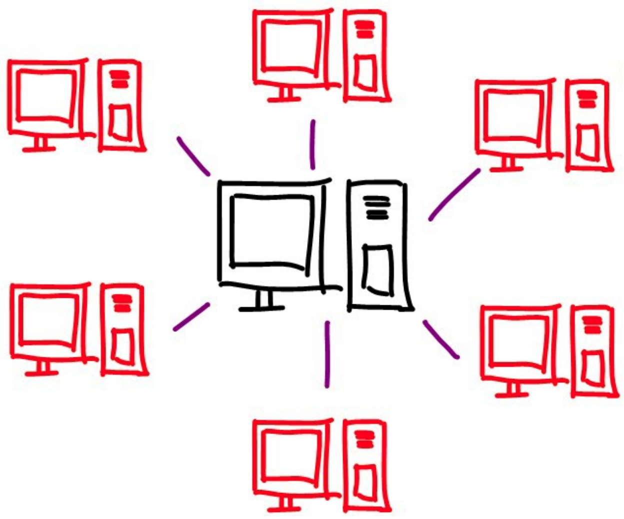


Distributed Matrix Multiplication System

Department: **Department of Computer Science**
Course: **Operating Systems (CSC301)**

Members

Name: Moiz Farooq	Email: f19csc01@shu.edu.pk
Name: Samad Farooq	Email: f19csc02@shu.edu.pk
Name: Rabia Shaikh	Email: f19csc33@shu.edu.pk

Lead by

Name: Dr. Anjum Nazir	Email: anjum.nazir@shu.edu.pk

SALIM HABIB UNIVERSITY
(FORMERLY BARRETT HODGSON UNIVERSITY)

Table of Content

Project Description

This project shows how to use the distributed system to perform some computations, in our case, we will use matrix multiplication.

Project Scope

This project includes multiple clients that can send two matrices to the Manager(Server) for computing the multiplication. The Manager is responsible for dividing the task and sending them to Workers (Servers), getting the computed result from Workers, and then merging all the task results and sending it to the client.

This project uses

- Distributed System Architecture
- Sockets for client/server communication
- Multithreading in Servers (Workers and Manager)
- Matrix Multiplication

Architecture Diagram

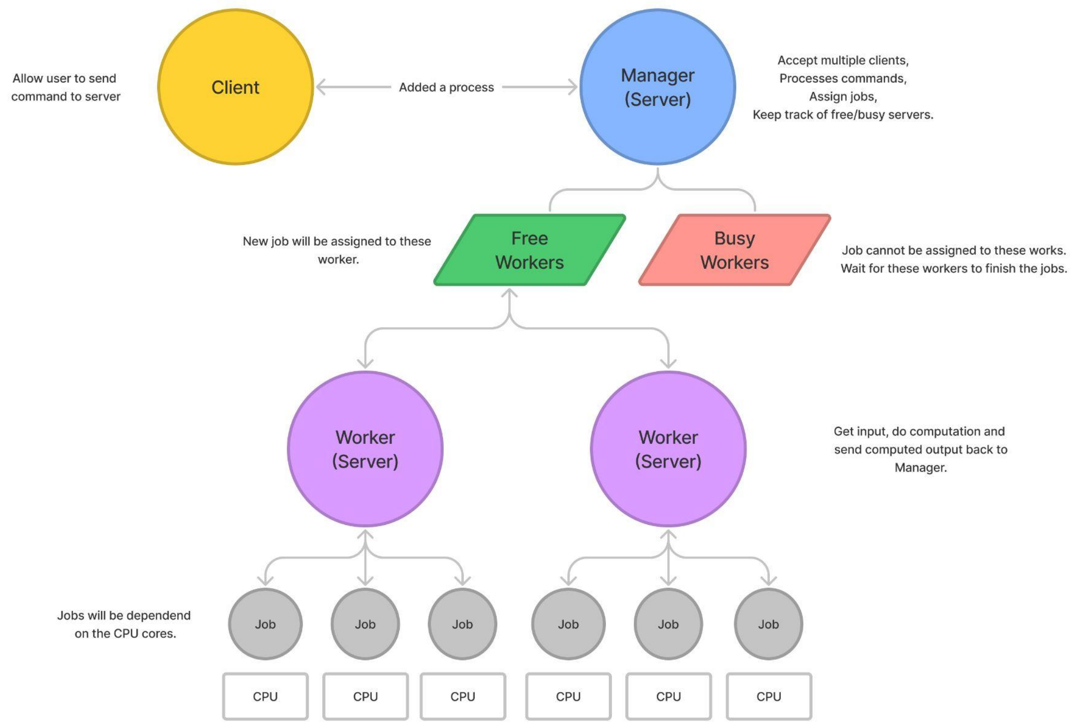
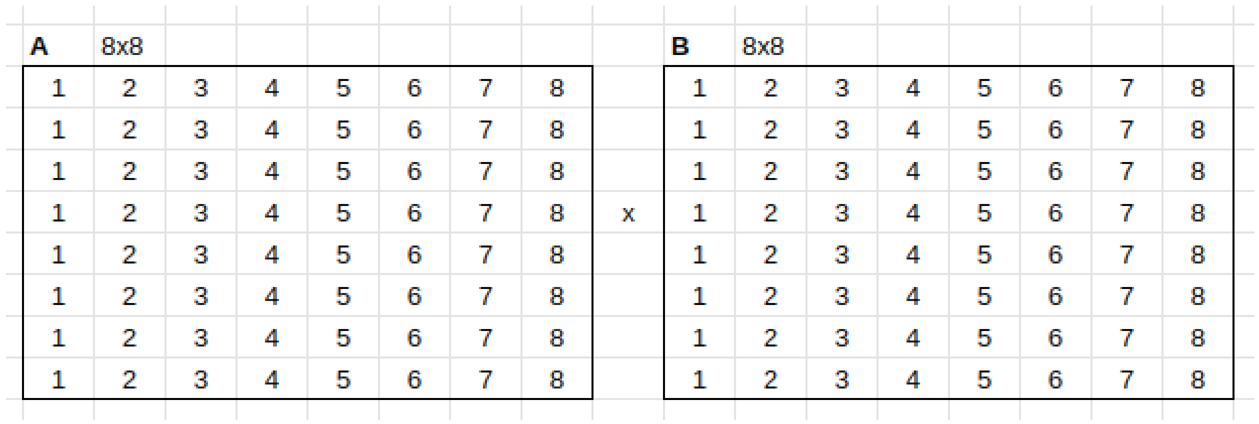

Allow user to send command to server
Client
Added a process
Manager (Server)
Accept multiple clients, Processes commands, Assign jobs, Keep track of free/busy servers.
New job will be assigned to these worker.
Free Workers
Busy Workers
Job cannot be assigned to these works. Wait for these workers to finish the jobs.
Worker (Server)
Worker (Server)
Get input, do computation and send computed output back to Manager.
Jobs will be dependend on the CPU cores.
Job
Job
Job
Job
Job
Job
CPU
CPU
CPU
CPU
CPU
CPU

Fig 1. Architecture diagram for distributed computing

Methodology

Let's take an 8x8 matrices



A	8x8						
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

x

B	8x8						
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8

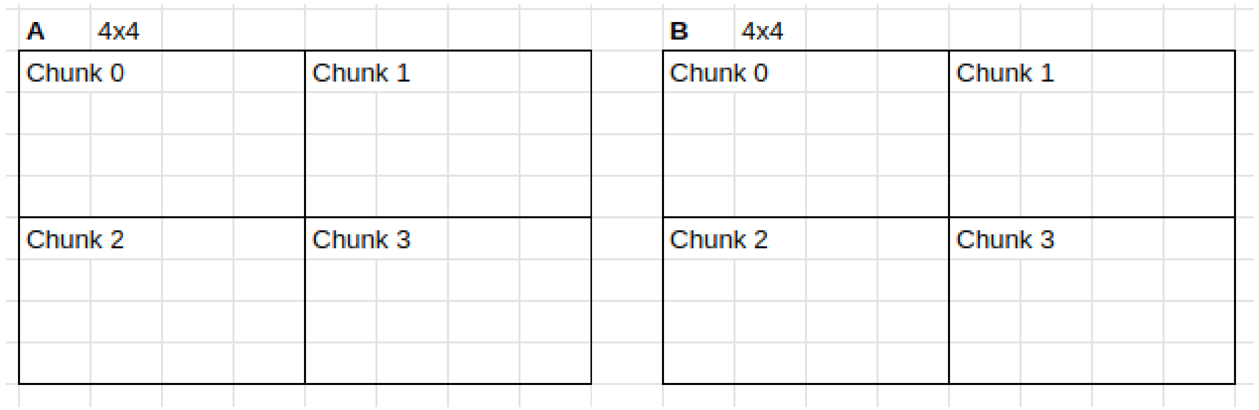
Fig 2.1. 8x8 matrics of A and B

Now we will perform the following steps:

1. We will send these two matrices to our Manager. If required, the manager will add zero paddings to these matrices.
2. Now partition these matrices into a size of 4.

