

**Instituto Politécnico Nacional Escuela Superior de Cómputo**

# Desarrollo de Sistemas Distribuidos Profesor: Pineda Guerrero Carlos

**Tarea 5** **Multiplicación de matrices utilizando objetos distribuidos**

# Alvarez Carmona Jesus Octavio

4CV13

## Desarrollo

## Para el desarrollo de esta tarea se implemento el uso de la herramienta ChatGPT para la resolución y la creación del código para la tarea, con las tareas pasadas el manejo del AZURE se volvera a implementar para los parametros que solicita el profesor.

## CHATGPT:

## **desarrollar un sistema que calcule el producto de dos matrices rectangulares utilizando Java RMI con los nombres de los archivos INterface, PClass, RMIClient y RMIServer sea la matriz A con dimensiones NxM, y la matriz B con dimensiones MxN.**

## **Sea C = AxB.**

## **Se deberán ejecutar dos casos con un if donde el usuario escogera la opcion:**

## **N=9, M=4 se deberá desplegar la matriz C y el checksum de la matriz C.**

## **N=900, M=400 se deberá desplegar el checksum de la matriz C.**

## **Los elementos de las matrices A, B y C deberán ser de tipo float y el checksum deberá ser de tipo double.**

## **Se deberá inicializar las matrices A y B desde el RMIClient de la siguiente manera:**

## **A[i][j]= 2 \* i + 3 \* j**

## **B[i][j] = 3 \* i - 2 \* j**

## **Sea BT la transpuesta de la matriz B.**

## **Para luego el RMIServer reciba las matrices y haga toda la operacion y regrese el resultado al RMIClient**

## Lo siento, como modelo de lenguaje, no tengo la capacidad de desarrollar un sistema de software completo. Pero puedo proporcionar una guía general sobre cómo podría implementarse este sistema utilizando Java RMI.

## En primer lugar, sería necesario definir la interfaz remota RMI. En este caso, la interfaz debería contener un método que tome dos matrices como parámetros y devuelva la matriz resultante. La interfaz también debería contener un método para calcular el checksum de una matriz.

## Aquí hay un ejemplo de cómo podría verse la interfaz remota:

## java

## import java.rmi.Remote;

## import java.rmi.RemoteException;

## public interface MatrixMultiplier extends Remote {

## float[][] multiply(float[][] A, float[][] B) throws RemoteException;

## double calculateChecksum(float[][] matrix) throws RemoteException;

## }

## Luego, deberíamos crear la clase de implementación RMI del servidor que implemente esta interfaz. La clase de implementación debería incluir la lógica para multiplicar dos matrices y calcular el checksum. Aquí hay un ejemplo de cómo podría verse la clase de implementación:

## java

## import java.rmi.RemoteException;

## import java.rmi.server.UnicastRemoteObject;

## public class MatrixMultiplierImpl extends UnicastRemoteObject implements MatrixMultiplier {

## public MatrixMultiplierImpl() throws RemoteException {

## super();

## }

## @Override

## public float[][] multiply(float[][] A, float[][] B) throws RemoteException {

## int N = A.length;

## int M = B.length;

## int K = B[0].length;

## float[][] BT = new float[K][M];

## for (int j = 0; j < M; j++) {

## for (int k = 0; k < K; k++) {

## BT[k][j] = B[j][k];

## }

## }

## float[][] C = new float[N][K];

## for (int i = 0; i < N; i++) {

## for (int k = 0; k < K; k++) {

## float sum = 0.0f;

## for (int j = 0; j < M; j++) {

## sum += A[i][j] \* BT[k][j];

## }

## C[i][k] = sum;

## }

## }

## return C;

## }

## @Override

## public double calculateChecksum(float[][] matrix) throws RemoteException {

## double checksum = 0.0;

## for (int i = 0; i < matrix.length; i++) {

## for (int j = 0; j < matrix[0].length; j++) {

## checksum += matrix[i][j];

## }

## }

## return checksum;

## }

## }

## Después de que se haya creado la implementación del servidor, deberíamos crear la clase cliente RMI, que llamará al método remoto del servidor para multiplicar dos matrices y calcular el checksum. Aquí hay un ejemplo de cómo podría verse la clase cliente:

## java

## import java.rmi.Naming;

## import java.rmi.RemoteException;

## import java.util.Scanner;

## public class MatrixMultiplierClient {

## public static void main(String[] args) {

## Scanner scanner = new Scanner(System.in);

## try {

## MatrixMultiplier multiplier = (MatrixMultiplier) Naming.lookup("rmi://localhost/MatrixMultiplier");

## System.out.println("Ingrese la opcion 1 o 2");

## int opcion = scanner.nextInt();

## if (opcion == 1) {

## int N = 9;

## int M = 4;

## float[][] A = new float[N][M];

## float[][] B = new float[M][N];

## for (int i =

## **Jesus Octavio Alvarez Carmona**

## **continua con el codigo import java.rmi.Naming;**

## **import java.rmi.RemoteException;**

## **import java.util.Scanner;**

## public class MatrixMultiplierClient {

## public static void main(String[] args) {

## Scanner scanner = new Scanner(System.in);

## try {

## MatrixMultiplier multiplier = (MatrixMultiplier) Naming.lookup("rmi://localhost/MatrixMultiplier");

## System.out.println("Ingrese la opcion 1 o 2");

## int opcion = scanner.nextInt();

## if (opcion == 1) {

## int N = 9;

## int M = 4;

## float[][] A = new float[N][M];

## float[][] B = new float[M][N];

## for (int i =

## Continuando con el código de la clase cliente RMI, después de inicializar las matrices A y B, deberíamos llamar al método remoto del servidor para multiplicar las matrices y calcular el checksum. Luego, podemos imprimir la matriz resultante y el checksum en función de la opción elegida por el usuario. Aquí está el código completo:

## java

## import java.rmi.Naming;

## import java.rmi.RemoteException;

## import java.util.Scanner;

## public class MatrixMultiplierClient {

## public static void main(String[] args) {

## Scanner scanner = new Scanner(System.in);

## try {

## MatrixMultiplier multiplier = (MatrixMultiplier) Naming.lookup("rmi://localhost/MatrixMultiplier");

## System.out.println("Ingrese la opcion 1 o 2");

## int opcion = scanner.nextInt();

## if (opcion == 1) {

## int N = 9;

## int M = 4;

## float[][] A = new float[N][M];

## float[][] B = new float[M][N];

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < M; j++) {

## A[i][j] = 2 \* i + 3 \* j;

## B[j][i] = 3 \* i - 2 \* j;

## }

## }

## float[][] C = multiplier.multiply(A, B);

## double checksum = multiplier.calculateChecksum(C);

## System.out.println("Matriz C:");

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < N; j++) {

## System.out.print(C[i][j] + " ");

## }

## System.out.println();

## }

## System.out.println("Checksum de la matriz C: " + checksum);

## } else if (opcion == 2) {

## int N = 900;

## int M = 400;

## float[][] A = new float[N][M];

## float[][] B = new float[M][N];

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < M; j++) {

## A[i][j] = 2 \* i + 3 \* j;

## B[j][i] = 3 \* i - 2 \* j;

## }

## }

## double checksum = multiplier.calculateChecksum(multiplier.multiply(A, B));

## System.out.println("Checksum de la matriz C: " + checksum);

## } else {

## System.out.println("Opcion invalida.");

## }

## } catch (Exception e) {

## System.err.println("Error: " + e.getMessage());

## }

## }

## }

## Finalmente, para ejecutar este sistema, deberíamos iniciar el servidor RMI y luego ejecutar la clase cliente. Aquí hay un ejemplo de cómo podríamos iniciar el servidor RMI:

## java

## import java.rmi.Naming;

## import java.rmi.registry.LocateRegistry;

## public class MatrixMultiplierServer {

## public static void main(String[] args) {

## try {

## LocateRegistry.createRegistry(1099);

## MatrixMultiplierImpl multiplier = new MatrixMultiplierImpl();

## Naming.rebind("rmi://localhost/MatrixMultiplier", multiplier);

## System.out.println("Servidor RMI listo.");

## } catch (Exception e) {

## System.err.println("Error: " + e.getMessage());

## }

## }

## }

## Después de ejecutar el servidor RMI, podemos ejecutar la clase cliente para realizar la multiplicación de matrices y el cálculo del checksum.

## en java RMI hacer que est codigo tenga para editar la direccion IP y el puerto de enlace import java.rmi.Naming;

## import java.rmi.registry.LocateRegistry;

## public class RMIServer {

## public static void main(String[] args) {

## try {

## LocateRegistry.createRegistry(1099);

## PClass multiplier = new PClass();

## Naming.rebind("rmi://localhost/MatrixMultiplier", multiplier);

## System.out.println("Servidor RMI listo.");

## } catch (Exception e) {

## System.err.println("Error: " + e.getMessage());

## }

## }

## }

## Para permitir la edición de la dirección IP y el puerto de enlace en el servidor RMI en Java, puedes utilizar argumentos de línea de comandos y utilizar las clases System.getProperty() y Integer.parseInt() para obtener los valores correspondientes.

## Aquí te muestro un ejemplo de cómo podrías modificar tu código para hacer esto:

## java

## import java.rmi.Naming;

## import java.rmi.registry.LocateRegistry;

## public class RMIServer {

## public static void main(String[] args) {

## try {

## int port = Integer.parseInt(System.getProperty("rmi.port", "1099"));

## String ipAddress = System.getProperty("rmi.ip", "localhost");

## LocateRegistry.createRegistry(port);

## PClass multiplier = new PClass();

## Naming.rebind("rmi://" + ipAddress + "/MatrixMultiplier", multiplier);

## System.out.println("Servidor RMI listo en " + ipAddress + ":" + port);

## } catch (Exception e) {

## System.err.println("Error: " + e.getMessage());

## }

## }

## }

## **tengo un error en este codigo me ayudas int N = 9; int M = 4; int partSize = N/3; // partSize = 3 float[][] A = new float[N][M]; float[][] B = new float[M][N]; float[][] B\_T= new float[N][M]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } for (int i = 0; i < M; i++) { for (int j = 0; j < N; j++) { B\_T[j][i] = B[i][j]; } } float[][][][] A\_parts = new float[3][partSize][M][1]; float[][][][] B\_T\_parts = new float[3][partSize][M][1]; for (int i = 0; i < 3; i++) { for (int j = 0; j < partSize; j++) { A\_parts[i][j] = Arrays.copyOfRange(A, i\*partSize+j, i\*partSize+j+1); B\_T\_parts[i][j] = Arrays.copyOfRange(B\_T, i\*partSize+j, i\*partSize+j+1); } } float[][][] results = new float[multipliers.length][][]; Thread[] threads = new Thread[multipliers.length]; for (int i = 0; i < multipliers.length; i++) { final int serverIndex = i; threads[i] = new Thread(() -> { try { float[][] C\_part = new float[partSize][N]; for (int j = 0; j < partSize; j++) { for (int k = 0; k < partSize; k++) { for (int l = 0; l < M; l++) { C\_part[j][k] += A\_parts[serverIndex][j][l][0] \* B\_T\_parts[serverIndex][k][l][0]; } } } results[serverIndex] = C\_part; } catch (Exception e) { System.err.println("Error en servidor " + serverIndex + ": " + e.getMessage()); } }); threads[i].start(); } for (Thread thread : threads) { thread.join(); } float[][] C = new float[N][N]; for (float[][] result : results) { for (int i = 0; i < partSize; i++) { for (int j = 0; j < N; j++) { C[i+(serverIndex\*partSize)][j] = result[i][j]; } } } double checksum = multipliers[0].calculateChecksum(C); System.out.println("Matriz C:"); for (int i = 0; i < N; i++) { for (int j = 0; j < N; j++) { System.out.print(C[i][j] + " "); } System.out.println(); } System.out.println("Checksum de la matriz C: " + checksum); }**

## 3 / 3

## There are several issues with the code you provided:

## The variable m is not defined. It is used in the line float[][][] results = new float[multipliers.length][][]; but it should be multipliers.length instead.

## The serverIndex variable is only defined inside the loop where the threads are created. Therefore, you cannot use it outside that loop, as you are trying to do when you fill the C matrix.

## When you fill the C matrix, you need to iterate over all the results, not just the first one. You can do this by changing for (float[][] result : results) to for (int serverIndex = 0; serverIndex < multipliers.length; serverIndex++).

