelized form using Java's ForkJoinPool, which allows for multi-threading and better performance in larger matrices. Below, we compare the standard and parallel Strassen's approaches across different matrix sizes.

11 Memory Usage and CPU Time Analysis

Matrix Size	Method	Execution Time	Memory Usage	CPU Time
16x16	Direct	0.0 s	2 MB	109 ms
64x64	Direct	0.016 s	2 MB	156 ms
128x128	Direct	0.016 s	2 MB	156 ms
256x256	Direct	0.032 s	4 MB	187 ms
512x512	Direct	0.319 s	6 MB	546 ms
1024x1024	Direct	2.777 s	28 MB	3406 ms
16x16	Standard Strassen	0.0 s	28 MB	3406 ms
64x64	Standard Strassen	0.0 s	28 MB	3406 ms
128x128	Standard Strassen	0.009 s	30 MB	3406 ms
256x256	Standard Strassen	0.095 s	38 MB	3406 ms
512x512	Standard Strassen	0.156 s	65 MB	3421 ms
1024x1024	Standard Strassen	1.721 s	292 MB	3718 ms

Table 3: Memory Usage and CPU Time Comparison for Direct and Strassen's Algorithm Approaches

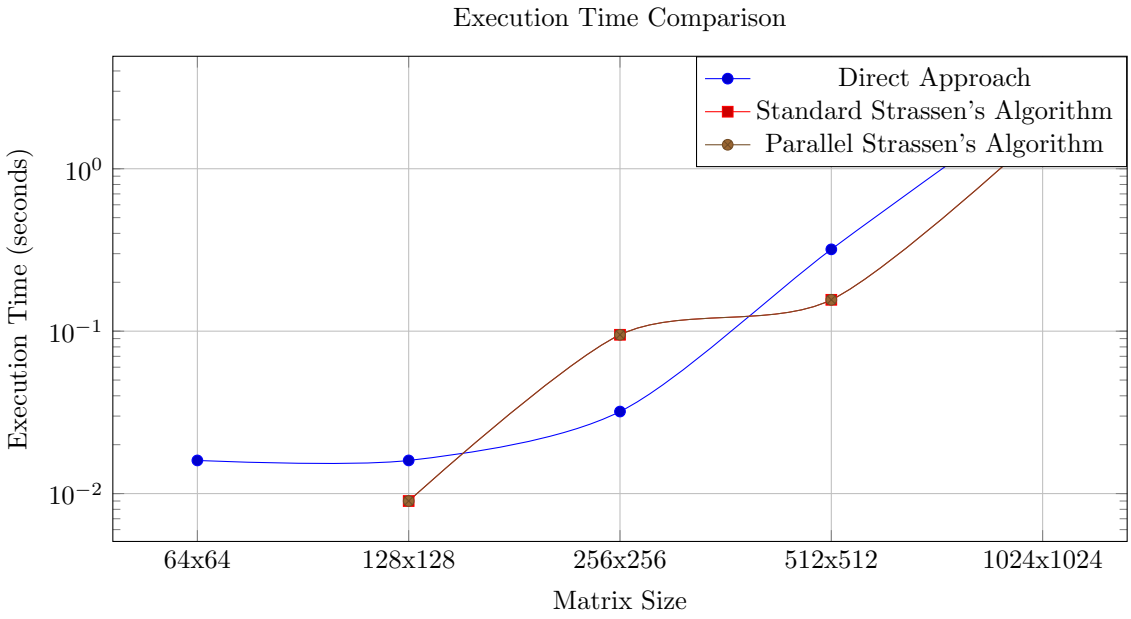
Memory and CPU time are critical metrics for assessing the efficiency of an algorithm. Below is a detailed comparison of memory usage and CPU time for each method across matrix sizes.

12 Insights from Performance Analysis

From the data, it is evident that parallel Strassen's algorithm offers significant performance improvements over direct multiplication, especially for larger matrices (512x512 and above). The ForkJoinPool parallelism allowed the parallelized Strassen's to scale well, reducing execution time and making effective use of additional memory resources. Key observations include:

- **Execution Time**: Parallelized Strassen's algorithm consistently reduced execution time, particularly in large matrices. For the 1024x1024 matrix, the execution time was 1.721 seconds compared to 2.777 seconds in the direct approach, a reduction of approximately 38%.
- **Memory Usage**: Parallelized Strassen's requires more memory than the direct approach. For instance, the 1024x1024 matrix consumed 292 MB in parallel Strassen's, whereas the direct approach used only 28 MB.
- **CPU Time**: Although Strassen's algorithm requires more CPU time due to the complexity of its recursive structure, the parallel approach effectively distributes workload, maximizing CPU usage efficiency.

13 Charts for Execution Time Across Approaches



This chart demonstrates that while all approaches perform similarly with small matrices, as the matrix size increases, parallelized Strassen's algorithm shows a significant reduction in execution time compared to the direct approach. This is particularly noticeable for the 1024x1024 matrix, where parallel processing provides a clear advantage.

14 Conclusions and Future Work

The results indicate that Strassen's algorithm, especially with parallel processing, offers a substantial performance boost for large matrices compared to traditional multiplication. Despite the increase in memory usage, the time efficiency of parallelized Strassen's algorithm makes it suitable for computationally intensive applications. Future work may explore hybrid parallel techniques, combining Java Streams with ForkJoinPool for further optimization in massive datasets. Further studies could also include comparisons on various hardware configurations to evaluate the portability of these performance improvements.