Note:

- Matrices must be the power of 2.
- Partition size must be a power of 4.
- To calculate chunk size, we will use
  - chunk size = sqrt(dim(M) x dim(M) / partition size)

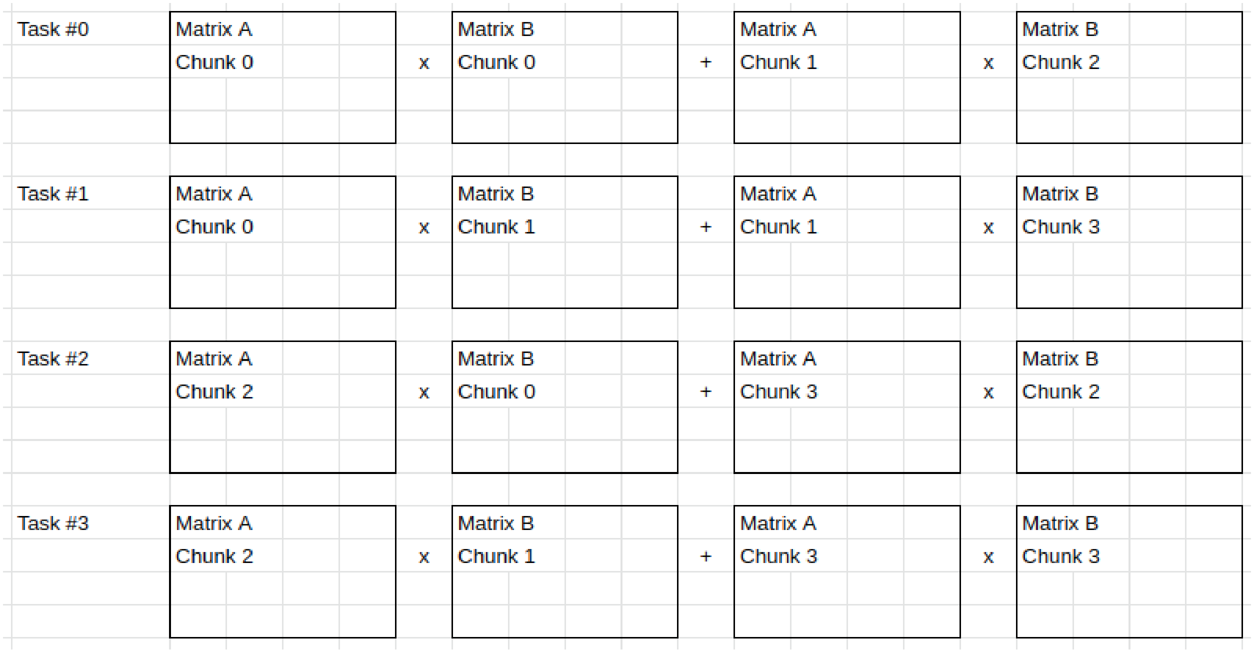


A	4x4
Chunk 0	Chunk 1
Chunk 2	Chunk 3

B	4x4
Chunk 0	Chunk 1
Chunk 2	Chunk 3

Fig 2.2 Divide A and B matrices into 4 chunks

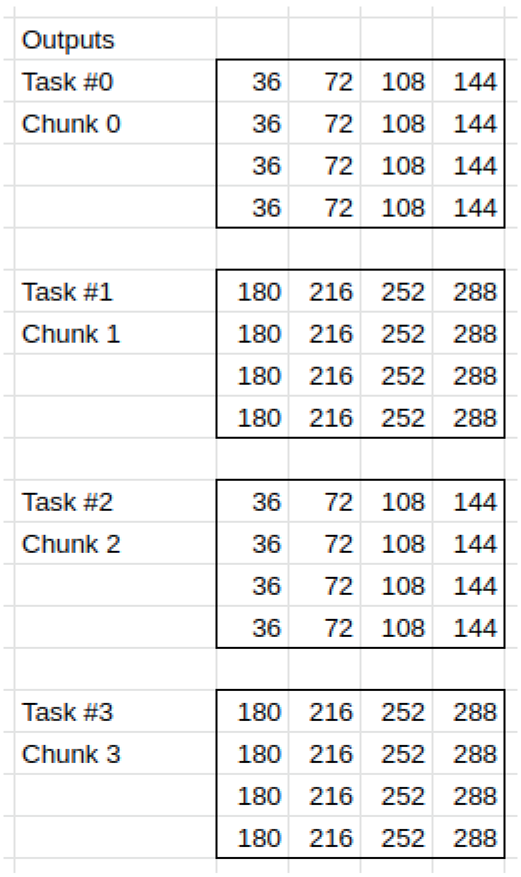
After dividing both matrixes into chunks, we will now create tasks for our workers.



Task #0	Matrix A Chunk 0	x	Matrix B Chunk 0	+	Matrix A Chunk 1	x	Matrix B Chunk 2
Task #1	Matrix A Chunk 0	x	Matrix B Chunk 1	+	Matrix A Chunk 1	x	Matrix B Chunk 3
Task #2	Matrix A Chunk 2	x	Matrix B Chunk 0	+	Matrix A Chunk 3	x	Matrix B Chunk 2
Task #3	Matrix A Chunk 2	x	Matrix B Chunk 1	+	Matrix A Chunk 3	x	Matrix B Chunk 3

Fig 2.3. Converting chunks into tasks for our workers

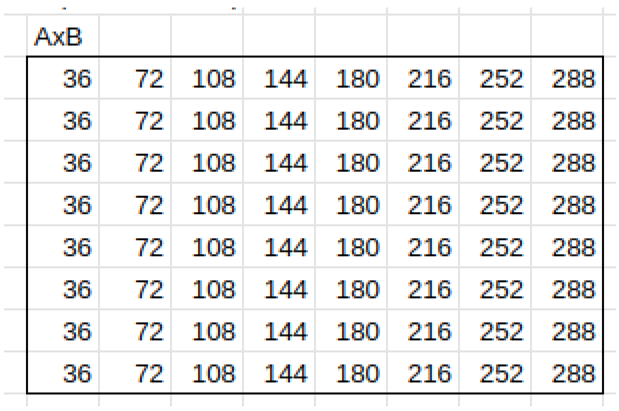
We will create a thread for each task depending on the free workers. Now we will have an output for each task from our workers and stored all the output in the result array.



Outputs				
Task #0	36	72	108	144
Chunk 0	36	72	108	144
	36	72	108	144
	36	72	108	144
Task #1	180	216	252	288
Chunk 1	180	216	252	288
	180	216	252	288
	180	216	252	288
Task #2	36	72	108	144
Chunk 2	36	72	108	144
	36	72	108	144
	36	72	108	144
Task #3	180	216	252	288
Chunk 3	180	216	252	288
	180	216	252	288
	180	216	252	288

Fig 2.4. Output from workers for our task

We will merge all the output in a single matrix and will remove zero paddings if needed. (to make it 2^n)



AxB							
36	72	108	144	180	216	252	288
36	72	108	144	180	216	252	288
36	72	108	144	180	216	252	288
36	72	108	144	180	216	252	288
36	72	108	144	180	216	252	288
36	72	108	144	180	216	252	288
36	72	108	144	180	216	252	288
36	72	108	144	180	216	252	288

FIg 2.5. Combine all results from our worker's

Performance

We will be running the code with randomly generated inputs for matrix dimensions of 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192.

