CS 4504 Project Report - Part 2

William Vaselaros, Graham Allen

Parallel Distributed Computing
Section W01

Fall 2024

Abstract

This report will explore the integration of distributed and parallel computing paradigms by implementing Client-Server architecture using TCP. This Client-server architecture leverages multithreading to optimize matrix computations. The system has been designed to facilitate communication between a client, a server router, and a server. This communication involves sending randomly generated matrices to a server that will apply Strassen's matrix manipulation algorithm with a server-router acting as a bridge. The hybrid design of the system allows elements of distributed client-server computing with parallel processing to optimize performance, flexibility, and scalability.

The goal of the project is to implement Strassen's matrix multiplication algorithm for matrix-chain multiplication[1]. The server must implement a binary tree structure where the leaf nodes represent an individual matrix in order to divide the work across multiple threads or cores where smaller subsections of each matrix are solved in parallel. The matrices will then combine back together to produce a final product.

To evaluate performance, the system is tested with a different number of threads or cores (1, 3, 7, 15, 31) and matrix sizes that range from 50 x 50, 75 x 75, 100 x 100, 150 x 150, and 200 x 200. The collected data is used to calculate the execution or run time, the speed up, and the overall efficiency of the system. The results provide insights into how the system performs under different conditions and how well the system will scale as needed.

Introduction

This report presents the design and implementation of a hybrid computing system that combines distributed client-server communication with parallel processing techniques. A TCP-based client-server architecture is, supported by a server-router acting as a bridge, implemented for effective communication over a locally hosted network. This architecture facilitates dynamic and scalable communication for multiple clients and server threads. By leveraging multithreading the server performs intensive computational tasks in parallel further showcasing the benefits that a distributed system provides in computationally heavy scenarios.

Implementing a TCP-based client-server architecture offers seamless communication across a network. TCPClient will randomly generate an N x N matrix that will be converted into a string for future transmission. By using a BufferedReader and a PrintWriter the converted matrix can be sent to the TCPServerRouter which acts as a transportation hub for both the client and Server. Once TCPServerRouter receives the String matrix it will convert it back to a two-dimension integer array and perform Strassen's algorithm through a binary tree structure. Once the matrix has undergone Strassen's algorithm calculations[1] run times, speed up, and efficiency will be calculated and stored. These three variables

will then be converted into Strings and sent back to the TCPServerRouter. Here these values can be observed and tabulated for further analysis.

While Strassen's algorithm is known for its ability to multiply matrices it encounters performance issues the larger the number of matrices get. The data presented in the **Tabulated Data** section of the report offers an in-depth evaluation of the run time, speed up, and efficiency that a parallel and distributed implementation of Strassen's algorithm offers.

Design Approach

The project follows a TCP-based client-server communication architecture that implements Strassen's algorithm for matrix-chain multiplication over a locally-hosted network as seen in *Figure 1*.

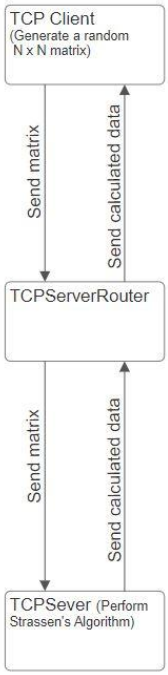


Figure 1: **Base TCP Structure**

This is the project's base design, simply sending the randomly generated matrix from TCPClient to TCPServer, with the TCPServerRouter acting as a bridge between the two. TCPClient will randomly generate a matrix of N x N size, and TCPServer will receive the matrix, perform Strassen's algorithm[1] on said matrix, and send back data for the total run time, speedup, and efficiency.

In order to demonstrate the benefits that parallel and distributed computing can offer, a binary tree structure is used alongside Strassen's algorithm[1] where each leaf node will represent an individual matrix for calculation.

Implementation

TCPClient

To achieve a client-server architecture a client class must be implemented and is named TCPClient.java for readability.