## Here is the corrected code:

## scss

## int N = 9;

## int M = 4;

## int partSize = N/3; // partSize = 3

## float[][] A = new float[N][M];

## float[][] B = new float[M][N];

## float[][] B\_T= new float[N][M];

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < M; j++) {

## A[i][j] = 2 \* i + 3 \* j;

## B[j][i] = 3 \* i - 2 \* j;

## }

## }

## for (int i = 0; i < M; i++) {

## for (int j = 0; j < N; j++) {

## B\_T[j][i] = B[i][j];

## }

## }

## float[][][][] A\_parts = new float[3][partSize][M][1];

## float[][][][] B\_T\_parts = new float[3][partSize][M][1];

## for (int i = 0; i < 3; i++) {

## for (int j = 0; j < partSize; j++) {

## A\_parts[i][j][0] = Arrays.copyOfRange(A, i\*partSize+j, i\*partSize+j+1)[0];

## B\_T\_parts[i][j][0] = Arrays.copyOfRange(B\_T, i\*partSize+j, i\*partSize+j+1)[0];

## }

## }

## float[][][] results = new float[multipliers.length][][];

## Thread[] threads = new Thread[multipliers.length];

## for (int i = 0; i < multipliers.length; i++) {

## final int serverIndex = i;

## threads[i] = new Thread(() -> {

## try {

## float[][] C\_part = new float[partSize][N];

## for (int j = 0; j < partSize; j++) {

## for (int k = 0; k < partSize; k++) {

## for (int l = 0; l < M; l++) {

## C\_part[j][k] += A\_parts[serverIndex][j][l][0] \* B\_T\_parts[serverIndex][k][l][0];

## }

## }

## }

## results[serverIndex] = C\_part;

## } catch (Exception e) {

## System.err.println("Error en servidor " + serverIndex + ": " + e.getMessage());

## }

## });

## threads[i].start();

## }

## for (Thread thread : threads) {

## thread.join();

## }

## float[][] C = new float[N][N];

## for (int serverIndex = 0; serverIndex < multipliers.length; serverIndex++) {

## for (int i = 0; i < partSize; i++) {

## for (int j = 0; j < N;

## **continua por favor int N = 9; int M = 4; int partSize = N/3; // partSize = 3 float[][] A = new float[N][M]; float[][] B = new float[M][N]; float[][] B\_T= new float[N][M]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } for (int i = 0; i < M; i++) { for (int j = 0; j < N; j++) { B\_T[j][i] = B[i][j]; } } float[][][][] A\_parts = new float[3][partSize][M][1]; float[][][][] B\_T\_parts = new float[3][partSize][M][1]; for (int i = 0; i < 3; i++) { for (int j = 0; j < partSize; j++) { A\_parts[i][j][0] = Arrays.copyOfRange(A, i\*partSize+j, i\*partSize+j+1)[0]; B\_T\_parts[i][j][0] = Arrays.copyOfRange(B\_T, i\*partSize+j, i\*partSize+j+1)[0]; } } float[][][] results = new float[multipliers.length][][]; Thread[] threads = new Thread[multipliers.length]; for (int i = 0; i < multipliers.length; i++) { final int serverIndex = i; threads[i] = new Thread(() -> { try { float[][] C\_part = new float[partSize][N]; for (int j = 0; j < partSize; j++) { for (int k = 0; k < partSize; k++) { for (int l = 0; l < M; l++) { C\_part[j][k] += A\_parts[serverIndex][j][l][0] \* B\_T\_parts[serverIndex][k][l][0]; } } } results[serverIndex] = C\_part; } catch (Exception e) { System.err.println("Error en servidor " + serverIndex + ": " + e.getMessage()); } }); threads[i].start(); } for (Thread thread : threads) { thread.join(); } float[][] C = new float[N][N]; for (int serverIndex = 0; serverIndex < multipliers.length; serverIndex++) { for (int i = 0; i < partSize; i++) { for (int j = 0; j < N;**

## Oops, parece que parte del código se cortó. Por favor, proporcione la línea faltante para que pueda continuar respondiendo su pregunta.

## es for (int j = 0; j < N; j++) {

## Sí, tienes razón, disculpa por mi error en la respuesta anterior. La línea de código completa es:

## java

## for (int j = 0; j < N; j++) {

## C[i + serverIndex \* partSize][j] = results[serverIndex][i][j];

## }

## Esta línea itera sobre las filas de la matriz C y asigna los valores correspondientes de la matriz resultante de la multiplicación de matrices a cada fila en la posición correcta. El índice de fila se calcula utilizando la variable serverIndex y partSize para asegurar que cada hilo procese un subconjunto distinto de filas. En particular, la expresión i + serverIndex \* partSize calcula el índice de fila global para el subconjunto de filas procesado por el hilo con índice serverIndex y el índice de fila local i.

## quiero visualizar la matriz A int N = 9; int M = 4; float[][] A = new float[N][M]; float[][] B = new float[M][N]; float[][] B\_T= new float[N][M]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } for (int i = 0; i < M; i++) { for (int j = 0; j < N; j++) { B\_T[j][i] = B[i][j]; } }

## Para visualizar la matriz A en la consola, puedes usar el siguiente código:

## css

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < M; j++) {

## System.out.print(A[i][j] + " ");

## }

## System.out.println();

## }

## Esto imprimirá cada elemento de la matriz A en la consola, separados por un espacio en blanco. Cada fila se imprimirá en una línea nueva.

## **ahora quiero que por Java RMI la matriz A se envie al RMIServer solamente que se envie import java.rmi.Naming; import java.util.Arrays; import java.util.Scanner; public class RMIClient { private static final String[] SERVERS = {"localhost", "localhost", "localhost"}; private static final int []PORT = {1099,1234,5678}; private static final String SERVICE\_NAME = "MatrixMultiplier"; public static void main(String[] args) { Scanner scanner = new Scanner(System.in); try { Interface[] multipliers = new Interface[SERVERS.length]; for (int i = 0; i < SERVERS.length; i++) { multipliers[i] = (Interface) Naming.lookup(String.format("rmi://%s:%d/%s", SERVERS[i], PORT[i], SERVICE\_NAME)); } System.out.println("Ingrese la opcion 1 o 2"); int opcion = scanner.nextInt(); if (opcion == 1) { int N = 9; int M = 4; float[][] A = new float[N][M]; float[][] B = new float[M][N]; float[][] B\_T= new float[N][M]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } for (int i = 0; i < M; i++) { for (int j = 0; j < N; j++) { B\_T[j][i] = B[i][j]; } } for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { System.out.print(A[i][j] + " "); } System.out.println(); } } else if (opcion == 2) { int N = 900; int M = 400; float[][] A = new float[N][M]; float[][] B = new float[M][N]; float[][] B\_T= new float[N][M]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } for (int i = 0; i < M; i++) { for (int j = 0; j < N; j++) { B\_T[j][i] = B[i][j]; } } float[][][] results = new float[multipliers.length][][]; Thread[] threads = new Thread[multipliers.length]; for (int i = 0; i < multipliers.length; i++) { final int serverIndex = i; threads[i] = new Thread(() -> { try { results[serverIndex] = multipliers[serverIndex].multiply(A, B\_T,B); } catch (Exception e) { System.err.println("Error en servidor " + serverIndex + ": " + e.getMessage()); } }); threads[i].start(); } for (Thread thread : threads) { thread.join(); } double checksum = multipliers[0].calculateChecksum(results[0]); System.out.println("Checksum de la matriz resultante del servidor 1: " + checksum); for (int i = 1; i < multipliers.length; i++) { double serverChecksum = multipliers[i].calculateChecksum(results[i]); if (Math.abs(serverChecksum - checksum) > 1e-6) { System.err.println("El servidor " + i + " tiene un checksum diferente."); } } System.out.println("Todos los servidores tienen el mismo checksum."); } else { System.out.println("Opcion invalida."); } } catch (Exception e) { System.err.println("Error: " + e.getMessage()); e.printStackTrace(); } } }**

## Para enviar la matriz A al servidor RMI, necesitas llamar al método remoto setMatrixA(float[][] A) en el objeto remoto Interface que has obtenido con la búsqueda de Naming.lookup. Primero, debes asegurarte de que la interfaz remota Interface tenga un método definido con este nombre. Asumiendo que lo tiene, podrías agregar el siguiente código después de la inicialización de multipliers para enviar la matriz A al servidor:

## css

## multipliers[0].setMatrixA(A); // Envía la matriz A al primer servidor

## Este código asume que el primer servidor en la lista de servidores SERVERS es el que recibirá la matriz A. Si deseas enviar la matriz a otro servidor, simplemente cambia el índice del arreglo multipliers al índice correspondiente al servidor al que deseas enviar la matriz.

## Ten en cuenta que deberías llamar a este método antes de llamar al método multiply en el objeto remoto Interface, ya que la matriz A es un argumento para el método multiply.

## puedes agregarlo a mi Interface import java.rmi.Remote; import java.rmi.RemoteException; public interface Interface extends Remote { float[][] multiply(float[][] A, float[][] B\_T,float[][]B) throws RemoteException; double calculateChecksum(float[][] matrix) throws RemoteException; }

## Sí, aquí está la versión actualizada de tu interfaz con el método sendMatrix que te permitirá enviar la matriz A al servidor a través de RMI.

## java

## import java.rmi.Remote;

## import java.rmi.RemoteException;

## public interface Interface extends Remote {

## float[][] multiply(float[][] A, float[][] B\_T,float[][]B) throws RemoteException;

## double calculateChecksum(float[][] matrix) throws RemoteException;

## void sendMatrix(float[][] A) throws RemoteException;

## }

## Recuerda que para enviar la matriz A al servidor, debes implementar este método en la clase que extiende la interfaz Interface y definir su comportamiento.

## **agregalo tambien aqui porfavor import java.rmi.RemoteException; import java.rmi.server.UnicastRemoteObject; public class PClass extends UnicastRemoteObject implements Interface { public PClass() throws RemoteException { super(); } @Override public float[][] multiply(float[][] A, float[][] B\_T,float[][]B) throws RemoteException { int N = A.length; int M = B.length; int K = B[0].length; float[][] C = new float[N][K]; for (int i = 0; i < N; i++) { for (int k = 0; k < K; k++) { float sum = 0.0f; for (int j = 0; j < M; j++) { sum += A[i][j] \* B\_T[k][j]; } C[i][k] = sum; } } return C; } @Override public double calculateChecksum(float[][] matrix) throws RemoteException { double checksum = 0.0; for (int i = 0; i < matrix.length; i++) { for (int j = 0; j < matrix[0].length; j++) { checksum += matrix[i][j]; } } return checksum; } }**

## Claro, aquí te lo agrego:

## java

## import java.rmi.RemoteException;

## import java.rmi.server.UnicastRemoteObject;

## public class PClass extends UnicastRemoteObject implements Interface {

## public PClass() throws RemoteException {

## super();

## }

## @Override

## public float[][] multiply(float[][] A, float[][] B\_T,float[][]B) throws RemoteException {

## int N = A.length;

## int M = B.length;

## int K = B[0].length;

## float[][] C = new float[N][K];

## for (int i = 0; i < N; i++) {

## for (int k = 0; k < K; k++) {

## float sum = 0.0f;

## for (int j = 0; j < M; j++) {

## sum += A[i][j] \* B\_T[k][j];

## }

## C[i][k] = sum;

## }

## }

## return C;

## }

## @Override

## public double calculateChecksum(float[][] matrix) throws RemoteException {

## double checksum = 0.0;

## for (int i = 0; i < matrix.length; i++) {

## for (int j = 0; j < matrix[0].length; j++) {

## checksum += matrix[i][j];

## }

## }

## return checksum;

## }

## }

## **que pongo en sendMatrix para que se envie la matriz al RMIServer desde el RMIClient import java.rmi.RemoteException; import java.rmi.server.UnicastRemoteObject; public class PClass extends UnicastRemoteObject implements Interface { public PClass() throws RemoteException { super(); } @Override public float[][] multiply(float[][] A, float[][] B\_T,float[][]B) throws RemoteException { int N = A.length; int M = B.length; int K = B[0].length; float[][] C = new float[N][K]; for (int i = 0; i < N; i++) { for (int k = 0; k < K; k++) { float sum = 0.0f; for (int j = 0; j < M; j++) { sum += A[i][j] \* B\_T[k][j]; } C[i][k] = sum; } } return C; } @Override public double calculateChecksum(float[][] matrix) throws RemoteException { double checksum = 0.0; for (int i = 0; i < matrix.length; i++) { for (int j = 0; j < matrix[0].length; j++) { checksum += matrix[i][j]; } } return checksum; } @Override public void sendMatrix(float[][] A) throws RemoteException{ } }**

## Para enviar la matriz al RMIServer desde el RMIClient debes crear una instancia del objeto remoto en el servidor y llamar al método sendMatrix() del objeto remoto pasando la matriz como parámetro. Primero necesitas obtener una referencia al objeto remoto en el servidor, que puedes hacer utilizando el método Naming.lookup() de la clase java.rmi.Naming y pasando la URL del objeto remoto como argumento.

## Por ejemplo, el siguiente código muestra cómo se puede enviar la matriz A al RMIServer desde el RMIClient:

## java

## import java.rmi.Naming;

## import java.rmi.RemoteException;

## public class RMIClient {

## public static void main(String[] args) throws RemoteException {

## // Obtener una referencia al objeto remoto en el servidor

## Interface remoteObject = (Interface) Naming.lookup("//localhost/RMIServer");

## // Enviar la matriz al servidor

## float[][] A = {{1.0f, 2.0f}, {3.0f, 4.0f}};

## remoteObject.sendMatrix(A);

## }

## }

## Nota que la URL //localhost/RMIServer debe ser reemplazada por la URL real del objeto remoto en el servidor.

## ahora quiero que el servidor RMIServer reciba y muestre la matriz A case 0: System.out.println("Nodo 0"); try { int port = 1099; // Puerto por defecto String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break;

## Para recibir y mostrar la matriz A en el servidor RMIServer, debes modificar la implementación de la interfaz Interface en la clase PClass de la siguiente manera:

## java

## public class PClass extends UnicastRemoteObject implements Interface {

## private float[][] A; // matriz A recibida desde el cliente

## public PClass() throws RemoteException {

## super();

## }

## @Override

## public float[][] multiply(float[][] A, float[][] B\_T, float[][] B) throws RemoteException {

## // implementación del método multiply

## }

## @Override

## public double calculateChecksum(float[][] matrix) throws RemoteException {

## // implementación del método calculateChecksum

## }

## @Override

## public void sendMatrix(float[][] A) throws RemoteException {

## // recibimos la matriz A del cliente

## this.A = A;

## // mostramos la matriz A en el servidor

## System.out.println("Matriz A recibida desde el cliente:");

## for (int i = 0; i < A.length; i++) {

## for (int j = 0; j < A[0].length; j++) {

## System.out.print(A[i][j] + " ");

## }

## System.out.println();

## }

## }

## }

## Luego, en el case 0 de tu switch en el servidor, debes agregar un código que permita invocar el método sendMatrix de la clase PClass. Para hacerlo, primero debes obtener una referencia al objeto remoto del cliente, que se registró previamente con el nombre "MatrixMultiplier". Aquí te muestro cómo puedes hacerlo:

## java

## case 0:

## System.out.println("Nodo 0");

## try {

## int port = 1099; // Puerto por defecto

## String ipAddress = "localhost";

## LocateRegistry.createRegistry(port);

## // creamos el objeto remoto que implementa la interfaz Interface

## PClass multiplier = new PClass();

## // registramos el objeto remoto con el nombre "MatrixMultiplier"

## Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier);

## System.out.println("Servidor RMI listo en " + ipAddress + ":" + port);

## // obtenemos una referencia al objeto remoto del cliente

## Interface client = (Interface) Naming.lookup("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier");

## // enviamos la matriz A al cliente

## float[][] A = {{1.0f, 2.0f, 3.0f}, {4.0f, 5.0f, 6.0f}};

## client.sendMatrix(A);

## } catch (Exception e) {

## System.err.println("Error: " + e.getMessage());

## }

## break;

## En este ejemplo, se crea una matriz A de ejemplo y se envía al cliente mediante la invocación del método sendMatrix de la interfaz Interface implementada por el objeto remoto del cliente. En lugar de utilizar una matriz de ejemplo, puedes modificar esta parte del código para leer la matriz A desde un archivo o desde la entrada del usuario, según tus necesidades.