We will work out two examples

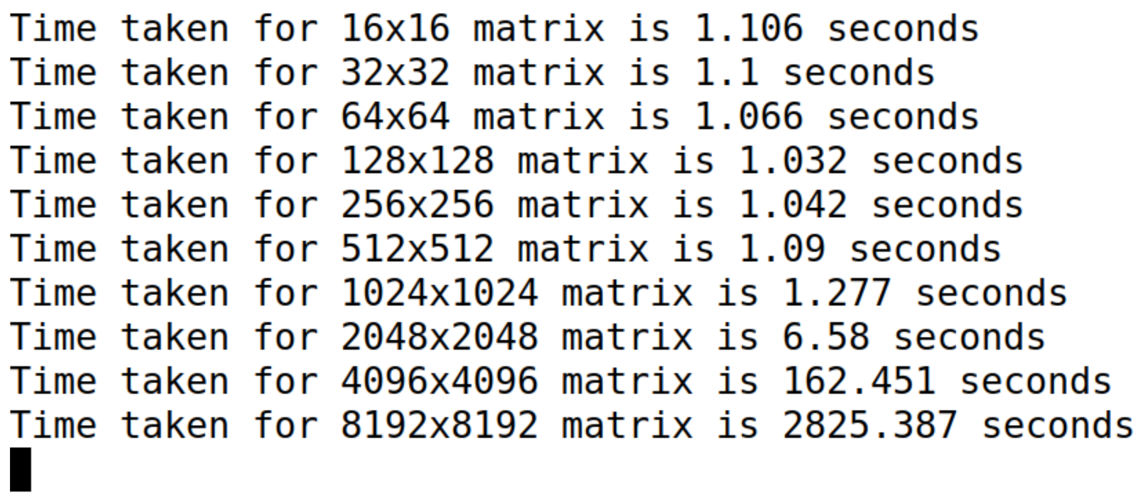
1. Single System
2. Distributed System (three separate workers)

System specs:

Processor	Intel® Core™ i7-9700 CPU @ 3.00GHz × 8
RAM	31.3 GiB
System type	64-bit Operating System, x64-based processor

1. Single System

Result:



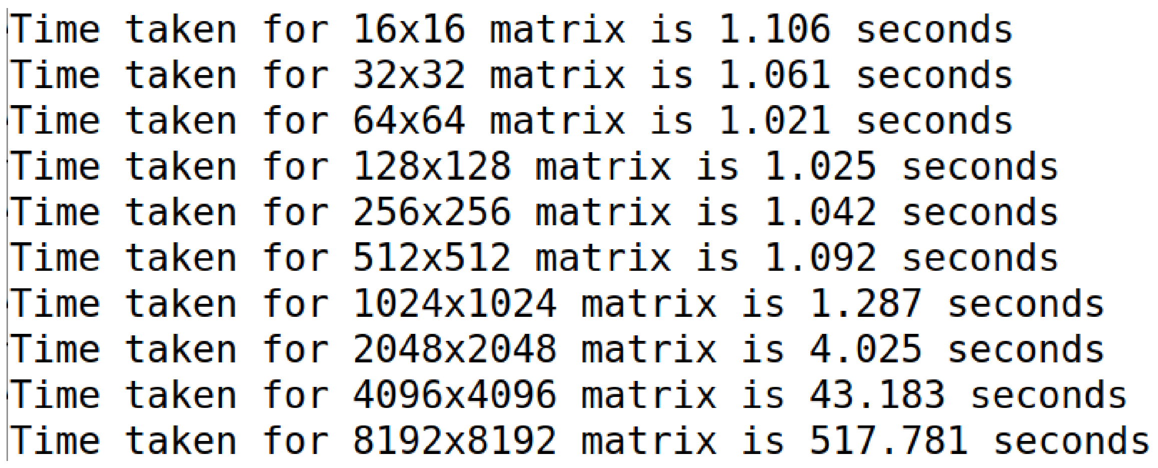
```
Time taken for 16x16 matrix is 1.106 seconds
Time taken for 32x32 matrix is 1.1 seconds
Time taken for 64x64 matrix is 1.066 seconds
Time taken for 128x128 matrix is 1.032 seconds
Time taken for 256x256 matrix is 1.042 seconds
Time taken for 512x512 matrix is 1.09 seconds
Time taken for 1024x1024 matrix is 1.277 seconds
Time taken for 2048x2048 matrix is 6.58 seconds
Time taken for 4096x4096 matrix is 162.451 seconds
Time taken for 8192x8192 matrix is 2825.387 seconds
```

FIg 3. Performance outputs when using single machine

2. Distributed System

Workers are running separate machines with the same system specs. We are using two workers for this test.

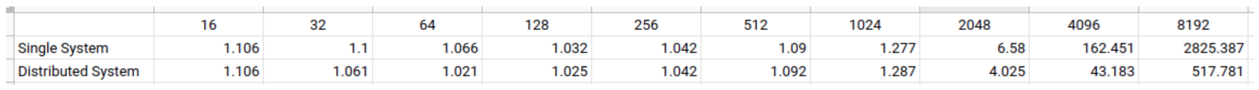
Result:



```
Time taken for 16x16 matrix is 1.106 seconds
Time taken for 32x32 matrix is 1.061 seconds
Time taken for 64x64 matrix is 1.021 seconds
Time taken for 128x128 matrix is 1.025 seconds
Time taken for 256x256 matrix is 1.042 seconds
Time taken for 512x512 matrix is 1.092 seconds
Time taken for 1024x1024 matrix is 1.287 seconds
Time taken for 2048x2048 matrix is 4.025 seconds
Time taken for 4096x4096 matrix is 43.183 seconds
Time taken for 8192x8192 matrix is 517.781 seconds
```

FIg 4. Performance outputs when using distributed systems

Final results



	16	32	64	128	256	512	1024	2048	4096	8192
Single System	1.106	1.1	1.066	1.032	1.042	1.09	1.277	6.58	162.451	2825.387
Distributed System	1.106	1.061	1.021	1.025	1.042	1.092	1.287	4.025	43.183	517.781

FIg 5. Comparing single vs distributed in tabular data.

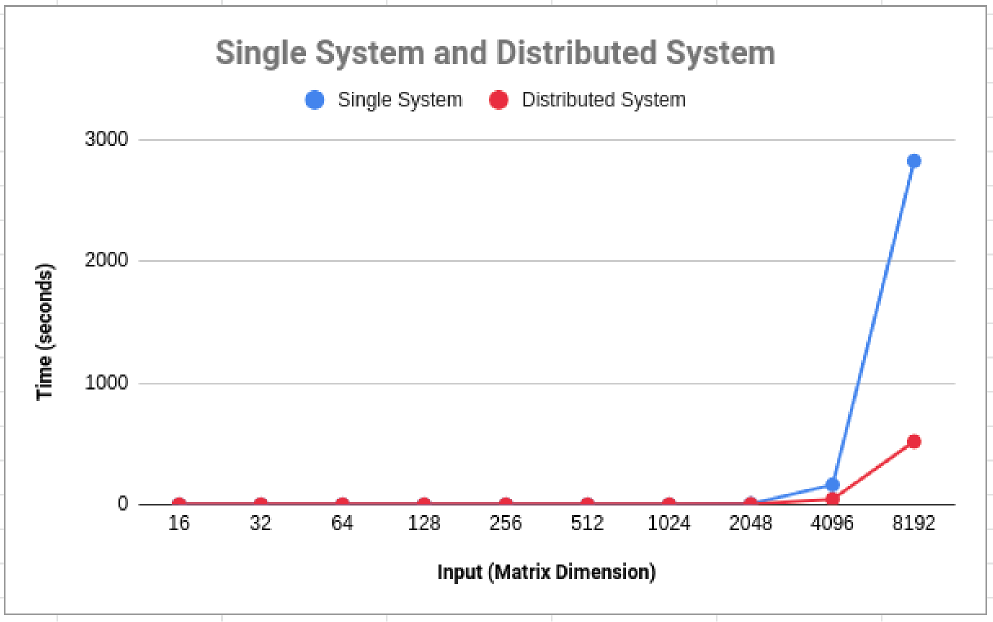


FIg 5. Single vs Distributed System comparison

How does it work?

We have three roles:

- Client - the client is the one who will send the data to the server.
- Manager - the manager is responsible for communicating with the client and workers.
- Worker - the worker is the one who will perform the computation.