To ensure a proper client-server architecture TCP protocol must be implemented. TCPClient achieves this by connecting to the router on a specified IP address and port number (5555). This establishes a socket connection amongst the design architecture and ensures a connection amongst the network. This architecture is seen in *Code Snippet 1*.

```
// Variables for setting up connection and communication
Socket Socket = null;
PrintWriter out = null;
BufferedReader in = null;
InetAddress addr = InetAddress.getLocalHost();
String host = addr.getHostAddress();
String routerName = "localhost";
int SockNum = 5555;

// Tries to connect to the ServerRouter
try {
    Socket = new Socket("[redacted]", SockNum);
    out = new PrintWriter(Socket.getOutputStream(), true);
    in = new BufferedReader(new InputStreamReader(Socket.getInputStream()));
} catch (UnknownHostException e) {
    System.err.println("Don't know about router: " + routerName);
    System.exit(1);
} catch (IOException e) {
    System.err.println("Couldn't get I/O for the connection to: " + routerNa
    System.exit(1);
}

String fromServer;
String address = "127.0.0.1"; // destination IP (Server)
```

Code Snippet 1: **Client Communication**

For privacy sake there is a green line covering the host client's IPV-4 address. TCPClient also uses a PrintWriter to send outgoing messages to the server and a BufferedReader to read incoming messages from the server. The last line in *Code Snippet 1* holds the IP address for the local host. This address is important as it will keep track of where the information from TCPClient is sent to.

To perform Strassen's algorithm for matrix multiplication there must be a matrix that can be sent from TCPClient to TCPServer. TCPClient contains a method named generateRandomMatrix(int N) where an N x N matrix is randomly generated and each position in the two-dimensional array can be filled with an integer from 0 - 99. It is important that for Strassen's algorithm to work with the binary tree structure this N value must be an even number. This is implemented through a nested for loop as seen in *Code Snippet 2*.

```
private static int[][] generateRandomMatrix(int n) {
    int[][] matrix = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            matrix[i][j] = (int)(Math.random() * 100);
        }
    }
    return matrix;
}
```

Code Snippet 2: **Generate Random Matrix**

When trying to send the matrix through TCPServerRouter to the TCPClient our team ran into an issue where the matrix would not send properly through each socket. To solve this issue there is a method named matrixToString() that takes in the randomly generated matrix and converts it to a semicolon and comma-delimited string. This implementation can be seen in *Code Snippet 3*.

```
private static String matrixToString(int[][] matrix) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < matrix.length; i++) {
        for (int j = 0; j < matrix.length; j++) {
            sb.append(matrix[i][j]);
            if (j < matrix.length - 1) sb.append(",");
        }
        if (i < matrix.length - 1) sb.append(";");
    }
    return sb.toString();
}
```

Code Snippet 3: **Matrix to String**

Converting the matrix into a string is essential for the client to send the matrix through the designed architecture. Involving both a BufferedReader and a PrintWriter to ensure the information is received and sent throughout the network.

TCPServerRouter

TCPSeverRouter acts as the bridge between TCPClient and TCPServer. The router listens for incoming connections and stores them in a simple routing table. The routing table, as seen

in *Code Snippet 4*, can store the routing information of up to ten connections.

```
Object [][] RoutingTable = new Object [10][2]; // routir
```

Code Snippet 4: **Routing Table**

The first column of the routing table stores the IP address of either the Client or Server that is connecting to it. The second column stores the corresponding socket that the connection is made on.

TCPServerRouter uses a ServerSocket as opposed to TCPClient and TCPServer who both use a normal socket. TCPServerRouter is hardcoded to use port 5555 as seen in *Code Snippet 5*.

```
ServerSocket serverSocket = null; // server socket for accepting conn
try {
   serverSocket = new ServerSocket(5555);
   System.out.println("ServerRouter is Listening on port: 5555.");
}
    catch (IOException e) {
      System.err.println("Could not listen on port: 5555.");
      System.exit(1);
   }
```

Code Snippet 5: **Server Socket**

This server socket allows for the router to connect to either TCPClient or TCPServer without the need for another socket.

Once a connection has been made the router will create an SThread, see SThread implementation below on page 7, which is used to store the connection information into the routing table. The router will then state that a connection has been made with the client or server that requested the connection. See *Code Snippet 6* for more details.

```
// Creating threads with accepted connections
while (Running == true)
{
try {
   clientSocket = serverSocket.accept();
   SThread t = new SThread(RoutingTable, clientSocket, ind); // creates a thread with a random port
   t.start(); // starts the thread
   ind++; // increments the index
System.out.println("ServerRouter connected with Client/Server: " + clientSocket.getInetAddress().getHostAddress());
}
 catch (IOException e) {
   System.err.println("Client/Server failed to connect.");
   System.exit(1);
}
}//end while
```