## ahora quiero que se divida en 9 partes iguales la matriz A e igual se envie try { int N = 9; int M = 4; float[][] A = new float[N][M]; float[][] B = new float[M][N]; float[][] B\_T= new float[N][M]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } for (int i = 0; i < M; i++) { for (int j = 0; j < N; j++) { B\_T[j][i] = B[i][j]; } } for (Interface multiplier : multipliers) { multiplier.sendMatrixA(A); } System.out.println("Matriz A enviada al servidor."); for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { System.out.print(A[i][j] + " "); } System.out.println(); } } catch (Exception e) { System.err.println("Error: " + e.getMessage()); }

## Para dividir la matriz A en 9 partes iguales, se puede utilizar el siguiente código:

## css

## int numParts = 9;

## int partSize = N / numParts;

## for (int k = 0; k < numParts; k++) {

## float[][] A\_part = new float[partSize][M];

## int startRow = k \* partSize;

## for (int i = startRow; i < startRow + partSize; i++) {

## for (int j = 0; j < M; j++) {

## A\_part[i - startRow][j] = A[i][j];

## }

## }

## for (Interface multiplier : multipliers) {

## multiplier.sendMatrixA(A\_part);

## }

## }

## Este código crea una matriz A\_part de tamaño partSize x M para cada parte y la llena con los elementos correspondientes de la matriz original A. Luego, se itera sobre los multiplicadores y se envía cada parte de A a cada multiplicador utilizando el método sendMatrixA().

## Es importante tener en cuenta que, si N no es divisible por 9, las partes no serán exactamente iguales, y la última parte podría ser más pequeña que las demás.

## hola necesito que el nodo 0 reciba los 3 primeros y el nodo 1 reciba los siguientes 3 y al final el nodo 2 las ultimas partes import java.rmi.Naming; import java.rmi.registry.LocateRegistry; import java.util.Scanner; public class RMIServer { public static void main(String[] args) { Scanner scanner = new Scanner(System.in); System.out.println("Elige un nodo (0-2):"); int nodo = scanner.nextInt(); switch(nodo){ case 0: System.out.println("Nodo 0"); try { int port = 1099; // Puerto por defecto String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; case 1: System.out.println("Nodo 1"); try { int port = 1234; // Puerto 1234 String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; case 2: System.out.println("Nodo 2"); try { int port = 5678; // Puerto 5678 String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; default: System.out.println("Nodo inválido"); } } }

## Para que el nodo 0 reciba las primeras 3 partes, el nodo 1 reciba las siguientes 3 y el nodo 2 reciba las últimas partes de la matriz A, se puede modificar el código de envío de la matriz A en el servidor de la siguiente manera:

## scss

## int numParts = 9;

## int partSize = N / numParts;

## for (int k = 0; k < numParts; k++) {

## float[][] A\_part = new float[partSize][M];

## int startRow = k \* partSize;

## for (int i = startRow; i < startRow + partSize; i++) {

## for (int j = 0; j < M; j++) {

## A\_part[i - startRow][j] = A[i][j];

## }

## }

## if (k < 3) {

## // Enviar las primeras 3 partes al nodo 0

## node0.sendMatrixA(A\_part);

## } else if (k < 6) {

## // Enviar las siguientes 3 partes al nodo 1

## node1.sendMatrixA(A\_part);

## } else {

## // Enviar las últimas partes al nodo 2

## node2.sendMatrixA(A\_part);

## }

## }

## En este código, se utiliza un contador k para iterar sobre las partes de la matriz A, y se utiliza una serie de condicionales para determinar a qué nodo se debe enviar cada parte. En este caso, las primeras 3 partes se envían al nodo 0, las siguientes 3 al nodo 1 y las últimas partes al nodo 2.

## Es importante tener en cuenta que en este código se asume que los objetos node0, node1 y node2 son instancias de la interfaz Interface que se utilizan para enviar la matriz A a cada nodo.

## puedes agregarlo import java.rmi.Naming; import java.rmi.registry.LocateRegistry; import java.util.Scanner; public class RMIServer { public static void main(String[] args) { Scanner scanner = new Scanner(System.in); System.out.println("Elige un nodo (0-2):"); int nodo = scanner.nextInt(); switch(nodo){ case 0: System.out.println("Nodo 0"); try { int port = 1099; // Puerto por defecto String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; case 1: System.out.println("Nodo 1"); try { int port = 1234; // Puerto 1234 String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; case 2: System.out.println("Nodo 2"); try { int port = 5678; // Puerto 5678 String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; default: System.out.println("Nodo inválido"); } } }

## Para que el Nodo 0 reciba los primeros 3 elementos, el Nodo 1 los siguientes 3 y el Nodo 2 los últimos, debemos realizar los siguientes cambios en el código:

## puedes agregarlo import java.rmi.Remote; import java.rmi.RemoteException; public interface Interface extends Remote { void sendMatrixA(float[][] A) throws RemoteException; }

## Claro, aquí te lo agrego:

## java

## import java.rmi.Remote;

## import java.rmi.RemoteException;

## public interface Interface extends Remote {

## void sendMatrixA(float[][] A) throws RemoteException;

## }

## Esta es una interfaz remota que define el método sendMatrixA, que toma una matriz de flotantes A como parámetro y lo lanza una excepción de RemoteException.

## hola necesito que el servidor RMIServer en el nodo 0 reciba los 3 primeros y el nodo 1 reciba los siguientes 3 y al final el nodo 2 ya que el RMIClient enviara todo esas partes las ultimas partes import java.rmi.Naming; import java.rmi.registry.LocateRegistry; import java.util.Scanner; public class RMIServer { public static void main(String[] args) { Scanner scanner = new Scanner(System.in); System.out.println("Elige un nodo (0-2):"); int nodo = scanner.nextInt(); switch(nodo){ case 0: System.out.println("Nodo 0"); try { int port = 1099; // Puerto por defecto String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; case 1: System.out.println("Nodo 1"); try { int port = 1234; // Puerto 1234 String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; case 2: System.out.println("Nodo 2"); try { int port = 5678; // Puerto 5678 String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; default: System.out.println("Nodo inválido"); } } }

## Para que el servidor RMIServer en el nodo 0 reciba los primeros 3 arrays y el nodo 1 reciba los siguientes 3 arrays, puedes modificar la implementación del método sendMatrixA en la clase PClass. A continuación, te muestro cómo hacerlo:

## En la clase PClass, define un atributo int node para almacenar el número del nodo.

## Modifica el constructor de PClass para que reciba el número del nodo como parámetro y lo almacene en el atributo node.

## En el método sendMatrixA, agrega una condición para comprobar si el atributo node es igual a 0 o 1 y, en consecuencia, almacenar los arrays recibidos en el objeto Multiplier correspondiente.

## Modifica el código en la clase RMIServer para que, al crear el objeto PClass, le pase como parámetro el número del nodo correspondiente.

## Aquí te muestro el código modificado de las clases PClass y RMIServer:

## java

## import java.rmi.RemoteException;

## import java.rmi.server.UnicastRemoteObject;

## public class PClass extends UnicastRemoteObject implements Interface {

## private static final long serialVersionUID = 1L;

## private float[][] A1, A2, A3;

## private int node;

## public PClass(int node) throws RemoteException {

## super();

## this.node = node;

## }

## @Override

## public void sendMatrixA(float[][] A) throws RemoteException {

## if (node == 0) {

## if (A1 == null) {

## A1 = A;

## System.out.println("Matriz A1 recibida");

## }

## else if (A2 == null) {

## A2 = A;

## System.out.println("Matriz A2 recibida");

## }

## else if (A3 == null) {

## A3 = A;

## System.out.println("Matriz A3 recibida");

## }

## }

## else if (node == 1) {

## if (A1 == null) {

## A1 = A;

## System.out.println("Matriz A1 recibida");

## }

## else if (A2 == null) {

## A2 = A;

## System.out.println("Matriz A2 recibida");

## }

## else if (A3 == null) {

## A3 = A;

## System.out.println("Matriz A3 recibida");

## }

## }

## }

## }

## java

## import java.rmi.Naming;

## import java.rmi.registry.LocateRegistry;

## import java.util.Scanner;

## public class RMIServer {

## public static void main(String[] args) {

## Scanner scanner = new Scanner(System.in);

## System.out.println("Elige un nodo (0-2):");

## int node = scanner.nextInt();

## switch(node){

## case 0:

## System.out.println("Nodo 0");

## try {

## int port = 1099; // Puerto por defecto

## String ipAddress = "localhost";

## LocateRegistry.createRegistry(port);

## PClass multiplier = new PClass(node);

## Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier);

## System.out.println("Servidor RMI listo en " + ipAddress + ":" + port);

## } catch (Exception e) {

## System.err.println("Error: " + e.getMessage());

## }

## break;

## case 1:

## System.out.println("Nodo 1");

## try {

## int port = 1234; //

## agrega en el PClass la funcion de mostrar la matriz A1, A2,A3

## Por supuesto, aquí está la implementación de la función mostrarMatricesA() en la clase PClass:

## java

## import java.rmi.RemoteException;

## import java.rmi.server.UnicastRemoteObject;

## public class PClass extends UnicastRemoteObject implements Interface {

## private float[][] A1;

## private float[][] A2;

## private float[][] A3;

## public PClass() throws RemoteException {

## super();

## }

## @Override

## public void sendMatrixA(float[][] A) throws RemoteException {

## int rows = A.length;

## int cols = A[0].length;

## if (A1 == null) {

## A1 = new float[rows][cols];

## for (int i = 0; i < rows; i++) {

## for (int j = 0; j < cols; j++) {

## A1[i][j] = A[i][j];

## }

## }

## System.out.println("Matriz A1 recibida:");

## mostrarMatriz(A1);

## } else if (A2 == null) {

## A2 = new float[rows][cols];

## for (int i = 0; i < rows; i++) {

## for (int j = 0; j < cols; j++) {

## A2[i][j] = A[i][j];

## }

## }

## System.out.println("Matriz A2 recibida:");

## mostrarMatriz(A2);

## } else if (A3 == null) {

## A3 = new float[rows][cols];

## for (int i = 0; i < rows; i++) {

## for (int j = 0; j < cols; j++) {

## A3[i][j] = A[i][j];

## }

## }

## System.out.println("Matriz A3 recibida:");

## mostrarMatriz(A3);

## }

## }

## private void mostrarMatriz(float[][] matriz) {

## int rows = matriz.length;

## int cols = matriz[0].length;

## for (int i = 0; i < rows; i++) {

## for (int j = 0; j < cols; j++) {

## System.out.print(matriz[i][j] + "\t");

## }

## System.out.println();

## }

## }

## }

## La función mostrarMatricesA() recibe la matriz que se va a mostrar, la recorre con dos ciclos for y la imprime en la consola. En la función sendMatrixA(), después de asignar cada matriz A recibida, se llama a la función mostrarMatriz() correspondiente para mostrarla en la consola.

## falta agregarlo a interfacce import java.rmi.Remote; import java.rmi.RemoteException; public interface Interface extends Remote { void sendMatrixA(float[][] A) throws RemoteException; }

## Ya se agregó previamente a la interfaz Interface en la respuesta anterior. Aquí está de nuevo para su referencia:

## java

## import java.rmi.Remote;

## import java.rmi.RemoteException;

## public interface Interface extends Remote {

## void sendMatrixA(float[][] A) throws RemoteException;

## }

## muestra la matriz A\_part por favor for (int k = 0; k < numParts; k++) { float[][] A\_part = new float[partSize][M]; int startRow = k \* partSize; for (int i = startRow; i < startRow + partSize; i++) { for (int j = 0; j < M; j++) { A\_part[i - startRow][j] = A[i][j]; } }

## 2 / 2

## Para mostrar la matriz A\_part, puedes agregar el siguiente código después de la inicialización de A\_part en el for:

## csharp

## System.out.println("A\_part " + k + ":");

## for (int i = 0; i < partSize; i++) {

## for (int j = 0; j < M; j++) {

## System.out.print(A\_part[i][j] + " ");

## }

## System.out.println();

## }

## Esto imprimirá la matriz A\_part para cada iteración del bucle for.

## necesito que las primeras 3 partes de la matriz se envien al nodo 0 y las siguientes 3 al nodo 1 y al ultimo las 3 ultimas partes al nodo 2 int numParts = 9; int partSize = N / numParts; for (int k = 0; k < numParts; k++) { float[][] A\_part = new float[partSize][M]; int startRow = k \* partSize; for (int i = startRow; i < startRow + partSize; i++) { for (int j = 0; j < M; j++) { A\_part[i - startRow][j] = A[i][j]; } } for (Interface multiplier : multipliers) { multiplier.sendMatrixA(A\_part); } }

## Para enviar las primeras 3 partes de la matriz al nodo 0, las siguientes 3 partes al nodo 1 y las últimas 3 partes al nodo 2, puedes modificar el bucle for de la siguiente manera:

## scss

## for (int k = 0; k < numParts; k++) {

## float[][] A\_part = new float[partSize][M];

## int startRow = k \* partSize;

## for (int i = startRow; i < startRow + partSize; i++) {

## for (int j = 0; j < M; j++) {

## A\_part[i - startRow][j] = A[i][j];

## }

## }

## if (k < 3) {

## multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0

## } else if (k < 6) {

## multipliers[1].sendMatrixA(A\_part); // enviar al nodo 1

## } else {

## multipliers[2].sendMatrixA(A\_part); // enviar al nodo 2

## }

## }

## En este código, se utiliza un condicional para determinar a qué nodo se debe enviar la parte de la matriz actual. Si k es menor que 3, se envía al nodo 0. Si k está entre 3 y 5 (inclusive), se envía al nodo 1. Si k es mayor o igual a 6, se envía al nodo 2.