The client will send the two matrices (AxB) to our Manager (server) and the Manager will divide that two matrices into chunks and distribute them to the workers. Then the workers will perform the multiplication and send the result back to the Manager. The Manager will merge the result and send it back to the Client.

How to install the project?

To install our project by cloning the repository.

```
$ git clone https://github.com/progrmoiz/distributed-matrix-multiplication
```

How to run the project?

First of all, let's compile and run the Worker:

```
$ javac Worker.java
$ java Worker 9001
```

You can run as many workers as you like and workers can be run on the same network or on a different network.

-- In the case of different networks, you can also use Ngrok to expose the port.

Then, we have to first define our worker's IP address and port in the Manager. You can check the Manager's `main` function code to see how to do that. Then, we can compile and run the Manager:

```
$ javac Manager.java
$ java Manager
```

-- Manager is by default listening on port 6666.

Then, we can run the Client. You can change the input matrix in the Client's `main` function. So, let's now compile and run the Client:

```
$ javac Client.java
$ java Client
```

Hurray! You can see the result in the console.

References - Distributed Matrix Multiplication

Distributed Systems

- https://github.com/remzi-arpacidusseau/ostep-projects/tree/master/filesystems-distributed
- https://pages.cs.wisc.edu/~remzi/OSTEP/dist-intro.pdf
- https://github.com/Huiming37/Distributed-Computing-Socked-Programming
- https://github.com/remzi-arpacidusseau/ostep-code/tree/master/dist-intro
- Shared Libraries
  - https://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html

Job Managers

- https://www.mymiller.name/wordpress/programming/java/java-build-your-own-jobmanager/
- https://www.ibm.com/docs/en/imdm/10.1?topic=SSWSR9_10.1.0/com.ibm.pim.java.doc/com/ibm/pim/job/JobManager.html

Socket Programming

- https://www.javatpoint.com/socket-programming
- https://www.baeldung.com/a-guide-to-java-sockets

Matrix Multiplication

- https://introcs.cs.princeton.edu/java/95linear/Matrix.java.html
- https://en.wikipedia.org/wiki/Strassen_algorithm
- https://github.com/jkoelmel/multithreaded-matrix-multiplication/blob/main/RowMultiply.java
- https://github.com/akarshbolar/Parallel_Computing/blob/master/StrassenThread.java
- https://akarshbolar.wordpress.com/2017/04/02/matrix-multiplication-in-java/
- Strassen Multiplication
  - https://gist.github.com/bumblePrime/8496acce44b13de366131020c5d01440
  - https://gist.github.com/OlinAlvarez/670dac77ecfb4f2e7942def2bd6682b3

Multi-threading

- https://docs.oracle.com/javase/tutorial/essential/concurrency/runthread.html
- https://www.geeksforgeeks.org/process-based-and-thread-based-multitasking/
- https://www.bestprog.net/en/2021/01/13/java-multitasking-threads-of-execution-threads-basic-concepts/