Code Snippet 6: **SThread Creation**

TCPServer

The TCPServer class handles the implementation of Strassen's matrix multiplication algorithm. TCPServer contains a simple tree data structure that will be used to implement threading as needed for matrix multiplication. Before any calculations can be performed the String that TCPClient sent must be converted back into a real matrix, this can be seen in *Code Snippet 7* below.

```
private static int[][] stringToMatrix(String str, int n) {
    String[] rowStrings = str.split(";");
    int[][] matrix = new int[n][n];

    for (int i = 0; i < n; i++) {
        String[] values = rowStrings[i].split(",");
        for (int j = 0; j < n; j++) {
            matrix[i][j] = Integer.parseInt(values[j]);
        }
    }
    return matrix;
}
```

Code Snippet 7: **String to Matrix**

With the String being converted back into a matrix calculations can begin. The matrix will be entered into an array list of type int[][] where a binary tree will be built as seen in *Code Snippet 8* below. Placing each section of the matrix into a binary tree is important for using threads to potentially increase the speed and efficiency.

```java
private static Node buildTree(List<int[][]> matrices) {
    if (matrices.isEmpty()) return null;
    if (matrices.size() == 1) return new Node(matrices.get(0));

    Queue<Node> nodes = new LinkedList<>();
    for (int[][] matrix : matrices) {
        nodes.add(new Node(matrix));
    }

    while (nodes.size() > 1) {
        Node left = nodes.poll();
        Node right = nodes.poll();
        Node parent = new Node(null);
        parent.left = left;
        parent.right = right;
        parent.isDone = false;
        nodes.add(parent);
    }
    return nodes.poll();
}
```

Code Snippet 8: **Building the Tree**

Once the tree has been constructed a helper method for multiplication will be called, seen in *Code Snippet 9.1* below. The tree structure will consist of leaf nodes that hold individual matrices while the internal nodes hold the result of their multiplication. This design choice is crucial to implementing parallelism.

```java
private static void startMultiplication(Node node) {
    if (node == null || node.isDone) return;
    startMultiplication(node.left);
    startMultiplication(node.right);
    node.multiplyThread = new StrassensThread(node);
    node.multiplyThread.start();
}
```

Code Snippet 9.1: **Start Multiplication**

```java
//strassens multiplication implementation
private int[][] multiply(int[][] matrixA, int[][] matrixB) {
    int colA = matrixA[0].length;
    int rowA = matrixA.length;
    int colB = matrixB[0].length;
    int rowB = matrixB.length;

    if (colA != rowB) {
        System.out.println("\nError: The number of columns in Matrix A must be equal to the number of rows in Matrix B\n");
        int[][] temp = new int[1][1];
        temp[0][0] = 0;
        return temp;
    }

    int[][] resultMatrix = new int[rowA][colB];
    initWithZeros(resultMatrix, rowA, colB);

    if (colA == 1) {
        resultMatrix[0][0] = matrixA[0][0] * matrixB[0][0];
    } else {
        int splitIndex = colA / 2;

        int[][] result00 = new int[splitIndex][splitIndex];
        int[][] result01 = new int[splitIndex][splitIndex];
        int[][] result10 = new int[splitIndex][splitIndex];
        int[][] result11 = new int[splitIndex][splitIndex];

        int[][] a00 = new int[splitIndex][splitIndex];
        int[][] a01 = new int[splitIndex][splitIndex];
        int[][] a10 = new int[splitIndex][splitIndex];
        int[][] a11 = new int[splitIndex][splitIndex];
        int[][] b00 = new int[splitIndex][splitIndex];
        int[][] b01 = new int[splitIndex][splitIndex];
        int[][] b10 = new int[splitIndex][splitIndex];
        int[][] b11 = new int[splitIndex][splitIndex];

        int[][] matrices[] = {result00, result01, result10, result11,
                              a00, a01, a10, a11, b00, b01, b10, b11};

        for (int[][] matrix : matrices) {
            initWithZeros(matrix, splitIndex, splitIndex);
        }

        for (int i = 0; i < splitIndex; i++) {
            for (int j = 0; j < splitIndex; j++) {
                a00[i][j] = matrixA[i][j];
                a01[i][j] = matrixA[i][j + splitIndex];
                a10[i][j] = matrixA[splitIndex + i][j];
                a11[i][j] = matrixA[i + splitIndex][j + splitIndex];
                b00[i][j] = matrixB[i][j];
                b01[i][j] = matrixB[i][j + splitIndex];
                b10[i][j] = matrixB[splitIndex + i][j];
                b11[i][j] = matrixB[i + splitIndex][j + splitIndex];
            }
        }

        addMatrix(multiply(a00, b00), multiply(a01, b10), result00, splitIndex);
        addMatrix(multiply(a00, b01), multiply(a01, b11), result01, splitIndex);
        addMatrix(multiply(a10, b00), multiply(a11, b10), result10, splitIndex);
        addMatrix(multiply(a10, b01), multiply(a11, b11), result11, splitIndex);

        for (int i = 0; i < splitIndex; i++) {
            for (int j = 0; j < splitIndex; j++) {
                resultMatrix[i][j] = result00[i][j];
                resultMatrix[i][j + splitIndex] = result01[i][j];
                resultMatrix[splitIndex + i][j] = result10[i][j];
                resultMatrix[i + splitIndex][j + splitIndex] = result11[i][j];
            }
        }
    }
    return resultMatrix;
}
```