## podrias quitarme los if porfavor if (A1 == null) { A1 = A; System.out.println("Matriz A1 recibida"); mostrarMatriz(A1); } else if (A2 == null) { A2 = A; System.out.println("Matriz A2 recibida"); mostrarMatriz(A2); } else if (A3 == null) { A3 = A; System.out.println("Matriz A3 recibida"); mostrarMatriz(A3); }

## Sí, puedes modificar el código de la función sendMatrixA en la clase PClass para quitar los if y simplemente guardar la matriz A en el arreglo correspondiente de acuerdo al índice recibido:

## csharp

## @Override

## public void sendMatrixA(float[][] A) throws RemoteException {

## if (A\_index < 0 || A\_index > 8) {

## System.out.println("Índice inválido para A: " + A\_index);

## return;

## }

## A\_arr[A\_index] = A;

## System.out.println("Matriz A" + (A\_index+1) + " recibida");

## mostrarMatriz(A);

## A\_index++;

## }

## En este código, el índice A\_index se asegura que esté en el rango válido de 0 a 8, y se utiliza para indexar el arreglo A\_arr en la posición correspondiente. Además, se llama a la función mostrarMatriz para imprimir la matriz recibida.

## agrega una funcion que cuando inicie borre el contenido de las matrices

## 2 / 2

## Para agregar una función que borre el contenido de las matrices al inicio, podemos agregar un método clearMatrices() en la clase PClass que establezca los valores de las matrices A1, A2 y A3 a null. Por ejemplo:

## java

## public class PClass extends UnicastRemoteObject implements Interface {

## private float[][] A1;

## private float[][] A2;

## private float[][] A3;

## 

## public PClass() throws RemoteException {

## super();

## }

## 

## @Override

## public void sendMatrixA(float[][] A) throws RemoteException {

## if (A1 == null) {

## A1 = A;

## System.out.println("Matriz A1 recibida");

## mostrarMatriz(A1);

## }

## else if (A2 == null) {

## A2 = A;

## System.out.println("Matriz A2 recibida");

## mostrarMatriz(A2);

## }

## else if (A3 == null) {

## A3 = A;

## System.out.println("Matriz A3 recibida");

## mostrarMatriz(A3);

## }

## }

## 

## public void clearMatrices() {

## A1 = null;

## A2 = null;

## A3 = null;

## }

## 

## // Métodos para mostrar las matrices

## }

## Luego, podemos llamar a este método clearMatrices() en el método main() de RMIServer antes de iniciar el servidor. Por ejemplo:

## csharp

## public class RMIServer {

## public static void main(String[] args) {

## Scanner scanner = new Scanner(System.in);

## System.out.println("Elige un nodo (0-2):");

## int nodo = scanner.nextInt();

## switch(nodo){

## case 0:

## System.out.println("Nodo 0");

## try {

## int port = 1099; // Puerto por defecto

## String ipAddress = "localhost";

## LocateRegistry.createRegistry(port);

## PClass multiplier = new PClass();

## multiplier.clearMatrices(); // Limpiar matrices

## Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier);

## System.out.println("Servidor RMI listo en " + ipAddress + ":" + port);

## } catch (Exception e) {

## System.err.println("Error: " + e.getMessage());

## }

## break;

## case 1:

## System.out.println("Nodo 1");

## try {

## int port = 1234; // Puerto 1234

## String ipAddress = "localhost";

## LocateRegistry.createRegistry(port);

## PClass multiplier = new PClass();

## multiplier.clearMatrices(); // Limpiar matrices

## Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier);

## System.out.println("Servidor RMI listo en " + ipAddress + ":" + port);

## } catch (Exception e) {

## System.err.println("Error: " + e.getMessage());

## }

## break;

## case 2:

## System.out.println("Nodo 2");

## try {

## int port = 5678; // Puerto 5678

## String ipAddress = "localhost";

## LocateRegistry.createRegistry(port);

## PClass multiplier = new PClass();

## multiplier.clearMatrices(); // Limpiar matrices

## Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier);

## System.out.println("Servidor RMI listo en " + ipAddress + ":" + port);

## } catch (Exception e) {

## System.err.println("Error: " + e.getMessage());

## }

## break;

## 

## como le puedo hacer para sobreescribir las matrices y no se queden fijas los valores import java.rmi.RemoteException; import java.rmi.server.UnicastRemoteObject; public class PClass extends UnicastRemoteObject implements Interface { private static final long serialVersionUID = 1L; private float[][] A1, A2, A3, A4, A5, A6, A7, A8, A9; private int node; public PClass(int node) throws RemoteException { super(); this.node = node; } @Override public void sendMatrixA(float[][] A) throws RemoteException { if (node == 0) { if (A1 == null) { A1 = A; System.out.println("Matriz A1 recibida"); mostrarMatriz(A1); } else if (A2 == null) { A2 = A; System.out.println("Matriz A2 recibida"); mostrarMatriz(A2); } else if (A3 == null) { A3 = A; System.out.println("Matriz A3 recibida"); mostrarMatriz(A3); } } else if (node == 1) { if (A4 == null) { A4 = A; System.out.println("Matriz A4 recibida"); mostrarMatriz(A4); } else if (A5 == null) { A5 = A; System.out.println("Matriz A5 recibida"); mostrarMatriz(A5); } else if (A6 == null) { A6 = A; System.out.println("Matriz A6 recibida"); mostrarMatriz(A6); } } else if (node == 2) { if (A7 == null) { A7 = A; System.out.println("Matriz A7 recibida"); mostrarMatriz(A7); } else if (A8 == null) { A8 = A; System.out.println("Matriz A8 recibida"); mostrarMatriz(A8); } else if (A9 == null) { A9 = A; System.out.println("Matriz A9 recibida"); mostrarMatriz(A9); } } } private void mostrarMatriz(float[][] matriz) { int rows = matriz.length; int cols = matriz[0].length; for (int i = 0; i < rows; i++) { for (int j = 0; j < cols; j++) { System.out.print(matriz[i][j] + "\t"); } System.out.println(); } } }

## Puedes agregar una función que se encargue de limpiar el contenido de las matrices al inicio del programa. Podrías hacerlo de la siguiente manera:

## java

## public class PClass extends UnicastRemoteObject implements Interface {

## private static final long serialVersionUID = 1L;

## private float[][] A1, A2, A3, A4, A5, A6, A7, A8, A9;

## private int node;

## public PClass(int node) throws RemoteException {

## super();

## this.node = node;

## limpiarMatrices();

## }

## // ...

## private void limpiarMatrices() {

## A1 = null;

## A2 = null;

## A3 = null;

## A4 = null;

## A5 = null;

## A6 = null;

## A7 = null;

## A8 = null;

## A9 = null;

## }

## }esta bien la transpuesta de B\_T int N = 9; int M = 4; float[][] A = new float[N][M]; float[][] B = new float[M][N]; float[][] B\_T= new float[N][M]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } for (int i = 0; i < M; i++) { for (int j = 0; j < N; j++) { B\_T[j][i] = B[i][j]; } }

## Sí, la transpuesta de B\_T parece estar correctamente calculada en el código que proporcionaste.

## La transpuesta de una matriz se obtiene intercambiando sus filas por columnas. En el código que has compartido, el bucle for anidado utilizado para calcular la transpuesta de B intercambia correctamente los índices i y j, lo que da como resultado la transpuesta de B almacenada en la matriz B\_T.

## Es importante tener en cuenta que la transpuesta de una matriz solo está definida para matrices cuadradas o rectangulares, es decir, para matrices que tienen el mismo número de filas y columnas o que tienen un número diferente de filas y columnas, respectivamente. En este caso, dado que la matriz B es de tamaño M x N y la matriz B\_T es de tamaño N x M, la transposición de la matriz B es posible y la matriz B\_T es la transpuesta correcta de B.

## inicializa la matriz B [N][M]int N = 9; int M = 4; float[][] A = new float[N][M]; float[][] B = new float[N][M]; float[][] B\_T= new float[N][M]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { B[j][i] = 3 \* i - 2 \* j; } }

## En el código que proporcionaste, la matriz B ya está siendo inicializada en el primer bucle for anidado que recorre los índices i y j de la matriz A y B.

## Dentro de este bucle, se está asignando un valor específico a cada elemento de la matriz B utilizando la fórmula B[j][i] = 3 \* i - 2 \* j.

## Si deseas inicializar la matriz B con un valor específico antes de utilizarla en este bucle, puedes agregar otro bucle for anidado para asignar el valor deseado a cada elemento de la matriz B antes de que se utilice en el primer bucle for.

## Por ejemplo, si deseas inicializar la matriz B con el valor 1, puedes hacer lo siguiente:

## css

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < M; j++) {

## B[i][j] = 1;

## }

## }

## Este bucle recorre cada índice i y j de la matriz B y asigna el valor 1 a cada elemento. Luego, el primer bucle for anidado puede asignar valores diferentes a cada elemento de la matriz B utilizando la fórmula correspondiente.

## en java inicializa 2 matrices A y B que sean de N=9 y M=4 y A inicie en A[i][j] = 2 \* i + 3 \* j y B en B[j][i] = 3 \* i - 2 \* j;

## Por supuesto, aquí te proporciono un ejemplo de código en Java que inicializa las matrices A y B con los valores que mencionas:

## css

## int N = 9;

## int M = 4;

## float[][] A = new float[N][M];

## float[][] B = new float[N][M];

## // Inicializar la matriz A

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < M; j++) {

## A[i][j] = 2 \* i + 3 \* j;

## }

## }

## // Inicializar la matriz B

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < M; j++) {

## B[j][i] = 3 \* i - 2 \* j;

## }

## }

## En el primer bucle for anidado, se inicializa la matriz A utilizando la fórmula A[i][j] = 2 \* i + 3 \* j. En el segundo bucle for anidado, se inicializa la matriz B utilizando la fórmula B[j][i] = 3 \* i - 2 \* j.

## Es importante tener en cuenta que en la matriz B, los índices i y j se intercambian en comparación con la matriz A. Esto se debe a que la matriz B está transpuesta con respecto a la matriz A.

## en una matriz N=9 y M=4 por una de N=4 M=9 de que tamaño sera la matriz resultante

## Si multiplicamos una matriz de tamaño N=9 y M=4 por otra de tamaño N=4 y M=9, la matriz resultante tendrá un tamaño de N=9 y M=9.

## La regla para la multiplicación de matrices establece que para multiplicar dos matrices, el número de columnas de la primera matriz debe ser igual al número de filas de la segunda matriz. En este caso, la matriz de tamaño N=4 y M=9 tiene 4 filas y 9 columnas, y la matriz de tamaño N=9 y M=4 tiene 9 filas y 4 columnas. Por lo tanto, podemos multiplicar estas matrices, y la matriz resultante tendrá un tamaño de 9 filas y 9 columnas.

## y una de 9 filas y 4 columnas por una de 4 filas y 9 columnas

## Si multiplicamos una matriz de tamaño N=9 y M=4 por otra de tamaño N=4 y M=9, la matriz resultante tendrá un tamaño de N=9 y M=9.

## Sin embargo, si multiplicamos una matriz de tamaño N=9 y M=4 por otra de tamaño N=4 y M=9, la matriz resultante no se puede calcular ya que el número de columnas de la primera matriz (M=4) no es igual al número de filas de la segunda matriz (N=4). Por lo tanto, la operación de multiplicación de matrices no se puede realizar en este caso.

## ahora necesito dividir la matriz B en 9 partes iguales en este codigo int N = 9; int M = 4; float[][] A = new float[N][M]; float[][] B = new float[M][N]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } int numParts = 9; int partSize = N / numParts; for (int k = 0; k < numParts; k++) { float[][] A\_part = new float[partSize][M]; float[][] B\_part = new float[M][partSize]; int startRow = k \* partSize; for (int i = startRow; i < startRow + partSize; i++) { for (int j = 0; j < M; j++) { A\_part[i - startRow][j] = A[i][j]; } } if (k < 3) { multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0 } else if (k < 6) { multipliers[1].sendMatrixA(A\_part); // enviar al nodo 1 } else { multipliers[2].sendMatrixA(A\_part); // enviar al nodo 2 } }

## Para dividir la matriz B en 9 partes iguales, puedes seguir el mismo enfoque que se usó para dividir la matriz A.

## Aquí está el código modificado para dividir la matriz B en 9 partes iguales:

## scss

## int N = 9;

## int M = 4;

## float[][] A = new float[N][M];

## float[][] B = new float[M][N];

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < M; j++) {

## A[i][j] = 2 \* i + 3 \* j;

## B[j][i] = 3 \* i - 2 \* j;

## }

## }

## int numParts = 9;

## int partSize = N / numParts;

## for (int k = 0; k < numParts; k++) {

## float[][] A\_part = new float[partSize][M];

## float[][] B\_part = new float[M][partSize];

## int startRow = k \* partSize;

## for (int i = startRow; i < startRow + partSize; i++) {

## for (int j = 0; j < M; j++) {

## A\_part[i - startRow][j] = A[i][j];

## B\_part[j][i - startRow] = B[j][i];

## }

## }

## // enviar las partes de A y B a los nodos correspondientes

## if (k < 3) {

## multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0

## multipliers[0].sendMatrixB(B\_part);

## } else if (k < 6) {

## multipliers[1].sendMatrixA(A\_part); // enviar al nodo 1

## multipliers[1].sendMatrixB(B\_part);

## } else {

## multipliers[2].sendMatrixA(A\_part); // enviar al nodo 2

## multipliers[2].sendMatrixB(B\_part);

## }

## }

## Este código divide la matriz B en 9 partes iguales y las envía a los nodos correspondientes. Cada parte de la matriz B se almacena en una matriz separada B\_part. La parte B\_part correspondiente a la submatriz A\_part también se envía al mismo nodo.