Code

```java
// MainClient.java
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.logging.Logger;

/**
 * It connects to the manager and sends the data to the manager and receives the result from the
manager.
 */
public class MainClient {
  private Socket clientSocket;
  private ObjectOutputStream outputStream;
  private ObjectInputStream inputStream;

  // Logger for this class
  private static final Logger LOGGER = Logger.getLogger(Manager.class.getName());

  /**
   * Create a socket connection to the server
   *
   * @param ip   The IP address of the server.
   * @param port The port to connect to.
   */
  public void startConnection(String ip, int port) throws IOException {
    clientSocket = new Socket(ip, port);
    LOGGER.info("Connected to " + clientSocket.getInetAddress() + ":" + clientSocket.getPort());

    outputStream = new ObjectOutputStream(clientSocket.getOutputStream());
    inputStream = new ObjectInputStream(clientSocket.getInputStream());
  }

  /**
   * It reads the data from the input stream and returns it as a matrix
   *
   * @return Nothing is being returned.
   */
  public Matrix receiveData() {
    try {
      return (Matrix) inputStream.readObject();
    } catch (IOException e) {
      System.out.println("Receiving data failed");
      e.printStackTrace();
    } catch (ClassNotFoundException e) {
      System.out.println("Receiving data failed");
      e.printStackTrace();
    } catch (Exception e) {
      System.out.println(e);
      e.printStackTrace();
    }
    return null;
  }

  /**
   * Send the data to the server
   *
   * @param data The data to send.
   */
  public void sendData(Matrix[] data) {
    try {
      outputStream.writeObject(data);
      outputStream.flush();
    } catch (IOException e) {
      System.out.println("Sending data failed");
      e.printStackTrace();
    } catch (Exception e) {
      System.out.println(e);
      e.printStackTrace();
    }
  }

  /**
   * It closes the connection.
   */
  public void stopConnection() {
    try {
      inputStream.close();
      outputStream.close();
      clientSocket.close();
    } catch (IOException e) {
      System.out.println("Closing connection failed");
      e.printStackTrace();
    } catch (Exception e) {
      System.out.println(e);
      e.printStackTrace();
    }
  }

  public static void main(String[] args) {
    Matrix matrixA =  new Matrix(8, 8);

    Matrix matrixB = new Matrix(8, 8);

    // create 8x8 matrix
    for (int i = 0; i < 8; i++) {
      for (int j = 0; j < 8; j++) {
        matrixA.set(i, j, j + 1);
        matrixB.set(i, j, j + 1);
      }
    }
    Matrix[] matrices = { matrixA, matrixB };

    try {
      MainClient mainClient = new MainClient();
      mainClient.startConnection("localhost", 6666);
      LOGGER.info("Sending following matrices to manager...");
      // matrixA.show("A");
      // matrixB.show("B");
      mainClient.sendData(matrices);

      // receive data
      Matrix receivedData = mainClient.receiveData();
      LOGGER.info("Received final output from manager.");
      receivedData.show("A x B");
      mainClient.stopConnection();
    } catch (IOException e) {
      LOGGER.severe("Connection failed");
      LOGGER.severe("Exiting...");
    }
  }
}


// Manager.java
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.logging.Logger;

/**
 * The Manager class is responsible for managing the workers and the client. It receives the matrices
 * from the client, divides the matrices into chunks, and sends the chunks to the workers. It also
 * receives the results from the workers and merges them into a single matrix.
 */
public class Manager {
  // Server socket for accepting client connections
  private ServerSocket serverSocket;

  // Logger for this class
  private static final Logger LOGGER = Logger.getLogger(Manager.class.getName());

  // List of InetSocketAddress
  // Host address is string and port number is integer
  // Example: "localhost", 1234
  private InetSocketAddress[] workerAddresses = {};

  // Key value pair of busy INetSocketAddress and busy boolean
  // Example: {new InetSocketAddress("localhost", 1234), false}
  private static Map<String, Boolean> workerStatus = new HashMap<>();

  private int partitionSize;

  /**
   * Add a worker to the list of workers
   *
   * @param workerAddress The address of the worker.
   */
  public void addWorker(InetSocketAddress workerAddress) {
    workerAddresses = Arrays.copyOf(workerAddresses, workerAddresses.length + 1);
    workerAddresses[workerAddresses.length - 1] = workerAddress;
  }

  // The constructor takes a partition size and throws an exception if it's not a
  // power of 4.
  public Manager(int partitionSize) {
    // partitionSize must be a power of 4
    if (partitionSize < 4 || (partitionSize & (partitionSize - 1)) != 0) {
      throw new IllegalArgumentException("partitionSize must be a power of 4");
    }

    this.partitionSize = partitionSize;
  }

  public Manager() {
    this.partitionSize = 4;
  }

  /**
   * Check if all workers are connected to the master
   *
   * @return The method returns a boolean value.
   */
  public boolean checkWorkersConnection() {
    boolean isConnected = true;

    // Check if all workers are open set to true
    for (InetSocketAddress workerAddress : workerAddresses) {
      Socket socket = new Socket();
      try {
        socket.connect(workerAddress);
        socket.close();
      } catch (IOException e) {
        isConnected = false;
      }
    }

    return isConnected;
  }

  /**
   * Start a server socket and wait for a connection.
   *
   * @param port the port number to listen on
   */
  public void start(int port) {
    try {
      // Check if workerAddresses is empty, terminate immediately
      if (workerAddresses.length == 0) {
        LOGGER.info("No worker available");
        return;
      }

      for (InetSocketAddress workerAddress : workerAddresses) {
        workerStatus.put(Helper.inetSocketAddressToString(workerAddress), false);
      }

      // Create a server socket
      serverSocket = new ServerSocket(port);
      // Print the IP address and port number
      LOGGER.info("Server started on " + serverSocket.getInetAddress() + ":" +
serverSocket.getLocalPort());

      // Print listening message
      LOGGER.info("Listening for connections...");

      // Wait for a client to connect
      while (true) {
        // Accept client connection and create a new thread for it
        new ManagerClientHandler(serverSocket.accept()).start();
      }

    } catch (IOException e) {
      LOGGER.severe("Connection failed");
      e.printStackTrace();
    }
  }

  /**
   * It closes the server socket.
   */
  public void stop() {
    try {
      serverSocket.close();
    } catch (IOException e) {
      LOGGER.info("Closing connection failed");
      e.printStackTrace();
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  /**
   * This is our manager job handler.
   * 1. Send a input to the worker
   * 2. Let the worker handle the job
   * 3. Receive a output from the worker
   *
   * @param Socket     clientSocket - The socket to the worker
   * @param Matrix[][] chunk - The data to send to the worker
   *
   * @return Matrix - The data received from the worker
   */
  public static Matrix job(Socket clientSocket, Matrix[][] chunk) throws IOException,
ClassNotFoundException {
    // Write the chunk to the socket
    ObjectOutputStream oos2 = new ObjectOutputStream(clientSocket.getOutputStream());
    oos2.writeObject(chunk);
    oos2.flush();

    // Read the result from the socket
    ObjectInputStream ois2 = new ObjectInputStream(clientSocket.getInputStream());
    Matrix result = (Matrix) ois2.readObject();

    // Close the socket
    ois2.close();
    oos2.close();
    // clientSocket.close();

    return result;
  }

  // Get all free servers
```

```java
  private InetSocketAddress[] getFreeWorkers() {
    // Return empty array if no servers are free
    if (workerStatus.isEmpty()) {
      return new InetSocketAddress[0];
    }

    return Arrays.stream(workerAddresses)
        .filter((InetSocketAddress workerAddress) ->
!workerStatus.get(Helper.inetSocketAddressToString(workerAddress)))
        .toArray(InetSocketAddress[]::new);
  }

  /**
   * Given a set of chunks of A and B, and the dimension of the matrix,
   * this function will return a set of chunks of A and B that will be fed to each
   * worker
   * Given a set of matrices, this function will divide them into chunks and
   * assign each chunk to a
   * worker
   *
   * @param aChunks           an array of matrices that represent the chunks of
   *                          the first matrix
   * @param bChunks           the chunks of the B matrix
   * @param dimensionOfMatrix the dimension of the matrix.
   * @return The resultMatrices is a 3D array of matrices. The first dimension is
   *         the worker number.
   *         The second dimension is the matrix that the worker is working on. The
   *         third dimension is the chunk
   *         of the matrix that the worker is working on.
   */
  public static Matrix[][][] arrangeTasks(Matrix[] aChunks, Matrix[] bChunks, int dimensionOfMatrix) {
    int chunkSize = aChunks[0].getM(); // keep in mind that this is dimension of chunk
    int elementsInChunks = (int) Math.pow(chunkSize, 2);
    int numWorkers = (int) Math.pow(dimensionOfMatrix, 2) / elementsInChunks; // 8*8 = num elements in
each matrix
    int gameChanger = (int) Math.sqrt(numWorkers);

    Matrix[][][] resultMatrices = new Matrix[numWorkers][2][gameChanger];

    for (int i = 0; i < numWorkers; i++) {
      int startA = (i / gameChanger) * gameChanger;
      int endA = startA + gameChanger;

      Matrix[] aChunksToWorker = new Matrix[gameChanger];

      int aChunkCounter = 0;
      for (int x = startA; x < endA; x++) {
        aChunksToWorker[aChunkCounter++] = aChunks[x];
      }
      int startB = (i % gameChanger);
      int numElementsInB = 0;

      Matrix[] bChunksToWorker = new Matrix[gameChanger];

      int bChunkCounter = 0;
      for (int x = startB;; x += gameChanger) {
        bChunksToWorker[bChunkCounter++] = bChunks[x];

        numElementsInB++;