Code Snippet 9.2: **Strassen's implementation**

Once the received matrix has undergone Strassen's algorithm, TCPServer will print out data for sequential time, parallel time, speed up, threads used, and efficiency. The following code in *Code Snippet 10* shows the formulas used to calculate all performance metrics.

```
if (matrices.size() == numMatrices) {
    Node root = buildTree(matrices);

    long startParallel = System.nanoTime();
    int[][] resultMatrix = null;
    try {
        resultMatrix = startMultiplication(root);
    } catch (InterruptedException e) {
        System.out.println("Multiplication interrupted: " + e.getMessage());
    }
    double parallelTime = (System.nanoTime() - startParallel) / 1e9;

    int threadsUsed = numMatrices - 1;
    long startSeq = System.nanoTime();
    int[][] seqResult = matrices.get(0);
    for (int i = 1; i < matrices.size(); i++) {
        seqResult = new StrassensThread(null).multiply(seqResult, matrices.get(i));
    }
    double seqTime = (System.nanoTime() - startSeq) / 1e9;
    double speedup = seqTime / parallelTime;
    double efficiency = speedup / threadsUsed;

    // Print the result matrix

    System.out.println("Sequential time: " + seqTime);
    System.out.println("Parallel time: " + parallelTime);
    System.out.println("Speedup: " + speedup);
    System.out.println("Threads used: " + threadsUsed);
    System.out.println("Efficiency: " + efficiency * 100);
    processingComplete = true;
}
```

Code Snippet 10: **Calculating Performance Metrics**

Performance metrics must be calculated in order to assess how viable a parallel system is for a computationally heavy scenario. All performance metrics will be analyzed in the **Tabulated Data** section below on pages 9 through 11.

To ensure a proper client-server architecture TCP protocol must be implemented. TCPServer achieves this by connecting to the router through the local host IP address (127.0.0.1) and port number (5555). This establishes a socket connection amongst the design architecture and ensures a connection amongst the network. This architecture is seen below in *Code Snippet 11*.

```
try {
    Socket = new Socket("localhost", 5555);
    out = new PrintWriter(Socket.getOutputStream(), true);
    in = new BufferedReader(new InputStreamReader(Socket.getInputStream()));
} catch (UnknownHostException e) {
    System.err.println("Could not connect to router");
    System.exit(1);
} catch (IOException e) {
    System.err.println("Couldn't get I/O for the connection");
    System.exit(1);
}

String fromClient;
String address = "[redacted]"; // Server's address
```

Code Snippet 11: **Server Communication**

TCPServer uses a PrintWriter to send outgoing messages to the server and a BufferedReader to read incoming messages from the server. The last line in *Code Snippet 11* holds the host computer's IPV-4 address and for privacy's sake is covered by a green line. This address is important as it will ensure that the newly calculated matrix and performance metrics can be output.

SThread

SThread is a fairly simple class involving a PrintWriter and a BufferedReader to handle incoming and outgoing data. A SThread is created through a constructor that must have the Routing table, a socket for where the data is being sent to, and an index. This can be seen in *Code Snippet 12* below.

```
public SThread(Object [][] Table, Socket toClient, int index) throws IOException
{
    out = new PrintWriter(toClient.getOutputStream(), true);
    in = new BufferedReader(new InputStreamReader(toClient.getInputStream()));
    RTable = Table;
    addr = toClient.getInetAddress().getHostAddress();
    RTable[index][0] = addr;
    RTable[index][1] = toClient;
    ind = index;
}
```