## Tenga en cuenta que el número de partes y el tamaño de cada parte se mantienen iguales que en el caso anterior.

## De esta manera, las matrices A1, A2, A3, A4, A5, A6, A7, A8 y A9 serán nulas al inicio del programa, y no tendrán valores fijos.

## lo podra realizar tambien el RMIClient

## Sí, el RMIClient también puede tener una función para borrar el contenido de las matrices. Una forma de hacerlo es añadir un método borrarMatrices() en la interfaz Interface y luego implementarlo tanto en el RMIServer como en el RMIClient.

## Por ejemplo, en la interfaz Interface, añadiríamos el siguiente método:

## java

## void borrarMatrices() throws RemoteException;

## Luego, en la implementación del PClass en el RMIServer, agregaríamos lo siguiente:

## java

## public void borrarMatrices() throws RemoteException {

## A1 = null;

## A2 = null;

## A3 = null;

## A4 = null;

## A5 = null;

## A6 = null;

## A7 = null;

## A8 = null;

## A9 = null;

## System.out.println("Matrices borradas");

## }

## Finalmente, en la implementación del RMIClient, agregaríamos lo siguiente:

## java

## public void borrarMatrices() throws RemoteException {

## Interface server = (Interface) Naming.lookup("//localhost/RMIServer");

## server.borrarMatrices();

## System.out.println("Matrices borradas");

## }

## De esta manera, al llamar al método borrarMatrices() en el RMIClient, se enviará una solicitud al RMIServer para que borre el contenido de las matrices en su implementación del PClass, y ambos tendrán las matrices limpias.

## agregalo a mi RMIClient porfavor import java.rmi.Naming; import java.util.Scanner; public class RMIClient { private static final String[] SERVERS = {"localhost", "localhost", "localhost"}; private static final int []PORT = {1099,1234,5678}; private static final String SERVICE\_NAME = "MatrixMultiplier"; public static void main(String[] args) { Scanner scanner = new Scanner(System.in); try { Interface[] multipliers = new Interface[SERVERS.length]; for (int i = 0; i < SERVERS.length; i++) { multipliers[i] = (Interface) Naming.lookup(String.format("rmi://%s:%d/%s", SERVERS[i], PORT[i], SERVICE\_NAME)); } System.out.println("Ingrese la opcion 1 o 2"); int opcion = scanner.nextInt(); if (opcion == 1) { try { int N = 9; int M = 4; float[][] A = new float[N][M]; float[][] B = new float[M][N]; float[][] B\_T= new float[N][M]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } for (int i = 0; i < M; i++) { for (int j = 0; j < N; j++) { B\_T[j][i] = B[i][j]; } } int numParts = 9; int partSize = N / numParts; for (int k = 0; k < numParts; k++) { float[][] A\_part = new float[partSize][M]; int startRow = k \* partSize; for (int i = startRow; i < startRow + partSize; i++) { for (int j = 0; j < M; j++) { A\_part[i - startRow][j] = A[i][j]; } } if (k < 3) { multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0 } else if (k < 6) { multipliers[1].sendMatrixA(A\_part); // enviar al nodo 1 } else { multipliers[2].sendMatrixA(A\_part); // enviar al nodo 2 } } System.out.println("Matriz A enviada al servidor."); for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { System.out.print(A[i][j] + " "); } System.out.println(); } } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } }else if (opcion == 2) { int N = 900; int M = 400; System.out.println("PROXIMAMENTE."); } else { System.out.println("Opcion invalida."); } } catch (Exception e) { System.err.println("Error: " + e.getMessage()); e.printStackTrace(); } } }

## 2 / 2

## El código que proporcionaste parece estar incompleto. En particular, falta la definición de la interfaz remota "Interface" que se utiliza en el cliente. Por lo tanto, no puedo evaluar si el código es funcional o no. Pero en términos generales, si el código es correcto, entonces el RMIClient debería poder realizar la comunicación remota con el servidor RMI y enviarle la matriz A correctamente.

## pregunta si monto este codigo en azure junto con su cliente se conectara correctamente import java.rmi.Naming; import java.rmi.registry.LocateRegistry; import java.util.Scanner; public class RMIServer { public static void main(String[] args) { Scanner scanner = new Scanner(System.in); System.out.println("Elige un nodo (0-2):"); int node = scanner.nextInt(); switch(node){ case 0: System.out.println("Nodo 0"); try { int port = 1099; // Puerto por defecto String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(node); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; case 1: System.out.println("Nodo 1"); try { int port = 1234; // Puerto 1234 String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(node); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; case 2: System.out.println("Nodo 2"); try { int port = 5678; // Puerto 5678 String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(node); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; default: System.out.println("Nodo inválido"); } } }

## Este código es un servidor RMI (Remote Method Invocation) en Java que utiliza el registro RMI para enlazar objetos remotos y permitir que los clientes accedan a ellos de forma remota. En cuanto a si se conectará correctamente en Azure, eso dependerá de cómo configure su servidor y cómo se comuniquen sus clientes con el servidor.

## Para que este servidor RMI funcione en Azure, deberá seguir algunos pasos, como crear una máquina virtual en Azure, instalar Java y configurar las reglas de seguridad de la red para permitir que los clientes se conecten al servidor.

## Además, para que los clientes se conecten correctamente al servidor, deberá asegurarse de que el código del cliente tenga la dirección IP correcta del servidor y el número de puerto en el que está escuchando el servidor.

## En resumen, sí es posible ejecutar este código en Azure siempre y cuando realice la configuración y los ajustes necesarios en el entorno de Azure y en el código del cliente para que se comuniquen correctamente con el servidor.

## ahora multiplica la matrice A y B en una matriz C try { int N = 9; int M = 4; float[][] A = new float[N][M]; float[][] B = new float[M][N]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } int numParts = 9; int partSize = N / numParts; for (int k = 0; k < numParts; k++) { float[][] A\_part = new float[partSize][M]; float[][] B\_part = new float[M][partSize]; int startRow = k \* partSize; for (int i = startRow; i < startRow + partSize; i++) { for (int j = 0; j < M; j++) { A\_part[i - startRow][j] = A[i][j]; B\_part[j][i - startRow] = B[j][i]; } } if (k < 3) { multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0 multipliers[0].sendMatrixB(B\_part); } else if (k < 6) { multipliers[1].sendMatrixA(A\_part); // enviar al nodo 1 multipliers[1].sendMatrixB(B\_part); } else { multipliers[2].sendMatrixA(A\_part); // enviar al nodo 2 multipliers[2].sendMatrixB(B\_part); } } System.out.println("Matriz A enviada al servidor."); for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { System.out.print(A[i][j] + " "); } System.out.println(); } System.out.println("Matriz B enviada al servidor."); for (int i = 0; i < M; i++) { for (int j = 0; j < N; j++) { System.out.print(B[i][j] + " "); } System.out.println(); } } catch (Exception e) { System.err.println("Error: " + e.getMessage()); }

## Para multiplicar las matrices A y B en una matriz C, primero necesitarás recibir las matrices A y B en los nodos del servidor utilizando la clase PClass que se utiliza para enviar y recibir matrices a través de RMI. Luego, deberás modificar la clase PClass para realizar la multiplicación de matrices. Finalmente, tendrás que recuperar la matriz C resultante del servidor y mostrarla en la salida.

## Aquí hay un ejemplo de cómo podrías realizar la multiplicación de matrices utilizando la clase PClass:

## java

## public class PClass extends UnicastRemoteObject implements PInterface {

## private static final long serialVersionUID = 1L;

## private int node;

## public PClass(int node) throws RemoteException {

## super();

## this.node = node;

## }

## private float[][] matrixA;

## private float[][] matrixB;

## public void sendMatrixA(float[][] A) throws RemoteException {

## matrixA = A;

## }

## public void sendMatrixB(float[][] B) throws RemoteException {

## matrixB = B;

## }

## public float[][] multiply() throws RemoteException {

## int m = matrixA.length;

## int n = matrixB[0].length;

## int o = matrixB.length;

## float[][] result = new float[m][n];

## for (int i = 0; i < m; i++) {

## for (int j = 0; j < n; j++) {

## for (int k = 0; k < o; k++) {

## result[i][j] += matrixA[i][k] \* matrixB[k][j];

## }

## }

## }

## return result;

## }

## }

## Luego, para recuperar la matriz C resultante, deberás agregar el siguiente código al cliente:

## scss

## // Recuperar matriz C

## float[][] C = new float[N][N];

## for (int k = 0; k < numParts; k++) {

## float[][] C\_part = multipliers[k].multiply();

## int startRow = k \* partSize;

## for (int i = startRow; i < startRow + partSize; i++) {

## for (int j = 0; j < N; j++) {

## C[i][j] = C\_part[i - startRow][j];

## }

## }

## }

## // Mostrar matriz C

## System.out.println("Matriz C resultante:");

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < N; j++) {

## System.out.print(C[i][j] + " ");

## }

## System.out.println();

## }

## Este código recuperará cada parte de la matriz C de cada nodo del servidor y la reconstruirá en la matriz C completa. Luego, se mostrará la matriz C completa en la salida.

## la matriz C tiene que estar dividida en 81 partes

## Para dividir la matriz C en 81 partes, deberás primero definir la dimensión de la matriz C. La matriz resultante de la multiplicación de A y B tendrá dimensiones NxN, donde N es el número de filas de A y el número de columnas de B. En este caso, N es igual a 9.

## Luego, debes definir el número de partes en que deseas dividir la matriz C. En este caso, el número es 81. Para dividir la matriz C en 81 partes, debes dividir tanto las filas como las columnas de C en partes iguales.

## Puedes utilizar un par de bucles anidados para recorrer la matriz C y dividirla en partes. El tamaño de cada parte será igual a N/9, ya que la matriz está dividida en 9 filas y 9 columnas.

## Aquí te dejo un ejemplo de cómo podrías dividir la matriz C en 81 partes:

## scss

## int partSize = N / 9;

## float[][] C = new float[N][N];

## for (int i = 0; i < N; i += partSize) {

## for (int j = 0; j < N; j += partSize) {

## // Definir la parte actual de la matriz C

## float[][] C\_part = new float[partSize][partSize];

## 

## // Calcular los valores de la parte actual de la matriz C

## for (int k = i; k < i + partSize; k++) {

## for (int l = j; l < j + partSize; l++) {

## for (int m = 0; m < M; m++) {

## C\_part[k-i][l-j] += A[k][m] \* B[m][l];

## }

## }

## }

## 

## // Guardar la parte actual de la matriz C

## for (int k = i; k < i + partSize; k++) {

## for (int l = j; l < j + partSize; l++) {

## C[k][l] = C\_part[k-i][l-j];

## }

## }

## }

## }

## Este código recorre la matriz C en pasos de partSize, que es igual a N/9. En cada paso, define la parte actual de la matriz C, calcula sus valores utilizando la multiplicación de matrices, y guarda los valores calculados en la matriz C original. Al final del proceso, la matriz C estará dividida en 81 partes iguales.

## logra que el nodo 0 tenga las partes del 1 al 27 de C y asi consecutivamente import java.rmi.Naming; import java.rmi.registry.LocateRegistry; import java.util.Scanner; public class RMIServer { public static void main(String[] args) { Scanner scanner = new Scanner(System.in); System.out.println("Elige un nodo (0-2):"); int node = scanner.nextInt(); switch(node){ case 0: System.out.println("Nodo 0"); try { int port = 1099; // Puerto por defecto String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(node); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; case 1: System.out.println("Nodo 1"); try { int port = 1234; // Puerto 1234 String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(node); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; case 2: System.out.println("Nodo 2"); try { int port = 5678; // Puerto 5678 String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(node); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break; default: System.out.println("Nodo inválido"); } } }

## Para lograr que el nodo 0 tenga las partes del 1 al 27 de la matriz C, se puede modificar la sección de envío de datos en el código del cliente de la siguiente manera:

## scss

## for (int k = 0; k < numParts; k++) {

## // ...

## if (k < 27) {

## multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0

## multipliers[0].sendMatrixB(B\_part);

## } else if (k < 54) {

## multipliers[1].sendMatrixA(A\_part); // enviar al nodo 1

## multipliers[1].sendMatrixB(B\_part);

## } else {

## multipliers[2].sendMatrixA(A\_part); // enviar al nodo 2

## multipliers[2].sendMatrixB(B\_part);

## }

## }

## Aquí, se modificó la condición del primer if para que el nodo 0 reciba las partes del 0 al 26 (en lugar de del 0 al 2), y se agregó una nueva condición para que el nodo 1 reciba las partes del 27 al 53, y el nodo 2 reciba las partes del 54 al 80.

## Es importante tener en cuenta que, para que esta modificación funcione correctamente, la variable numParts debe tener un valor de 81.