        if (numElementsInB == gameChanger) {
          break;
        }
      }
      Matrix[][] temp = { aChunksToWorker, bChunksToWorker };
      resultMatrices[i] = temp;
    }

    return resultMatrices;
  }

 /**
  * We divide the matrices into chunks, send the chunks to the servers, and merge the results from the
  * servers
  */
  private class ManagerClientHandler extends Thread {
    private Socket clientSocket;
    private ObjectOutputStream outputStream;
    private ObjectInputStream inputStream;

    // Creating a new ManagerClientHandler object and passing the clientSocket to it.
    public ManagerClientHandler(Socket clientSocket) {
      this.clientSocket = clientSocket;
    }

    // We divide the matrices into chunks, send the chunks to the servers, and merge
    // the results from the servers.
    public void run() {
      try {
        outputStream = new ObjectOutputStream(clientSocket.getOutputStream());
        inputStream = new ObjectInputStream(clientSocket.getInputStream());

        Matrix[] data = (Matrix[]) inputStream.readObject();

        Matrix tempMatrixA = data[0];
        Matrix matrixA = Matrix.padding(tempMatrixA);

        Matrix tempMatrixB = data[1];
        Matrix matrixB = Matrix.padding(tempMatrixB);

        LOGGER.info("Received matrices from client: ");
        matrixA.show("A");
        matrixB.show("B");

        // Divide the integers array into chunks of size n
        int chunkSize = (int) Math.sqrt(Math.pow(matrixA.getM(), 2) / partitionSize);

        Matrix[] matrixAChunks = matrixA.divide(chunkSize);
        Matrix[] matrixBChunks = matrixB.divide(chunkSize);

        Matrix[][][] chunks = arrangeTasks(matrixAChunks, matrixBChunks, matrixA.getM());

        // Create a copy of the chunks
        Matrix[] resultChunks = new Matrix[chunks.length];

        // We will keep track of our threads so later we can pause execution
        // to wait for all threads to finish
        List<Thread> threads = new ArrayList<>();
        // send the chunks to servers clients in threads
        try {
          int chunkIndex = 0;

          while (true) { // 4 chunks
            InetSocketAddress[] workerAddresses = getFreeWorkers();

            // log all free servers
            LOGGER.info("Free servers: " + Arrays.toString(workerAddresses));

            if (workerAddresses.length == 0) {
              LOGGER.info("No free servers at the moment. Waiting for free servers...");
              Thread.sleep(1000);
              continue;
            }

            // calculate the remaining chunks
            int remainingChunks = chunks.length - chunkIndex;

            // If remaining chunks is less than the number of free servers,
            // assign the remaining chunks to the free servers
            if (workerAddresses.length > remainingChunks) {
              workerAddresses = Arrays.copyOfRange(workerAddresses, 0, remainingChunks);
            }

            // Iterate over the free servers
            for (InetSocketAddress workerAddress : workerAddresses) { // 1st server
              // Fix: Local variable chunkIndex defined in an enclosing scope must be final or
              // effectively final
              final int chunkIndexFinal = chunkIndex;

              // Set the server status to busy
              workerStatus.put(Helper.inetSocketAddressToString(workerAddress), true);

              Thread thread = new Thread(() -> {
                try {
                  // Log chunk index
                  LOGGER.info("Sending chunk " + chunkIndexFinal + " to worker "
                      + Helper.inetSocketAddressToString(workerAddress));

                  // Create a new socket
                  Socket workerClientSocket = new Socket(workerAddress.getHostName(),
workerAddress.getPort());
                  LOGGER.info("Connected to " + Helper.inetSocketAddressToString(workerAddress));

                  // Send the chunk to the server
                  Matrix result = job(workerClientSocket, chunks[chunkIndexFinal]);
                  LOGGER.info("Received result from worker " +
Helper.inetSocketAddressToString(workerAddress));

                  // Merge the result chunks
                  resultChunks[chunkIndexFinal] = result;

                  // Close the socket
                  workerClientSocket.close();

                  LOGGER.info("Closed connection to " +
Helper.inetSocketAddressToString(workerAddress));
```

```
            // Free the server
            LOGGER.info("Freeing worker: " + Helper.inetSocketAddressToString(workerAddress));
            workerStatus.put(Helper.inetSocketAddressToString(workerAddress), false);

          } catch (IOException e) {
            e.printStackTrace();
          } catch (ClassNotFoundException e) {
            e.printStackTrace();
          }
        });
        threads.add(thread);
        thread.start();

        chunkIndex++;

      }

      if (chunkIndex == chunks.length) {
        break;
      }

    }

  } catch (Exception e) {
    e.printStackTrace();
  }

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

  // Merge the results from the workers
  LOGGER.info("Merging results...");
  Matrix tempMerged = new Matrix(matrixA.getM(), matrixA.getN());
  tempMerged.joinAll(resultChunks);

  // If filled with zeros, remove extra zeros from the output
  Matrix merged = Matrix.cut(tempMerged, tempMatrixA.getM(), tempMatrixB.getN());
  merged.show();

  // Send the merged result to the client
  outputStream.writeObject(merged);
  outputStream.flush();

  inputStream.close();
  outputStream.close();
  clientSocket.close();

} catch (IOException e) {
  LOGGER.severe("Error while handling client: " + e.getMessage());
  e.printStackTrace();
} catch (ClassNotFoundException e) {
  e.printStackTrace();
}
```

```
    }
  }

  public static void main(String[] args) {
    Manager manager = new Manager((int) Math.pow(4, 1));
    manager.addWorker(new InetSocketAddress("localhost", 9001));
    manager.addWorker(new InetSocketAddress("localhost", 9002));

    manager.start(6666);
  }
}


// Worker.java
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;
import java.util.logging.Logger;

/**
 * It creates a server socket and waits for a connection. It accepts a client connection and creates a
 * new thread for it. The thread will handle the client separately. It is a multi client because this
 * accept multiple request from multiple client
 */
public class Worker {
  // Server socket for accepting client connections
  private ServerSocket serverSocket;

  // Logger for this class
  private static final Logger LOGGER = Logger.getLogger(Worker.class.getName());

  /**
   * Start a server socket and wait for a connection.
   *
   * @param port the port number to listen on
   */
  /**
   * Accept a client connection and create a new thread for it
   *
   * @param port The port number to which the server socket is bound.
   */
  public void start(int port) {
    try {
      // Create a server socket
      serverSocket = new ServerSocket(port);
      // Print the IP address and port number
      LOGGER.info("Server started on " + serverSocket.getInetAddress() + ":" +
serverSocket.getLocalPort());

      // Print listening message
      LOGGER.info("Listening for connections...");

      // Wait for a client to connect
      while (true) {
        // Accept client connection and create a new thread for it, This thread will
        // handle the client separately
        // It is a multi client because this accept multiple request from multiple
        // client
        new WorkerClientHandler(serverSocket.accept()).start();
      }

    } catch (IOException e) {
      LOGGER.info("Connection failed");
      e.printStackTrace();
    }
  }

  /**
   * It creates a server socket and listens for incoming connections
   */
  public void stop() {
    try {
      serverSocket.close();
    } catch (IOException e) {
      LOGGER.info("Closing connection failed");
      e.printStackTrace();
    } catch (Exception e) {
      LOGGER.info(e.getMessage());
      e.printStackTrace();
    }
  }

  private static class WorkerClientHandler extends Thread {
    private Socket clientSocket;
    private ObjectOutputStream outputStream;
    private ObjectInputStream inputStream;

    public WorkerClientHandler(Socket clientSocket) {
      this.clientSocket = clientSocket;
    }

    public static class RowMultiply implements Runnable {
      private Matrix mat1;
      private Matrix mat2;
      private Matrix result;
      private int row;

     // Initializing the variables.
      public RowMultiply(Matrix result, Matrix mat1, Matrix mat2, int row) {
        this.result = result;
        this.mat1 = mat1;
        this.mat2 = mat2;
        this.row = row;
      }

      // For each row in mat1, multiply that row by mat2 and store the result in
      // result.
      @Override
      public void run() {
        for (int i = 0; i < mat2.getN(); i++) {
          // result[row][i] = 0;
          result.set(row, i, 0);
          for (int j = 0; j < mat1.getN(); j++) {
            // result[row][i] += mat1[row][j] * mat2[j][i];
            result.set(row, i, result.get(row, i) + mat1.get(row, j) * mat2.get(j, i));
          }
        }
      }
    }

    public static class DotProduct implements Runnable {
      private Matrix mat1;
      private Matrix mat2;
      private Matrix finalResult;
      private int threshold;
      private int bigThreshold;

      // The constructor takes in the final result matrix, the two matrices that will
      // be multiplied, and a threshold value.
      public DotProduct(Matrix finalResult, Matrix mat1, Matrix mat2) {
        this.finalResult = finalResult;
        this.mat1 = mat1;
        this.mat2 = mat2;
        this.threshold = 100;
        this.bigThreshold = 2000;

      }

      // The `RowMultiply` class is a thread that multiplies a row of `mat1` by a row
      // of `mat2` and adds it to `finalResult`.
      // The `run` method of the `RowMultiply` class creates a new thread and starts
      // it.
      // The `run` method also adds the thread to a list of threads.
      // The `run` method checks if the number of threads is greater than 50.