Code Snippet 12: **SThread Constructor**

SThread will increment through the Routing table in order to find a destination that is similar to the one given by the server router, as seen below in *Code Snippet 13*. This ensures communication is correct. Who receives information in what order is crucial to the project and must be kept track of.

```
try {
    destination = in.readLine();
    System.out.println("Forwarding to " + destination);
    out.println("Connected to the router.");

    try {
        Thread.currentThread().sleep(10000);
    } catch(InterruptedException ie) {
        System.out.println("Thread interrupted");
    }

    for (int i=0; i<10; i++) {
        if (destination.equals((String) RTable[i][0])) {
            outSocket = (Socket) RTable[i][1];
            System.out.println("Found destination: " + destination);
            outTo = new PrintWriter(outSocket.getOutputStream(), true);
        }
    }
```

Code Snippet 13: **Correct Destination**

Simulation

Simulating the system running is crucial for gathering the performance metrics in the following section. It is important to note that when running the files, they are run in the following order: TCPServerRouter> TCPServer > TCPClient. This ensures that connections are made in the correct order and communication can run seamlessly. For the sake of showing the simulation output, all files have been run and completed their given jobs. Also, note that for privacy reasons all host computer's IPV-4 addresses and file paths have been covered by a green line.

The following simulation output in *Simulation Output 1* is the result of running a fully connected TCPServerRouter.

```
ServerRouter is Listening on port: 5555.
ServerRouter connected with Client/Server: 127.0.0.1
Forwarding to [redacted]
ServerRouter connected with Client/Server: [redacted]
Forwarding to 127.0.0.1
Found destination: [redacted]
Found destination: 127.0.0.1
```

Simulation Output 1: **TCPServerRouter**

The server router will display that it has made a connection and is listening on port 5555. From there it will make a connection with the local host and forward the data to the client holding the host computer's IPV-4 address. It will then repeat the process in the reverse order and if all is successful it will display that it has found both destinations and a connection is made. With this complete the server router has been established and connection for both the client and server can ensue.

The following output is from, *Simulation Output 2,* a connected TCPClient. After the client has randomly generated its matrices they will be sent to the server router and then to the server.

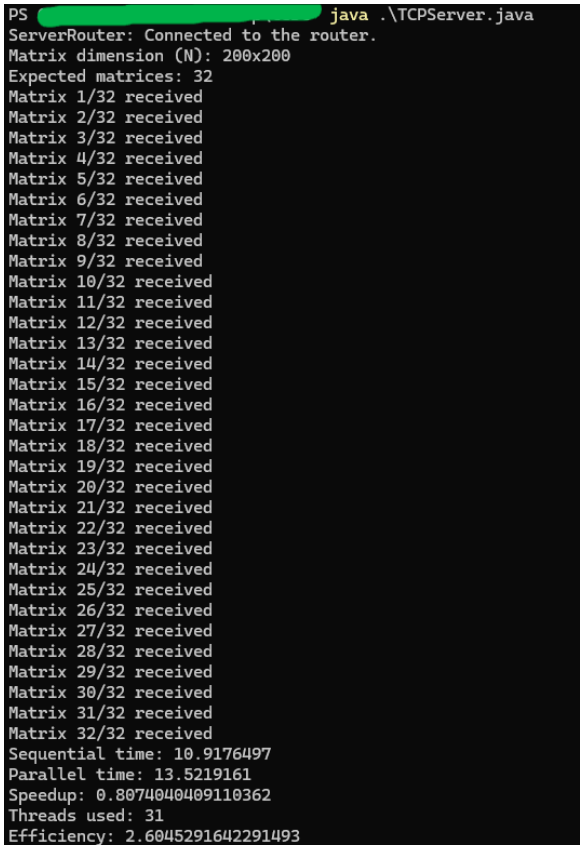
```
Sending matrix dimension (N): 10
Sending number of matrices: 4
Sending matrix string: 86,71,10,77,56,15,36,92,37,83;71,17,5,58,6,23,53,66,58,88;64,79,77,25,32,73,32,21,34,60;64,34,64,
0,99,16,39,28,78,13;50,16,0,9,70,67,94,78,62,74;44,37,42,99,61,68,97,54,69,53;56,95,72,14,30,82,20,42,43,76;70,73,81,13,
70,26,55,3,65,38;21,84,60,78,20,75,73,41,11,64;11,86,68,73,13,6,35,63,23,8
Server Response: Size received
Sending matrix string: 15,29,24,27,62,83,67,37,36,6;82,50,78,6,25,64,89,10,29,31;42,69,50,88,58,83,53,77,65,65;1,37,23,4
0,69,56,34,63,22,79;90,45,95,87,3,49,67,48,24,49;29,46,44,19,86,11,72,33,61,83;18,86,28,52,79,19,84,5,14,78;26,41,21,52,
79,59,68,28,71,39;11,68,97,75,48,4,78,81,60,92;61,14,92,18,15,16;85,33,68,84
Server Response: Count received
Sending matrix string: 0,10,56,63,14,76,3,40,51,0;45,9,11,85,60,85,75,51,16,12;36,62,13,24,3,96,94,15,4,37;85,45,92,8,68
,72,9,69,21,32;69,21,19,78,38,97,45,28,66,36;27,53,70,67,18,7,32,69,28,47;56,13,52,24,42,12,15,31,11,85;85,45,32,33,58,2
5,73,46,99,82,51,10,97,56,4,77,43,90,90,0;8,52,38,55,99,84,28,11,79,25
Server Response: Matrix 1 stored
Sending matrix string: 7,16,92,4,25,74,15,77,93,3;37,19,34,57,23,62,88,47,79,55;10,83,89,59,75,66,65,99,94,15;41,59,29,5
1,47,84,11,8,94,59;33,80,17,87,0,90,92,24,51,90;18,86,68,95,89,59,83,88,76,58;75,64,66,93,22,50,56,68,63,91;48,77,69,74,
39,31,59,66,74,47;83,68,99,34,24,63,42,3,73,17;57,94,45,52,16,95,54,56,41,28
Server Response: Matrix 2 stored
PS [redacted]
```