## con lo recibido de la matriz A y B en el nodo 0, 1 y 2 forma la matriz C resultante entre la multiplicacion de la amtriz A y B y la amtriz C tiene que estar dividida en 27 partes iguales en cada nodo

## Para formar la matriz C resultante de la multiplicación de las matrices A y B en los nodos 0, 1 y 2, podemos hacer lo siguiente:

## En cada

## con lo recibido de la matriz A y B en el nodo 0, 1 y 2 forma la matriz C resultante entre la multiplicacion de la amtriz A y B y la amtriz C tiene que estar dividida en 27 partes iguales en cada nodo import java.rmi.RemoteException; import java.rmi.server.UnicastRemoteObject; public class PClass extends UnicastRemoteObject implements Interface { private static final long serialVersionUID = 1L; private float[][] A1, A2, A3, A4, A5, A6, A7, A8, A9; private float[][] B1, B2, B3, B4, B5, B6, B7, B8, B9; private int node; public PClass(int node) throws RemoteException { super(); this.node = node; } public void borrarMatrices() throws RemoteException { A1 = null; A2 = null; A3 = null; A4 = null; A5 = null; A6 = null; A7 = null; A8 = null; A9 = null; B1 = null; B2 = null; B3 = null; B4 = null; B5 = null; B6 = null; B7 = null; B8 = null; B9 = null; System.out.println("Matrices borradas"); } @Override public void sendMatrixA(float[][] A) throws RemoteException { if (node == 0) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; System.out.println("Matriz A1, A2, A3 recibida"); } } else if (node == 1) { if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; System.out.println("Matriz A4, A5, A6 recibida"); } } else if (node == 2) { if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; System.out.println("Matriz A7, A8, A9 recibida"); } } } @Override public void sendMatrixB(float[][] B) throws RemoteException { if (node == 0) { if (B1 == null) { B1 = B; } else if (B2 == null) { B2 = B; } else if (B3 == null) { B3 = B; System.out.println("Matriz B1, B2, B3 recibida"); } } else if (node == 1) { if (B4 == null) { B4 = B; } else if (B5 == null) { B5 = B; } else if (B6 == null) { B6 = B; System.out.println("Matriz B4, B5, B6 recibida"); } } else if (node == 2) { if (B7 == null) { B7 = B; } else if (B8 == null) { B8 = B; } else if (B9 == null) { B9 = B; System.out.println("Matriz B7, B8, B9 recibida"); } } } private void mostrarMatriz(float[][] matriz) { int rows = matriz.length; int cols = matriz[0].length; for (int i = 0; i < rows; i++) { for (int j = 0; j < cols; j++) { System.out.print(matriz[i][j] + "\t"); } System.out.println(); } } }

## 2 / 2

## Este código Java implementa la interfaz remota Interface y define la clase PClass, que representa un nodo del sistema distribuido. La clase contiene métodos para recibir las matrices A y B divididas en partes en los nodos 0, 1 y 2, y para mostrar las matrices.

## Los métodos sendMatrixA y sendMatrixB se encargan de recibir las partes de las matrices A y B respectivamente. En el nodo 0, la matriz A se divide en A1, A2 y A3, y la matriz B se divide en B1, B2 y B3. En el nodo 1, la matriz A se divide en A4, A5 y A6, y la matriz B se divide en B4, B5 y B6. En el nodo 2, la matriz A se divide en A7, A8 y A9, y la matriz B se divide en B7, B8 y B9.

## El método borrarMatrices se utiliza para liberar la memoria de las matrices A y B en el nodo actual.

## El método mostrarMatriz se utiliza para imprimir una matriz en la consola.

## Es importante destacar que este código solo recibe las partes de las matrices A y B, pero no realiza la multiplicación de matrices ni la división de la matriz resultante en partes iguales entre los nodos. Esa funcionalidad debe ser implementada en otro código.

## quiero que la multilpicacion la realice el RMIServer

## Para realizar la multiplicación de las matrices en el RMIServer, puedes crear un método en la clase RMIServer que reciba las matrices A y B, realice la multiplicación y devuelva la matriz C resultante. Por ejemplo:

## java

## public class RMIServer {

## // ...

## public float[][] multiplyMatrices(float[][] A, float[][] B) throws RemoteException {

## // Verificar que las matrices son compatibles para multiplicación

## int rowsA = A.length;

## int colsA = A[0].length;

## int rowsB = B.length;

## int colsB = B[0].length;

## if (colsA != rowsB) {

## throw new IllegalArgumentException("Las matrices no son compatibles para multiplicación");

## }

## // Realizar la multiplicación de las matrices

## float[][] C = new float[rowsA][colsB];

## for (int i = 0; i < rowsA; i++) {

## for (int j = 0; j < colsB; j++) {

## float sum = 0;

## for (int k = 0; k < colsA; k++) {

## sum += A[i][k] \* B[k][j];

## }

## C[i][j] = sum;

## }

## }

## return C;

## }

## }

## Luego, en la implementación de la interfaz remota en la clase PClass, puedes llamar a este método en el RMIServer para obtener la matriz C resultante y dividirla en 27 partes iguales. Por ejemplo:

## java

## public class PClass extends UnicastRemoteObject implements Interface {

## // ...

## private RMIServer server;

## public PClass(int node, RMIServer server) throws RemoteException {

## super();

## this.node = node;

## this.server = server;

## }

## @Override

## public void sendMatrixA(float[][] A) throws RemoteException {

## // ...

## }

## @Override

## public void sendMatrixB(float[][] B) throws RemoteException {

## // ...

## if (B9 != null) {

## // Realizar la multiplicación de las matrices A y B en el servidor

## float[][] C = server.multiplyMatrices(combineMatrices(A1, A2, A3, A4, A5, A6, A7, A8, A9),

## combineMatrices(B1, B2, B3, B4, B5, B6, B7, B8, B9));

## // Dividir la matriz C en 27 partes iguales

## float[][][] dividedC = divideMatrix(C, 27);

## // Enviar cada parte de la matriz C al nodo correspondiente

## for (int i = 0; i < 27; i++) {

## Interface p = (Interface) Naming.lookup("//localhost/P" + i);

## p.receiveMatrixC(dividedC[i]);

## }

## // Borrar las matrices recibidas

## borrarMatrices();

## }

## }

## // ...

## }

## En este ejemplo, el método combineMatrices combina las nueve submatrices A y B en una sola matriz, y el método divideMatrix divide la matriz C resultante en 27 partes iguales. Estos métodos se pueden implementar de diversas maneras según tus necesidades específicas.

## del siguiente RMIClient quiero que realice la multiplicacion de A\_part \* B\_part dentro de una matriz C que debe estar dividida en 81 partes

## Para realizar la multiplicación de A\_part \* B\_part dentro de una matriz C dividida en 81 partes, se debe hacer lo siguiente:

## En el código del RMIClient, agregar un método que calcule la multiplicación de dos matrices. Este método debe tomar como parámetros las matrices A y B, y devolver la matriz resultante C.

## Una

## del siguiente RMIClient quiero que realice la multiplicacion de A\_part \* B\_part dentro de una matriz C que debe estar dividida en 81 partes import java.rmi.Naming; import java.util.Scanner; public class RMIClient { private static final String[] SERVERS = {"localhost", "localhost", "localhost"}; private static final int []PORT = {1099,1234,5678}; private static final String SERVICE\_NAME = "MatrixMultiplier"; public static void main(String[] args) { Scanner scanner = new Scanner(System.in); try { Interface[] multipliers = new Interface[SERVERS.length]; for (int i = 0; i < SERVERS.length; i++) { multipliers[i] = (Interface) Naming.lookup(String.format("rmi://%s:%d/%s", SERVERS[i], PORT[i], SERVICE\_NAME)); } multipliers[0].borrarMatrices(); multipliers[1].borrarMatrices(); multipliers[2].borrarMatrices(); System.out.println("Ingrese la opcion 1 o 2"); int opcion = scanner.nextInt(); if (opcion == 1) { try { int N = 9; int M = 4; float[][] A = new float[N][M]; float[][] B = new float[M][N]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } int numParts = 9; int partSize = N / numParts; for (int k = 0; k < numParts; k++) { float[][] A\_part = new float[partSize][M]; float[][] B\_part = new float[M][partSize]; int startRow = k \* partSize; for (int i = startRow; i < startRow + partSize; i++) { for (int j = 0; j < M; j++) { A\_part[i - startRow][j] = A[i][j]; B\_part[j][i - startRow] = B[j][i]; } } if (k < 3) { multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0 multipliers[0].sendMatrixB(B\_part); } else if (k < 6) { multipliers[1].sendMatrixA(A\_part); // enviar al nodo 1 multipliers[1].sendMatrixB(B\_part); } else { multipliers[2].sendMatrixA(A\_part); // enviar al nodo 2 multipliers[2].sendMatrixB(B\_part); } } int m = A.length; int n = B[0].length; int o = B.length; float[][] result = new float[m][n]; for (int i = 0; i < m; i++) { for (int j = 0; j < n; j++) { for (int k = 0; k < o; k++) { result[i][j] += A[i][k] \* B[k][j]; } } } // Mostrar matriz C System.out.println("Matriz C resultante:"); for (int i = 0; i < N; i++) { for (int j = 0; j < N; j++) { System.out.print(result[i][j] + " "); } System.out.println(); } } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } }else if (opcion == 2) { int N = 900; int M = 400; System.out.println("PROXIMAMENTE."); } else { System.out.println("Opcion invalida."); } } catch (Exception e) { System.err.println("Error: " + e.getMessage()); e.printStackTrace(); } } }

## Para realizar la multiplicación de A\_part \* B\_part dentro de una matriz C que debe estar dividida en 81 partes en el código proporcionado, debes hacer lo siguiente:

## Elimina la sección de código que realiza la multiplicación utilizando bucles for, ya que no se utilizará en este caso.

## Agrega una nueva sección de código después del ciclo for que envía las matrices A y B a los diferentes nodos. Dentro de esta sección, deberás recuperar las matrices C\_part de cada nodo, combinarlas en una única matriz C y mostrarla en la salida.

## Aquí está el código modificado:

## scss

## import java.rmi.Naming;

## import java.util.Scanner;

## public class RMIClient {

## private static final String[] SERVERS = {"localhost", "localhost", "localhost"};

## private static final int []PORT = {1099,1234,5678};

## private static final String SERVICE\_NAME = "MatrixMultiplier";

## public static void main(String[] args) {

## Scanner scanner = new Scanner(System.in);

## try {

## Interface[] multipliers = new Interface[SERVERS.length];

## for (int i = 0; i < SERVERS.length; i++) {

## multipliers[i] = (Interface) Naming.lookup(String.format("rmi://%s:%d/%s", SERVERS[i], PORT[i], SERVICE\_NAME));

## }

## multipliers[0].borrarMatrices();

## multipliers[1].borrarMatrices();

## multipliers[2].borrarMatrices();

## System.out.println("Ingrese la opcion 1 o 2");

## int opcion = scanner.nextInt();

## if (opcion == 1) {

## try {

## int N = 9;

## int M = 4;

## float[][] A = new float[N][M];

## float[][] B = new float[M][N];

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < M; j++) {

## A[i][j] = 2 \* i + 3 \* j;

## B[j][i] = 3 \* i - 2 \* j;

## }

## }

## int numParts = 9;

## int partSize = N / numParts;

## for (int k = 0; k < numParts; k++) {

## float[][] A\_part = new float[partSize][M];

## float[][] B\_part = new float[M][partSize];

## int startRow = k \* partSize;

## for (int i = startRow; i < startRow + partSize; i++) {

## for (int j = 0; j < M; j++) {

## A\_part[i - startRow][j] = A[i][j];

## B\_part[j][i - startRow] = B[j][i];

## }

## }

## if (k < 3) {

## multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0

## multipliers[0].sendMatrixB(B\_part);

## } else if (k < 6) {

## multipliers[1].sendMatrixA(A\_part); // enviar al nodo 1

## multipliers[1].sendMatrixB(B\_part);

## } else {

## multipliers[2].sendMatrixA(A\_part); // enviar al nodo 2

## multipliers[2].sendMatrixB(B\_part);

## }

## }

## // Recuperar las matrices C\_part de cada nodo y combinarlas en una única matriz C

## int C\_SIZE = N;

## float[][] C = new float[C\_SIZE][C

## porfavor continua con el codigo import java.rmi.Naming; import java.util.Scanner; public class RMIClient { private static final String[] SERVERS = {"localhost", "localhost", "localhost"}; private static final int []PORT = {1099,1234,5678}; private static final String SERVICE\_NAME = "MatrixMultiplier"; public static void main(String[] args) { Scanner scanner = new Scanner(System.in); try { Interface[] multipliers = new Interface[SERVERS.length]; for (int i = 0; i < SERVERS.length; i++) { multipliers[i] = (Interface) Naming.lookup(String.format("rmi://%s:%d/%s", SERVERS[i], PORT[i], SERVICE\_NAME)); } multipliers[0].borrarMatrices(); multipliers[1].borrarMatrices(); multipliers[2].borrarMatrices(); System.out.println("Ingrese la opcion 1 o 2"); int opcion = scanner.nextInt(); if (opcion == 1) { try { int N = 9; int M = 4; float[][] A = new float[N][M]; float[][] B = new float[M][N]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } int numParts = 9; int partSize = N / numParts; for (int k = 0; k < numParts; k++) { float[][] A\_part = new float[partSize][M]; float[][] B\_part = new float[M][partSize]; int startRow = k \* partSize; for (int i = startRow; i < startRow + partSize; i++) { for (int j = 0; j < M; j++) { A\_part[i - startRow][j] = A[i][j]; B\_part[j][i - startRow] = B[j][i]; } } if (k < 3) { multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0 multipliers[0].sendMatrixB(B\_part); } else if (k < 6) { multipliers[1].sendMatrixA(A\_part); // enviar al nodo 1 multipliers[1].sendMatrixB(B\_part); } else { multipliers[2].sendMatrixA(A\_part); // enviar al nodo 2 multipliers[2].sendMatrixB(B\_part); } } // Recuperar las matrices C\_part de cada nodo y combinarlas en una única matriz C int C\_SIZE = N; float[][] C = new float[C\_SIZE][C

## 2 / 2

## \_SIZE]; for (int k = 0; k < numParts; k++) { float[][] C\_part; if (k < 3) { C\_part = multipliers[0].getMatrixC\_part(k); // obtener la matriz C\_part del nodo 0 } else if (k < 6) { C\_part = multipliers[1].getMatrixC\_part(k - 3); // obtener la matriz C\_part del nodo 1 } else { C\_part = multipliers[2].getMatrixC\_part(k - 6); // obtener la matriz C\_part del nodo 2 }