      // If it is, it waits for the threads to finish.
      @Override
      public void run() {
        List<Thread> threads = new ArrayList<>();
        Matrix temp = new Matrix(mat1.getM(), mat1.getN());

        int rows = mat1.getM();

        for (int i = 0; i < rows; i++) {
          RowMultiply task = new RowMultiply(temp, mat1, mat2, i);
          Thread thread = new Thread(task);
          thread.start();
          threads.add(thread);
          if (threads.size() % 50 == 0) {
            Helper.waitForThreads(threads);
          }
        }
        Helper.waitForThreads(threads);
        finalResult.plusInPlace(temp);
      }
    }

    public static class ThreadCreation {

      /**
       * Given two matrices, create a thread for each matrix and run the dot product
       * on each thread
       *
       * @param mat1        The first matrix to be multiplied.
```

```
         * @param mat2        The matrix that is being multiplied by mat1.
         * @param finalResult the result matrix
         */
        public static void multiply(Matrix[] mat1, Matrix[] mat2, Matrix finalResult) {
          List<Thread> threads = new ArrayList<>();

          int numChunks = mat1.length;

          for (int i = 0; i < numChunks; i++) {
            DotProduct task = new DotProduct(finalResult, mat1[i], mat2[i]);
            Thread thread = new Thread(task);
            thread.start();
            threads.add(thread);
            if (threads.size() % 50 == 0) {
              Helper.waitForThreads(threads);
            }
          }
          Helper.waitForThreads(threads);
        }
      }

      public void run() {
        try {
          outputStream = new ObjectOutputStream(clientSocket.getOutputStream());
          inputStream = new ObjectInputStream(clientSocket.getInputStream());

          // Read the input stream into a 2D array of matrices.
          Matrix[][] data = (Matrix[][]) inputStream.readObject();
          Matrix[] matrixAChunks = data[0];
          Matrix[] matrixBChunks = data[1];
          LOGGER.info("Received data from " + clientSocket.getInetAddress() + ":" +
clientSocket.getPort());

          LOGGER.info("Starting computation...");

          Matrix result = new Matrix(matrixAChunks[0].getM(), matrixAChunks[0].getN());
          int rowsInChunk = matrixAChunks[0].getM();

          if (rowsInChunk < 2) {
            LOGGER.info("Calling matrix multiplication without threads. Give a bigger challenge to use
threads. :p");
            // Doing matrix multiplication.
            result = Matrix.dot(matrixAChunks, matrixBChunks);
          } else {
            LOGGER.info("Invoking threaded multiplication...");
            // The code is creating threads to multiply matrices.
            ThreadCreation.multiply(matrixAChunks, matrixBChunks, result);
          }

          result.show("Computed result");
          outputStream.writeObject(result);
          outputStream.flush();

          inputStream.close();
          outputStream.close();
          clientSocket.close();

        } catch (IOException e) {
          LOGGER.info("Connection failed" + e.getMessage());
          e.printStackTrace();
        } catch (ClassNotFoundException e) {
          e.printStackTrace();
        }
      }
    }

    public static void main(String[] args) {
      // Accept the port number from the command line
      int port = Integer.parseInt(args[0]);

      // Print help message if no port number is given
      if (port == 0) {
        LOGGER.info("Usage: java Worker <port>");
        System.exit(0);
      }

      // Start the server
      Worker worker = new Worker();
      worker.start(port);
    }
  }


  // Helper.java
  import java.net.InetSocketAddress;
  import java.awt.Point;
  import java.util.List;

  public class Helper {
    /**
     * Convert InetSocketAddress to String. We implement this function because
     * InetSocketAddress.toString() is not giving the unifiying format every time.
     *
     * @param inetSocketAddress
     * @return
     */
    public static String inetSocketAddressToString(InetSocketAddress inetSocketAddress) {
      return inetSocketAddress.getHostString() + ":" + inetSocketAddress.getPort();
    }

    // Method to convert single dimensional array index to Point x, y
    public static Point convertToXY(int i, int n) {
      int x = i % n;
      int y = i / n;

      return new Point(x, y);
    }

    // Method to convert Point x, y to single dimensional array index
    public static int convertToIndex(int x, int y, int n) {
      return y * n + x;
    }

    public static void waitForThreads(List<Thread> threads) {
      threads.forEach(thread -> {
        try {
          thread.join();
        } catch (InterruptedException ex) {
          ex.printStackTrace();
        }
      });
      threads.clear();
    }
  }


  // Matrix.java
  import java.io.Serializable;
  import java.util.Arrays;
  import java.util.logging.Logger;

  /******************************************************************************
   * Compilation: javac Matrix.java
   * Execution: java Matrix
   *
   * A bare-bones immutable data type for M-by-N matrices.
   *
   * Credit: The code is adapted from
   * https://introcs.cs.princeton.edu/java/95linear/Matrix.java.html
   ******************************************************************************/

  /**
   * Creating a class called Matrix.
   */
  final public class Matrix implements Serializable {
    private final int M; // number of rows
    private final int N; // number of columns
    private final double[][] data; // M-by-N array

    // Logger for this class
    private static final Logger LOGGER = Logger.getLogger(Matrix.class.getName());

    // create M-by-N matrix of 0's
    // Creating a new Matrix object with M rows and N columns.
    public Matrix(int M, int N) {
      this.M = M;
      this.N = N;
      data = new double[M][N];
    }

    /**
     * Returns the number of rows in the matrix
     *
     * @return The value of the instance variable M.
     */
    public int getM() {
      return M;
    }

    /**
     * Returns the number of elements in the array
     *
     * @return The number of elements in the array.
     */
    public int getN() {
      return N;
    }
```

```java
/**
 * Given an integer i and an integer j, set the value of the element at the i-th
 * row and j-th column
 * of the matrix to value
 *
 * @param i     The row of the matrix.
 * @param j     The column index of the element to be set.
 * @param value the value to set the cell to
 */
public void set(int i, int j, double value) {
  data[i][j] = value;
}

/**
 * Given an integer i and an integer j, return the value of the element at the
 * ith row and jth column
 * of the matrix
 *
 * @param i The row of the matrix.
 * @param j the column index
 * @return The value of the element at the specified indices.
 */
public double get(int i, int j) {
  return data[i][j];
}

/**
 * Given an index, return the row of the matrix
 *
 * @param i The row index.
 * @return The row of data.
 */
public double[] getRow(int i) {
  return data[i];
}

/**
 * Return a new array that contains the elements of the jth column of the matrix
 *
 * @param j the column index
 * @return A new array of doubles.
 */
public double[] getColumn(int j) {
  double[] column = new double[M];
  for (int i = 0; i < M; i++) {
    column[i] = data[i][j];
  }
  return column;
}

// The constructor takes a double[][] as an argument and creates a new
// double[][] that is a copy of the argument.
public Matrix(double[][] data) {
  M = data.length;
  N = data[0].length;
  this.data = new double[M][N];
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
```

```
      this.data[i][j] = data[i][j];
}

// Creating a new Matrix object with the same data as the Matrix A.
private Matrix(Matrix A) {
  this(A.data);
}

// create and return a random M-by-N matrix with values between 0 and 1
/**
 * Generate a random matrix of size M by N
 *
 * @param M the number of rows in the matrix
 * @param N the number of rows in the matrix
 * @return A new Matrix object.
 */
public static Matrix random(int M, int N) {
  Matrix A = new Matrix(M, N);
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
      A.data[i][j] = Math.random();
  return A;
}

/**
 * Generate a random matrix with elements in the range [min, max]
 *
 * @param M   the number of rows in the matrix
 * @param N   the number of rows in the matrix
 * @param min the minimum value of the random numbers
 * @param max The maximum value of the random numbers.
 * @return A new matrix object.
 */
public static Matrix random(int M, int N, double min, double max) {
  Matrix A = new Matrix(M, N);
  for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++)
      A.data[i][j] = min + (max - min) * Math.random();
  return A;
}

/**
 * Create a NxN matrix with the diagonal elements set to 1
 *
 * @param N the size of the matrix
 * @return A new Matrix object.
 */
public static Matrix identity(int N) {
  Matrix I = new Matrix(N, N);
  for (int i = 0; i < N; i++)
    I.data[i][i] = 1;
  return I;
}

/**
 * Swap the values of the data array at indices i and j
 *
 * @param i The index of the first element to swap.
 * @param j The index of the element to swap with.
```

```
     */
    private void swap(int i, int j) {
      double[] temp = data[i];
      data[i] = data[j];
      data[j] = temp;
    }

    /**
     * Transpose() returns a new matrix that is the transpose of this matrix
     *
     * @return A new matrix.
     */
    public Matrix transpose() {
      Matrix A = new Matrix(N, M);
      for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
          A.data[j][i] = this.data[i][j];
      return A;
    }

    /**
     * Add the elements of matrix B to the elements of matrix A
     *
     * @param B the matrix to be added to this matrix
     */
    public void plusInPlace(Matrix B) {
      if (B.M != M || B.N != N)
        throw new RuntimeException("Illegal matrix dimensions.");

      for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
          data[i][j] += +B.data[i][j];
    }

    /**
     * Add two matrices together
     *