Simulation Output 2: **TCPClient**

The simulation will show the output for the dimensions of the matrix as well as the number of individual matrices that Strassen's will be performed on. The number of individual matrices is kept track of for ease of calculation with the number of threads that will be used in Strassen's algorithm. Our group has encountered a problem where it will only print the number of matrices - 2. In *Simulation Output 2* this can be seen where the matrix Strings are being sent it will say:
Sending matrix string: **SENT MATRIX**.
Server Response: Matrix 1 stored
Sending matrix string: **SENT MATRIX**.
Server Response: Matrix 2 stored

This is an output error that does not affect Strassen's algorithm or calculated performance metrics in any way. The server will still receive all four matrices and perform correct calculations on them.

The following output is from, *Simulation Output 3*, a connected TCPServer. Once the client has sent its matrices to the server router and the server has received them from the server router that completion will be output.



Simulation Output 3: **TCPServer**

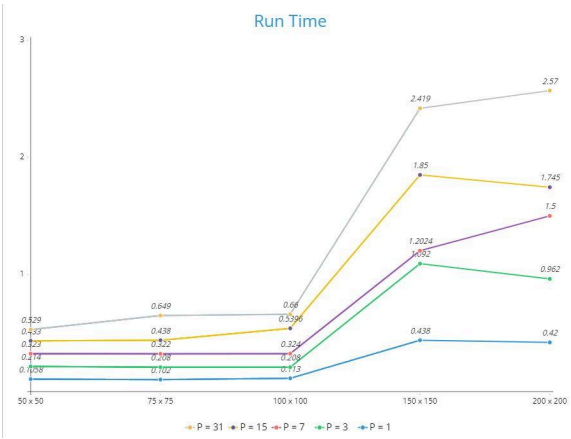
This simulation has been done with a 200 x 200 matrix and 32 threads, as seen above in *Simulation Output 3*, for better performance metrics to discuss. The server will output each of the thirty-two matrices and confirm that they have been received. Once the server has received all individual matrices performance metrics will begin to be calculated as seen in the bottom five lines of *Simulation Output 3*. While the matrices do not print the information that is contained in them it is important to note that the actual matrices are stored and further returned into a full matrix once Strassen's algorithm has finished.

Tabulated Data

The experimental data presented below provides a comprehensive analysis of the runtime, speedup, and efficiency when implementing Strassen's algorithm for matrix multiplication. Strassen's algorithm can handle matrix multiplication but can be slow when large numbers of matrices are calculated at once. However, this can be optimized through parallel and distributed processing with a binary tree of threads. The primary objective of this experiment was to evaluate the impact of increasing thread or core count (from 1 to 31 threads or cores) on the performance of the algorithm across various matrix sizes (50 x 50, 75 x 75, 100 x 100, 150 x 150, and 200 x 200).