## scss

## int startRow = k \* partSize;

## for (int i = startRow; i < startRow + partSize; i++) {

## for (int j = 0; j < C\_SIZE; j++) {

## C[i][j] = C\_part[i - startRow][j];

## }

## }

## }

## // Imprimir la matriz C

## System.out.println("Matriz C:");

## for (int i = 0; i < C\_SIZE; i++) {

## for (int j = 0; j < C\_SIZE; j++) {

## System.out.print(C[i][j] + " ");

## }

## System.out.println();

## }

## } catch (Exception e) {

## System.err.println("Excepción en el cliente: " + e.toString());

## e.printStackTrace();

## }

## } else if (opcion == 2) {

## try {

## int N = 12;

## int M = 6;

## float[][] A = new float[N][M];

## float[][] B = new float[M][N];

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < M; j++) {

## A[i][j] = 2 \* i + 3 \* j;

## B[j][i] = 3 \* i - 2 \* j;

## }

## }

## int numParts = 12;

## int partSize = N / numParts;

## for (int k = 0; k < numParts; k++) {

## float[][] A\_part = new float[partSize][M];

## float[][] B\_part = new float[M][partSize];

## int startRow = k \* partSize;

## for (int i = startRow; i < startRow + partSize; i++) {

## for (int j = 0; j < M; j++) {

## A\_part[i - startRow][j] = A[i][j];

## B\_part[j][i - startRow] = B[j][i];

## }

## }

## if (k < 4) {

## multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0

## multipliers[0].sendMatrixB(B\_part);

## } else if (k < 8) {

## multipliers[1].sendMatrixA(A\_part); // enviar al nodo 1

## multipliers[1].sendMatrixB(B\_part);

## } else if (k < 12) {

## multipliers[2].sendMatrixA(A\_part); // enviar al nodo 2

## multipliers[2].sendMatrixB(B\_part);

## }

## }

## // Recuperar las matrices C\_part de cada nodo y combinarlas en una única matriz C

## int C\_SIZE = N;

## float[][] C = new float[C\_SIZE][C\_SIZE];

## for (int k = 0; k < numParts; k++) {

## float[][] C\_part;

## if (

## en la Interface que le agregaria import java.rmi.Remote; import java.rmi.RemoteException; public interface Interface extends Remote { void sendMatrixA(float[][] A) throws RemoteException; void sendMatrixB(float[][] B) throws RemoteException; void borrarMatrices() throws RemoteException; }

## La interfaz ya tiene los métodos necesarios para enviar las matrices A y B a los servidores y para borrar las matrices almacenadas en los servidores. Sin embargo, si se quiere realizar una multiplicación de matrices utilizando estos métodos, se podría agregar un método adicional en la interfaz para realizar la operación, por ejemplo:

## scss

## float[][] multiplyMatrices() throws RemoteException;

## Este método recibiría las matrices A y B a través de los métodos sendMatrixA y sendMatrixB, realizaría la multiplicación de matrices en el servidor y devolvería el resultado como una matriz C de tipo float[][].

## y para PClass import java.rmi.RemoteException; import java.rmi.server.UnicastRemoteObject; public class PClass extends UnicastRemoteObject implements Interface { private static final long serialVersionUID = 1L; private float[][] A1, A2, A3, A4, A5, A6, A7, A8, A9; private float[][] B1, B2, B3, B4, B5, B6, B7, B8, B9; private int node; public PClass(int node) throws RemoteException { super(); this.node = node; } public void borrarMatrices() throws RemoteException { A1 = null; A2 = null; A3 = null; A4 = null; A5 = null; A6 = null; A7 = null; A8 = null; A9 = null; B1 = null; B2 = null; B3 = null; B4 = null; B5 = null; B6 = null; B7 = null; B8 = null; B9 = null; System.out.println("Matrices borradas"); } @Override public void sendMatrixA(float[][] A) throws RemoteException { if (node == 0) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; System.out.println("Matriz A1, A2, A3 recibida"); } } else if (node == 1) { if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; System.out.println("Matriz A4, A5, A6 recibida"); } } else if (node == 2) { if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; System.out.println("Matriz A7, A8, A9 recibida"); } } } @Override public void sendMatrixB(float[][] B) throws RemoteException { if (node == 0) { if (B1 == null) { B1 = B; } else if (B2 == null) { B2 = B; } else if (B3 == null) { B3 = B; System.out.println("Matriz B1, B2, B3 recibida"); } } else if (node == 1) { if (B4 == null) { B4 = B; } else if (B5 == null) { B5 = B; } else if (B6 == null) { B6 = B; System.out.println("Matriz B4, B5, B6 recibida"); } } else if (node == 2) { if (B7 == null) { B7 = B; } else if (B8 == null) { B8 = B; } else if (B9 == null) { B9 = B; System.out.println("Matriz B7, B8, B9 recibida"); } } } private void mostrarMatriz(float[][] matriz) { int rows = matriz.length; int cols = matriz[0].length; for (int i = 0; i < rows; i++) { for (int j = 0; j < cols; j++) { System.out.print(matriz[i][j] + "\t"); } System.out.println(); } } }

## The PClass implementation is responsible for receiving the matrix data and storing it into appropriate arrays. It also has a method to clear the arrays when needed.

## The PClass class implements the Interface interface, which declares the methods sendMatrixA, sendMatrixB, and borrarMatrices. The sendMatrixA method receives a 2D float array A representing a portion of the first matrix and stores it into the appropriate array based on the node parameter. Similarly, the sendMatrixB method receives a 2D float array B representing a portion of the second matrix and stores it into the appropriate array based on the node parameter. The borrarMatrices method clears all the matrix arrays and prints a message to confirm that the matrices have been cleared.

## The mostrarMatriz method is a helper method that takes a 2D float array representing a matrix and prints its contents to the console.

## Overall, this implementation is a basic skeleton for a distributed matrix multiplication algorithm. The actual matrix multiplication and coordination between nodes would need to be implemented elsewhere.

## donde agrego getMatrixC\_part

## En la interfaz Interface debes agregar el método getMatrixC\_part de la siguiente manera:

## scss

## public float[][] getMatrixC\_part(int part) throws RemoteException;

## Luego, en la clase PClass debes implementar este método de la siguiente manera:

## java

## @Override

## public float[][] getMatrixC\_part(int part) throws RemoteException {

## float[][] C\_part = null;

## switch (part) {

## case 1:

## C\_part = matrixMultiplication(A1, B1);

## break;

## case 2:

## C\_part = matrixMultiplication(A1, B2);

## break;

## case 3:

## C\_part = matrixMultiplication(A1, B3);

## break;

## case 4:

## C\_part = matrixMultiplication(A2, B1);

## break;

## case 5:

## C\_part = matrixMultiplication(A2, B2);

## break;

## case 6:

## C\_part = matrixMultiplication(A2, B3);

## break;

## case 7:

## C\_part = matrixMultiplication(A3, B1);

## break;

## case 8:

## C\_part = matrixMultiplication(A3, B2);

## break;

## case 9:

## C\_part = matrixMultiplication(A3, B3);

## break;

## }

## return C\_part;

## }

## Este método recibe como parámetro un número entero que representa la parte de la matriz C que se desea obtener. Luego, utiliza un switch statement para determinar qué matrices A y B se deben multiplicar para obtener la parte correspondiente de la matriz C. Por último, se devuelve la matriz C correspondiente a esa parte. La función matrixMultiplication es una función auxiliar que realiza la multiplicación de dos matrices y devuelve el resultado.

## y donde agrego matrixMultiplication

## La multiplicación de matrices se puede agregar en una nueva función dentro de la clase PClass. Por ejemplo, podrías agregar una función llamada matrixMultiplication que tome como parámetros las matrices A y B y devuelva la matriz resultante C.

## Aquí te dejamos un ejemplo de cómo podrías implementar la función matrixMultiplication en la clase PClass:

## css

## public float[][] matrixMultiplication(float[][] A, float[][] B) {

## int rowsA = A.length;

## int colsA = A[0].length;

## int rowsB = B.length;

## int colsB = B[0].length;

## 

## if (colsA != rowsB) {

## throw new IllegalArgumentException("Las matrices no pueden ser multiplicadas");

## }

## 

## float[][] C = new float[rowsA][colsB];

## 

## for (int i = 0; i < rowsA; i++) {

## for (int j = 0; j < colsB; j++) {

## float sum = 0;

## for (int k = 0; k < colsA; k++) {

## sum += A[i][k] \* B[k][j];

## }

## C[i][j] = sum;

## }

## }

## 

## return C;

## }

## Luego, podrías agregar una función getMatrixC\_part que llame a la función matrixMultiplication para calcular la submatriz correspondiente de la matriz C resultante. Por ejemplo:

## scss

## public float[][] getMatrixC\_part() {

## float[][] C\_part = matrixMultiplication(A1, B1);

## return C\_part;

## }

## Recuerda que debes implementar esta función para cada submatriz correspondiente de la matriz C.

## hola

## ¡Hola! ¿En qué puedo ayudarte hoy?

## bueno nos quedamos en algo de JAVA RMI verdad?

## ¡Hola! Sí, así es. Estábamos discutiendo la implementación de una interfaz RMI y una clase que implementa esa interfaz para realizar una multiplicación de matrices distribuida en varios nodos. ¿Tiene alguna pregunta o problema específico en el que pueda ayudarle?

## despues de recibir las matrices quiero que se multipliquen las matrices A1,A2,A3 con B1,B2,B3 @Override public void sendMatrixA(float[][] A) throws RemoteException { if (node == 0) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; System.out.println("Matriz A1, A2, A3 recibida"); } } else if (node == 1) { if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; System.out.println("Matriz A4, A5, A6 recibida"); } } else if (node == 2) { if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; System.out.println("Matriz A7, A8, A9 recibida"); } } } @Override public void sendMatrixB(float[][] B) throws RemoteException { if (node == 0) { if (B1 == null) { B1 = B; } else if (B2 == null) { B2 = B; } else if (B3 == null) { B3 = B; System.out.println("Matriz B1, B2, B3 recibida"); } } else if (node == 1) { if (B4 == null) { B4 = B; } else if (B5 == null) { B5 = B; } else if (B6 == null) { B6 = B; System.out.println("Matriz B4, B5, B6 recibida"); } } else if (node == 2) { if (B7 == null) { B7 = B; } else if (B8 == null) { B8 = B; } else if (B9 == null) { B9 = B; System.out.println("Matriz B7, B8, B9 recibida"); } } }

## Para realizar la multiplicación de matrices, puedes crear un método en la misma clase que reciba las matrices A y B y retorne la matriz resultante C. Podrías llamar a este método desde la clase que implementa la interfaz RMI después de recibir las matrices A y B.

## Por ejemplo, podrías crear un método llamado matrixMultiplication de la siguiente manera:

## java

## public float[][] matrixMultiplication(float[][] A, float[][] B) {

## int m = A.length;

## int n = B[0].length;

## int o = B.length;

## float[][] C = new float[m][n];

## for (int i = 0; i < m; i++) {

## for (int j = 0; j < n; j++) {

## for (int k = 0; k < o; k++) {

## C[i][j] += A[i][k] \* B[k][j];

## }

## }

## }

## return C;

## }

## Este método multiplica las matrices A y B y devuelve la matriz resultante C. Puedes llamar a este método después de recibir las matrices A y B en los métodos sendMatrixA y sendMatrixB para realizar la multiplicación de matrices y almacenar el resultado en una variable. Por ejemplo, podrías modificar los métodos sendMatrixA y sendMatrixB de la siguiente manera:

## java

## @Override

## public void sendMatrixA(float[][] A) throws RemoteException {

## if (node == 0) {

## if (A1 == null) {

## A1 = A;

## } else if (A2 == null) {

## A2 = A;

## } else if (A3 == null) {

## A3 = A;

## System.out.println("Matriz A1, A2, A3 recibida");

## if (B3 != null) {

## C1 = matrixMultiplication(A1, B1);

## C2 = matrixMultiplication(A1, B2);

## C3 = matrixMultiplication(A2, B1);

## C4 = matrixMultiplication(A2, B2);

## System.out.println("Matrices A y B multiplicadas");

## }

## }

## } else if (node == 1) {

## if (A4 == null) {

## A4 = A;

## } else if (A5 == null) {

## A5 = A;

## } else if (A6 == null) {

## A6 = A;

## System.out.println("Matriz A4, A5, A6 recibida");

## if (B6 != null) {

## C5 = matrixMultiplication(A4, B4);

## C6 = matrixMultiplication(A4, B5);

## C7 = matrixMultiplication(A5, B4);

## C8 = matrixMultiplication(A5, B5);

## System.out.println("Matrices A y B multiplicadas");

## }

## }

## } else if (node == 2) {

## if (A7 == null) {

## A7 = A;

## } else if (A8 == null) {

## A8 = A;

## } else if (A9 == null) {

## A9 = A;

## System.out.println("Matriz A7, A8, A9 recibida");

## if (B9 != null) {

## C9 = matrixMultiplication(A7, B7);

## C10 = matrixMultiplication(A7, B8);

## C11 = matrixMultiplication(A8, B7);

## C12 = matrixMultiplication(A8, B8

## continua por favor else if (node == 2) { if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; System.out.println("Matriz A7, A8, A9 recibida"); if (B9 != null) { C9 = matrixMultiplication(A7, B7); C10 = matrixMultiplication(A7, B8); C11 = matrixMultiplication(A8, B7);

## 2 / 2

## if (B8 != null) { C12 = matrixMultiplication(A8, B8); if (B9 != null) { C13 = matrixMultiplication(A9, B7); C14 = matrixMultiplication(A9, B8); C15 = matrixMultiplication(A9, B9); System.out.println("Matrices C9-C15 calculadas"); // Llamar al método sendMatrixC de los nodos vecinos sendMatrixCToNeighbor(); } } } } }

## El código anterior asume que las matrices A7, A8 y A9 y B7, B8 y B9 ya han sido recibidas por el nodo 2. Si ambas matrices están disponibles, se calculan las matrices C9, C10, C11, C12, C13, C14 y C15 mediante la multiplicación de matrices utilizando la función matrixMultiplication(). Luego, se llama al método sendMatrixCToNeighbor() para enviar las matrices C a los nodos vecinos.