     * @param B the matrix to be added to A
     * @return A new matrix C.
     */
    public Matrix plus(Matrix B) {
      Matrix A = this;
      if (B.M != A.M || B.N != A.N)
        throw new RuntimeException("Illegal matrix dimensions.");
      Matrix C = new Matrix(M, N);
      for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
          C.data[i][j] = A.data[i][j] + B.data[i][j];
      return C;
    }

    /**
     * Subtracts the matrix B from the matrix A and returns the result in matrix C
     *
     * @param B The matrix to subtract from this matrix.
     * @return A new matrix C.
     */
    public Matrix minus(Matrix B) {
      Matrix A = this;
      if (B.M != A.M || B.N != A.N)
        throw new RuntimeException("Illegal matrix dimensions.");
      Matrix C = new Matrix(M, N);
      for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
          C.data[i][j] = A.data[i][j] - B.data[i][j];
      return C;
    }

    /**
     * If the matrices are equal, return true, otherwise return false
     *
     * @param B the matrix to compare to A
     * @return A boolean value.
     */
    public boolean eq(Matrix B) {
      Matrix A = this;
      if (B.M != A.M || B.N != A.N)
        throw new RuntimeException("Illegal matrix dimensions.");
      for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++)
          if (A.data[i][j] != B.data[i][j])
            return false;
      return true;
    }

    /**
     * Given a number, return true if it is a power of two, else return false
     *
     * @param n The number to check if it's a power of two.
     * @return The return value is a boolean value.
     */
    static boolean isPowerOfTwo(int n) {
      return (int) (Math.ceil((Math.log(n) / Math.log(2)))) == (int) (Math.floor(((Math.log(n) /
  Math.log(2)))));
    }

    /**
     * Given a matrix, cut it into a smaller matrix of a specified size
     *
     * @param a    The matrix to be cut.
     * @param rows The number of rows to cut out of the matrix.
     * @param cols the number of columns to cut the matrix into
     * @return The matrix that is cut from the original matrix.
     */
    public static Matrix cut(Matrix a, int rows, int cols) {
      Matrix temp = new Matrix(rows, cols);

      for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
          temp.data[i][j] = a.data[i][j];
        }
      }

      return temp;
    }

    /**
     * Given a matrix, split it into two matrices
     *
     * @param childMatrixLength the length of the child matrix.
     * @param fromIndex         The index of the first row of the matrix to be
     *                          copied.
     * @param toIndex           The index of the last element in the row of the
     *                          parent matrix that will be copied to
     *                          the child matrix.
     * @return The child matrix.
     */
    public Matrix split(int childMatrixLength, int fromIndex, int toIndex) {
      Matrix child = new Matrix(childMatrixLength, childMatrixLength);

      for (int i1 = 0, i2 = fromIndex; i1 < childMatrixLength; i1++, i2++)
        for (int j1 = 0, j2 = toIndex; j1 < childMatrixLength; j1++, j2++)
          child.data[i1][j1] = data[i2][j2];

      return child;
    }

    /**
     * Divide the matrix into smaller matrices of size childMatrixLength
     *
     * @param childMatrixLength The length of the child matrix.
     * @return An array of matrices.
     */
    public Matrix[] divide(int childMatrixLength) {
      // childMatrixLength should be less than or equal to M throw error
      if (childMatrixLength > M) {
        throw new IllegalArgumentException("childMatrixLength should be less than or equal to M");
      }

      // childMatrixLength should be a power of 2 throw error
      if (!isPowerOfTwo(childMatrixLength)) {
        throw new IllegalArgumentException("childMatrixLength should be a power of 2");
      }

      int len = M / childMatrixLength;

      Matrix[] groups = new Matrix[len * len];
      int multiplier = M / len;

      for (int i = 0; i < len; i++) {
        for (int j = 0; j < len; j++) {
          groups[i * len + j] = split(childMatrixLength, i * multiplier, j * multiplier);
        }
      }

      return groups;
    }

    /**
     * Given a matrix P, join the matrix to the matrix at the given index
     *
     * @param P         the matrix to be joined with this matrix
     * @param fromIndex the index of the first row of the matrix to be joined.
     * @param toIndex   the index of the first element in the new matrix.
     */
    public void join(Matrix P, int fromIndex, int toIndex) {
      for (int i1 = 0, i2 = fromIndex; i1 < P.getM(); i1++, i2++) {
        for (int j1 = 0, j2 = toIndex; j1 < P.getN(); j1++, j2++) {
          data[i2][j2] = P.data[i1][j1];
        }
      }
    }

    /**
     * Join all the matrices in the array into one matrix
     *
     * @param matrices an array of matrices to join
     */
    public void joinAll(Matrix[] matrices) {
      int childMatrixLength = matrices[0].getM();
      int len = M / childMatrixLength;
      int multiplier = M / len;

      for (int i = 0; i < len; i++) {
        for (int j = 0; j < len; j++) {
          join(matrices[i * len + j], i * multiplier, j * multiplier);
        }
      }
    }

    // Given a matrix, pad it with zeros until it is a power of 2.
    // Credit:
    //
  https://github.com/liangyue268/CPE-593/blob/00b590f46666f7bbf7a72af09dcb57d59f52c2d6/HW6/Strassen.java
    public static Matrix padding(Matrix a) {
      int length = (int) Math.pow(2, Math.ceil(Math.log(Math.max(a.getM(), a.getN())) / Math.log(2)));

      Matrix paddedMatrix = new Matrix(length, length);

      for (int i = 0; i < a.getM(); i++) {
        for (int j = 0; j < a.getN(); j++) {
          paddedMatrix.data[i][j] = a.data[i][j];
        }
        for (int j = a.getN(); j < length; j++) {
          paddedMatrix.data[i][j] = 0;
        }
      }

      for (int i = a.getM(); i < length; i++) {
        for (int j = 0; j < length; j++) {
          paddedMatrix.data[i][j] = 0;
        }
      }
      return paddedMatrix;
    }

    // For each of the four submatrices, copy the submatrix into the appropriate
    // location in the original matrix.
    public static void resetMatrix(Matrix mtx, Matrix submtx, int index) {

      for (int i = 0; i < submtx.M; i++) {
        for (int j = 0; j < submtx.N; j++) {
          switch (index) {
            case 0:
              mtx.data[i][j] = submtx.data[i][j];
              break;
            case 1:
              mtx.data[i][j + submtx.N] = submtx.data[i][j];
              break;
            case 2:
              mtx.data[i + submtx.M][j] = submtx.data[i][j];
              break;
            case 3:
              mtx.data[i + submtx.M][j + submtx.N] = submtx.data[i][j];
              break;
          }
        }
      }
    }

    /**
     * Multiply the matrix A by the matrix B, and store the result in the matrix C
     *
     * @param B the matrix to be multiplied
     * @return The product of A and B.
     */
    public Matrix times(Matrix B) {
      Matrix A = this;
      if (A.N != B.M)
        throw new RuntimeException("Illegal matrix dimensions.");
      Matrix C = new Matrix(A.M, B.N);
      for (int i = 0; i < C.M; i++)
        for (int j = 0; j < C.N; j++)
          for (int k = 0; k < A.N; k++)
            C.data[i][j] += (A.data[i][k] * B.data[k][j]);
      return C;
    }

    /**
     * It computes the dot product of A and B.
     *
     * @param A an array of matrices
     * @param B the matrix that is being multiplied
     * @return The result of the matrix multiplication.
     */
    public static Matrix dot(Matrix[] A, Matrix[] B) { // 10 chunks example in A and B so create 10
  threads
      Matrix result = new Matrix(A[0].getM(), A[0].getN());

      for (int i = 0; i < A.length; i++) {
        result = result.plus(A[i].times(B[i]));
      }

      return result;
    }

    /**
     * Given a matrix A and a vector b, solve for x in Ax = b
     *
     * @param rhs the right hand side of the equation, a matrix with as many rows as
     *            the
     * @return The solution matrix.
     */
    public Matrix solve(Matrix rhs) {
      if (M != N || rhs.M != N || rhs.N != 1)
```

```
      throw new RuntimeException("Illegal matrix dimensions.");

    // create copies of the data
    Matrix A = new Matrix(this);
    Matrix b = new Matrix(rhs);

    // Gaussian elimination with partial pivoting
    for (int i = 0; i < N; i++) {

      // find pivot row and swap
      int max = i;
      for (int j = i + 1; j < N; j++)
        if (Math.abs(A.data[j][i]) > Math.abs(A.data[max][i]))
          max = j;
      A.swap(i, max);
      b.swap(i, max);

      // singular
      if (A.data[i][i] == 0.0)
        throw new RuntimeException("Matrix is singular.");

      // pivot within b
      for (int j = i + 1; j < N; j++)
        b.data[j][0] -= b.data[i][0] * A.data[j][i] / A.data[i][i];

      // pivot within A
      for (int j = i + 1; j < N; j++) {
        double m = A.data[j][i] / A.data[i][i];
        for (int k = i + 1; k < N; k++) {
          A.data[j][k] -= A.data[i][k] * m;
        }
        A.data[j][i] = 0.0;
      }
    }

    // back substitution
    Matrix x = new Matrix(N, 1);
    for (int j = N - 1; j >= 0; j--) {
      double t = 0.0;
      for (int k = j + 1; k < N; k++)
        t += A.data[j][k] * x.data[k][0];
      x.data[j][0] = (b.data[j][0] - t) / A.data[j][j];
    }
    return x;

  }

  /**
   * Prints the matrix to the standard output
   */
  public void show() {
    for (int i = 0; i < M; i++) {
      for (int j = 0; j < N; j++)
        System.out.printf("%9.4f ", data[i][j]);
      System.out.println();
    }
  }

  /**
```

```
     * Prints the matrix to the standard output
     *
     * @param msg The message to display.
     */
    public void show(String msg) {
      System.out.println(msg + ":");
      for (int i = 0; i < M; i++) {
        for (int j = 0; j < N; j++)
          System.out.printf("%9.4f ", data[i][j]);
        System.out.println();
      }
    }

    // test client
    public static void main(String[] args) {
    }
  }
```