Run Time Analysis

The run-time data in *Table 1* below reveals a consistent trend through the experimental data: as the number of threads increases, the time required to perform the matrix multiplication also increases, but only up to a certain point. View *Graph 1* below for a virtualization. Focusing on the smaller matrices is where the increase in run time is best seen.



Graph 1: **Run Time Graph**

This noticeable increase in run time can best be seen in the 50 x 50 matrix size. When there is only one thread for the matrices the approximate runtime is 0.1058 however if the threads are increased to seven for example the approximate runtime increases to 0.323. The data for the smaller matrices shows that parallelism is not as effective at the smaller scale as the overhead of managing the threads offers longer run times than just using Strassen's algorithm without a parallel implementation. Even when focusing on large matrix sizes there will still be an increase in run time when more threads are used. When looking at the 200 x 200 matrix size only using a single thread results in approximately 0.42 run time but when scaled up to seven total threads the run time changes to 0.1.50.

Run Times	Matrix Sizes				
Number of Threads (Cores)	50	75	100	150	200
P = 1	0.1058	0.102	0.113	0.438	0.42
P = 3	0.214	0.208	0.208	1.092	0.962
P = 7	0.323	0.322	0.324	1.2024	1.50
P = 15	0.433	0.438	0.5396	1.85	1.745
P = 31	0.529	0.649	0.66	2.419	2.57

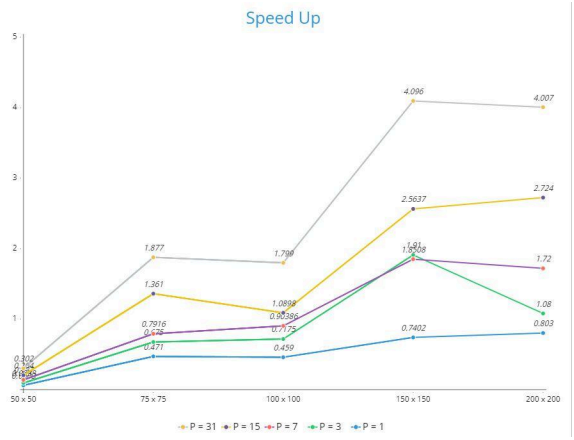
Table 1: **Approximate Run Time**

While these referenced run times may show that parallelism is not the fastest for calculating the matrix multiplication this is simply the approximated run time and not the time that parallelism saves when performing Strassen's algorithm on the implemented matrices.

Speed Up Analysis

Speedup (S) is a metric that measures the improvement in execution time achieved by using multiple threads or cores compared to a single-threaded execution. The speedup can be calculated by the following questions: $S = \frac{T1}{TP}$, where T1 is the run time of a single thread and TP is the run time of P number of threads. The data in *Table 2* starts to provide insight into the benefits of parallelism. The speedup data illustrates the effectiveness of parallelization, showing how much faster the

algorithm runs with multiple threads compared to a single thread. Speedup improves considerably when matrix sizes grow but leads to diminishing returns at a certain point. View *Graph 2* below for a virtualization. The results reveal that parallelism becomes increasingly beneficial as the matrix size grows.



Graph 2: **Speed Up Graph**

For smaller matrices, such as the 50 x 50, speedup remains relatively modest across all thread counts due to the overhead of thread management outweighing the benefits of parallel computation. For instance, at three threads the speedup is only 0.0938 while as the threads are increased to seven the speedup increases to 0.137 showing diminishing returns. This suggests that smaller matrices do not provide enough computational work to justify the overhead that the available threads provide.

In contrast, larger matrices begin to show a more significant speedup, demonstrating the scalability of the algorithm when a sufficient workload is present. For a larger matrix, such as the 200 x 200, the speedup at fifteen threads is approximately 2.5637, and for thirty-one threads it rises to 4.096. These improvements indicate that the benefits of parallel processing outweigh the thread management overhead for larger problem sizes. However, the data also highlights the phenomenon of diminishing returns. Doubling the thread count does not consistently double the speedup, as seen in the marginal increase between fifteen threads and thirty-one.

Speed Up	Matrix Sizes				
Number of Threads (Cores)	50	75	100	150	200
P = 1	0.0603	0.471	0.459	0.7402	0.803
P = 3	0.0938	0.675	0.7175	1.910	1.080
P = 7	0.137	.7916	0.90386	1.8508	1.720
P = 15	0.204	1.361	1.0898	2.5637	2.724
P = 31	0.302	1.877	1.799	4.096	4.007

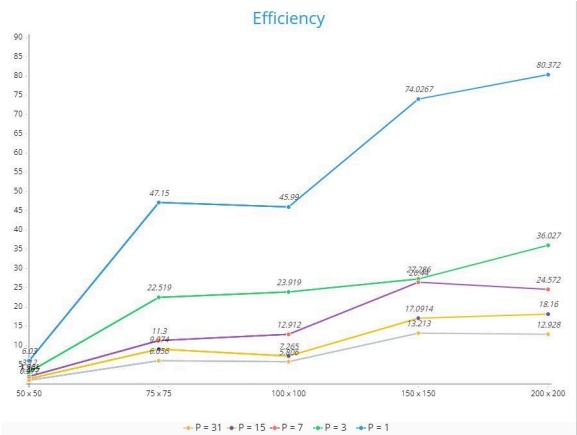
Table 2: **Approximate Speed Up**

Efficiency Analysis

Efficiency (E) measures how effectively the threads are being used when implemented and can be calculated by $E = \frac{S}{P} x\ 100$ where S is the speed up achieved and P is the number of threads used. The data in *Table 3* provides insight into how effectively each thread is being used at each matrix size. Similarly to speed up there are

diminishing returns once the matrices reach a certain size.

For smaller matrices, such as the 50 x 50 size, efficiency drops steeply due to the overhead of managing threads dominating the computation. For instance with three threads the efficiency is approximately 3.12%, and continues to decrease further to 0.977% when thirty-one threads are implemented. View *Graph 3* below for a virtualization. This indicates that the parallel implementation struggles to make optimal use of additional threads for small problem sizes, primarily due to insufficient workload to distribute across threads.



Graph 3: **Efficiency Graph**

This same trend continues when looking at the efficiency data for the larger matrices. At fifteen threads the efficiency for a 200 x 200 matrix is around 18.16%, and at thirty-one threads it drops to 12.928%.

Efficiency	Matrix Sizes				
Number of Threads (Cores)	50	75	100	150	200
P = 1	6.03	47.15	45.99	74.0267	80.372
P = 3	3.12	22.519	23.919	27.286	36.027
P = 7	1.961	11.30	12.912	26.44	24.572
P = 15	1.365	9.074	7.265	17.0914	18.16
P = 31	0.977	6.056	5.806	13.213	12.928

Table 3: **Approximate Efficiency**

Despite the decline in efficiency, the data from *Table 3* supports the conclusion that parallelism provides considerable benefits for large-scale matrix computations, even if the resources are not utilized with perfect efficiency.

Conclusion

This project demonstrates the implementation and analysis of Strassen's algorithm for matrix-chain multiplication within a TCP-based client-server architecture. By integrating a binary tree structure for parallel and distributed computing, the project explores the potential benefits of threading with performance metrics such as run time, speed up, and efficiency.

The approximate results reveal key insights into the scalability that parallelism offers to a developer. While smaller computational loads show limited benefits from the increased overhead of thread management, computationally heavy calculations exhibit significant improvements in all three performance categories. The run time analysis highlights that increasing the number of threads will initially reduce computation time for larger matrices, but diminishing returns become very apparent as the thread count increases. The speed-up and efficiency metrics further emphasize that a correct balance of threads to matrix size distribution parallelism offers a much-needed performance boost.

Despite the decline in efficiency with larger thread counts the results underscore the practical advantages of parallelism for computationally heavy tasks. The project validates the adaptability of Strassen's algorithm in a distributed environment and illustrates its effectiveness in leveraging modern multi-core architecture. In conclusion, the combination of a TCP-based client-server and parallel computation demonstrates a scalable and efficient approach for modern real-world problems.

References

[1] “Implementing Strassen’s Algorithm in Java,” *GeeksforGeeks*, Jan. 27, 2021. https://www.geeksforgeeks.org/implementing-strassens-algorithm-in-java/

[2] M. Liu. (2024). The Socket API [PowerPoint]. Available: https://kennesaw.view.usg.edu/d2l/le/content/3290732/viewContent/52300670/View

[3] M. Liu. (2024). The Client-Server Model – part 1 [PowerPoint]. Available: https://kennesaw.view.usg.edu/d2l/le/content/3290732/viewContent/52300671/View

[4] M. Liu. (2024). The Client-Server Model – part 2 [PowerPoint]. Available: https://kennesaw.view.usg.edu/d2l/le/content/3290732/viewContent/52300672/View