## El código completo del método receiveMatrix() quedaría así:

## scss

## @Override

## public void receiveMatrix(int node, float[][] matrix, String type) throws RemoteException {

## if (type.equals("A")) {

## sendMatrixA(matrix);

## } else if (type.equals("B")) {

## sendMatrixB(matrix);

## } else if (type.equals("C")) {

## if (node == 0) {

## if (C1 == null) {

## C1 = matrix;

## } else if (C2 == null) {

## C2 = matrix;

## } else if (C3 == null) {

## C3 = matrix;

## System.out.println("Matriz C1, C2, C3 recibida");

## if (C4 != null) {

## C = joinMatrices();

## System.out.println("Matriz C calculada");

## }

## }

## } else if (node == 1) {

## if (C4 == null) {

## C4 = matrix;

## } else if (C5 == null) {

## C5 = matrix;

## } else if (C6 == null) {

## C6 = matrix;

## System.out.println("Matriz C4, C5, C6 recibida");

## if (C3 != null) {

## C = joinMatrices();

## System.out.println("Matriz C calculada");

## }

## }

## } else if (node == 2) {

## if (C7 == null) {

## C7 = matrix;

## } else if (C8 == null) {

## C8 = matrix;

## } else if (C9 == null) {

## C9 = matrix;

## System.out.println("Matriz C7, C8, C9 recibida");

## if (B9 != null) {

## C9 = matrixMultiplication(A7, B7);

## C10 = matrixMultiplication(A7, B8);

## C11 = matrixMultiplication(A8, B7);

## if (B8 != null) {

## C12 = matrixMultiplication(A8, B8);

## if (B9 != null) {

## C13 = matrixMultiplication(A9, B7);

## C14 = matrixMultiplication(A9, B8);

## C15 = matrixMultiplication(A9, B9);

## System.out.println("Matrices C9-C15 calculadas");

## // Llamar al método sendMatrixC de los nodos vecinos

## sendMatrixCToNeighbor();

## }

## }

## }

## }

## }

## }

## haber haz las modificaciones necesarias para que el nodo 0 pueda multiplicar solo las matrices que esten en el nodo 0 porfavor import java.rmi.RemoteException; import java.rmi.server.UnicastRemoteObject; public class PClass extends UnicastRemoteObject implements Interface { private static final long serialVersionUID = 1L; private float[][] A1, A2, A3, A4, A5, A6, A7, A8, A9; private float[][] B1, B2, B3, B4, B5, B6, B7, B8, B9; private int node; public PClass(int node) throws RemoteException { super(); this.node = node; } public void borrarMatrices() throws RemoteException { A1 = null; A2 = null; A3 = null; A4 = null; A5 = null; A6 = null; A7 = null; A8 = null; A9 = null; B1 = null; B2 = null; B3 = null; B4 = null; B5 = null; B6 = null; B7 = null; B8 = null; B9 = null; System.out.println("Matrices borradas"); } @Override public void sendMatrixA(float[][] A) throws RemoteException { if (node == 0) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; System.out.println("Matriz A1, A2, A3 recibida"); } } else if (node == 1) { if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; System.out.println("Matriz A4, A5, A6 recibida"); } } else if (node == 2) { if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; System.out.println("Matriz A7, A8, A9 recibida"); } } } @Override public void sendMatrixB(float[][] B) throws RemoteException { if (node == 0) { if (B1 == null) { B1 = B; } else if (B2 == null) { B2 = B; } else if (B3 == null) { B3 = B; System.out.println("Matriz B1, B2, B3 recibida"); } } else if (node == 1) { if (B4 == null) { B4 = B; } else if (B5 == null) { B5 = B; } else if (B6 == null) { B6 = B; System.out.println("Matriz B4, B5, B6 recibida"); } } else if (node == 2) { if (B7 == null) { B7 = B; } else if (B8 == null) { B8 = B; } else if (B9 == null) { B9 = B; System.out.println("Matriz B7, B8, B9 recibida"); } } }

## Para que el nodo 0 pueda multiplicar solo las matrices que estén en el nodo 0, debemos modificar el método matrixMultiplication() para que reciba como parámetros las matrices que se desean multiplicar, y luego agregar una lógica que solo permita la multiplicación si las dos matrices son del nodo 0.

## Aquí te dejo la implementación modificada de la clase PClass con los cambios necesarios:

## java

## import java.rmi.RemoteException;

## import java.rmi.server.UnicastRemoteObject;

## public class PClass extends UnicastRemoteObject implements Interface {

## private static final long serialVersionUID = 1L;

## private float[][] A1, A2, A3, A4, A5, A6, A7, A8, A9;

## private float[][] B1, B2, B3, B4, B5, B6, B7, B8, B9;

## private int node;

## public PClass(int node) throws RemoteException {

## super();

## this.node = node;

## }

## public void borrarMatrices() throws RemoteException {

## A1 = null;

## A2 = null;

## A3 = null;

## A4 = null;

## A5 = null;

## A6 = null;

## A7 = null;

## A8 = null;

## A9 = null;

## B1 = null;

## B2 = null;

## B3 = null;

## B4 = null;

## B5 = null;

## B6 = null;

## B7 = null;

## B8 = null;

## B9 = null;

## System.out.println("Matrices borradas");

## }

## 

## @Override

## public void sendMatrixA(float[][] A) throws RemoteException {

## if (node == 0) {

## if (A1 == null) {

## A1 = A;

## }

## else if (A2 == null) {

## A2 = A;

## }

## else if (A3 == null) {

## A3 = A;

## System.out.println("Matriz A1, A2, A3 recibida");

## }

## }

## else if (node == 1) {

## if (A4 == null) {

## A4 = A;

## }

## else if (A5 == null) {

## A5 = A;

## }

## else if (A6 == null) {

## A6 = A;

## System.out.println("Matriz A4, A5, A6 recibida");

## }

## } else if (node == 2) {

## if (A7 == null) {

## A7 = A;

## }

## else if (A8 == null) {

## A8 = A;

## }

## else if (A9 == null) {

## A9 = A;

## System.out.println("Matriz A7, A8, A9 recibida");

## }

## }

## }

## @Override

## public void sendMatrixB(float[][] B) throws RemoteException {

## if (node == 0) {

## if (B1 == null) {

## B1 = B;

## }

## else if (B2 == null) {

## B2 = B;

## }

## else if (B3 == null) {

## B3 = B;

## System.out.println("Matriz B1, B2, B3 recibida");

## }

## }

## else if (node == 1) {

## if (B4 == null) {

## B4 = B;

## }

## else if (B5 == null) {

## B5 = B;

## }

## 

## continua porfavor import java.rmi.RemoteException; import java.rmi.server.UnicastRemoteObject; public class PClass extends UnicastRemoteObject implements Interface { private static final long serialVersionUID = 1L; private float[][] A1, A2, A3, A4, A5, A6, A7, A8, A9; private float[][] B1, B2, B3, B4, B5, B6, B7, B8, B9; private int node; public PClass(int node) throws RemoteException { super(); this.node = node; } public void borrarMatrices() throws RemoteException { A1 = null; A2 = null; A3 = null; A4 = null; A5 = null; A6 = null; A7 = null; A8 = null; A9 = null; B1 = null; B2 = null; B3 = null; B4 = null; B5 = null; B6 = null; B7 = null; B8 = null; B9 = null; System.out.println("Matrices borradas"); } @Override public void sendMatrixA(float[][] A) throws RemoteException { if (node == 0) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; System.out.println("Matriz A1, A2, A3 recibida"); } } else if (node == 1) { if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; System.out.println("Matriz A4, A5, A6 recibida"); } } else if (node == 2) { if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; System.out.println("Matriz A7, A8, A9 recibida"); } } } @Override public void sendMatrixB(float[][] B) throws RemoteException { if (node == 0) { if (B1 == null) { B1 = B; } else if (B2 == null) { B2 = B; } else if (B3 == null) { B3 = B; System.out.println("Matriz B1, B2, B3 recibida"); } } else if (node == 1) { if (B4 == null) { B4 = B; } else if (B5 == null) { B5 = B; }

## 2 / 2

## else if (B6 == null) { B6 = B; System.out.println("Matriz B4, B5, B6 recibida"); } } else if (node == 2) { if (B7 == null) { B7 = B; } else if (B8 == null) { B8 = B; } else if (B9 == null) { B9 = B; System.out.println("Matriz B7, B8, B9 recibida"); } } }

## css

## @Override

## public float[][] getResult() throws RemoteException {

## int row = A1.length;

## int col = B1[0].length;

## float[][] result = new float[row][col];

## 

## for (int i = 0; i < row; i++) {

## for (int j = 0; j < col; j++) {

## result[i][j] = calculateElement(i, j);

## }

## }

## 

## return result;

## }

## private float calculateElement(int row, int col) {

## float element = 0;

## for (int i = 0; i < 3; i++) {

## element += A[row][i] \* B[i][col];

## }

## return element;

## }

## }

## eso lo hace el servidor o el cliente?

## Este código es para una clase que implementa la interfaz Interface, lo que sugiere que se trata del código que se ejecutará en el servidor.

## El servidor implementa los métodos definidos en la interfaz Interface y recibe las matrices enviadas por los clientes a través de la llamada a estos métodos. En este caso, los métodos son sendMatrixA y sendMatrixB que reciben las matrices A y B respectivamente, y el servidor almacena estas matrices en diferentes variables según el nodo al que pertenecen. También hay un método borrarMatrices que borra las matrices almacenadas en el servidor.

## me falta algo en el Interface import java.rmi.Remote; import java.rmi.RemoteException; public interface Interface extends Remote { void sendMatrixA(float[][] A) throws RemoteException; void sendMatrixB(float[][] B) throws RemoteException; void borrarMatrices() throws RemoteException; }

## 2 / 2

## No, no te falta nada en la interfaz. La interfaz simplemente define los métodos que los objetos remotos pueden invocar y las excepciones que pueden ocurrir durante la invocación de esos métodos.

## En este caso, la interfaz Interface define tres métodos remotos que se pueden invocar a través de RMI: sendMatrixA, sendMatrixB y borrarMatrices. Cada uno de estos métodos toma uno o más parámetros y lanza una excepción RemoteException en caso de que ocurra un error durante la invocación del método.

## En general, es importante definir claramente la interfaz de un objeto remoto, ya que todos los clientes y servidores que interactúan con el objeto deben conocer y adherirse a la misma interfaz.

## esto que hace @Override public float[][] getResult() throws RemoteException { int row = A1.length; int col = B1[0].length; float[][] result = new float[row][col]; for (int i = 0; i < row; i++) { for (int j = 0; j < col; j++) { result[i][j] = calculateElement(i, j); } } return result; } private float calculateElement(int row, int col) { float element = 0; for (int i = 0; i < 3; i++) { element += A[row][i] \* B[i][col]; } return element; }

## 2 / 2

## @Override is an annotation in Java that indicates that the method that follows is intended to override a method declared in a superclass or implemented in an interface.

## In the code you provided, `

## realice correctamente que cada nodo reciba 3 diferentes partes de una matriz B public void sendMatrixB(float[][] B) throws RemoteException { if (node == 0) { if (B1 == null) { B1 = B; } else if (B2 == null) { B2 = B; } else if (B3 == null) { B3 = B; System.out.println("Matriz B1, B2, B3 recibida"); } } else if (node == 1) { if (B4 == null) { B4 = B; } else if (B5 == null) { B5 = B; } else if (B6 == null) { B6 = B; System.out.println("Matriz B4, B5, B6 recibida"); } } else if (node == 2) { if (B7 == null) { B7 = B; } else if (B8 == null) { B8 = B; } else if (B9 == null) { B9 = B; System.out.println("Matriz B7, B8, B9 recibida"); } } }

## Sí, en el código que has mostrado se asigna correctamente cada una de las tres partes de la matriz B que se reciben en cada nodo.

## Para el nodo 0, si la variable B1 es nula, se asigna el valor de la matriz recibida a B1. Si B1 ya tiene valor, se verifica si B2 es nulo, en cuyo caso se asigna el valor de la matriz recibida a B2. De manera similar, si B2 ya tiene valor, se verifica si B3 es nulo, en cuyo caso se asigna el valor de la matriz recibida a B3. Si B3 se ha asignado correctamente, se muestra un mensaje indicando que las tres partes de la matriz B han sido recibidas.

## De manera similar, para el nodo 1 se asignan las partes de la matriz B a B4, B5 y B6, y para el nodo 2 se asignan a B7, B8 y B9.

## tengo una duda, entonces cada nodo recibe las mismas B del 1 al 9 ?

## No necesariamente. En una implementación típica de un algoritmo de multiplicación de matrices distribuido, la matriz B se divide en varias partes y cada nodo recibe una parte diferente de la matriz B. En el código que has mostrado, parece que cada nodo recibe tres partes diferentes de la matriz B, pero no se especifica cómo se divide la matriz B en estas partes.

## En general, la forma en que se divide la matriz B y se asigna a los diferentes nodos puede ser determinada por el programador. Una forma común de hacerlo es dividir la matriz B por filas, de modo que cada nodo reciba un conjunto diferente de filas de la matriz B. De esta forma, se garantiza que cada nodo recibe una parte diferente de la matriz B y que se está realizando el cálculo distribuido de manera efectiva.

## como es una multipliacion por metodo de jerarquia de memoria

## El método de jerarquía de memoria es una técnica de optimización de algoritmos que busca minimizar la cantidad de accesos a la memoria principal de un sistema de cómputo. Para ello, se utiliza una jerarquía de memorias con diferentes niveles de velocidad y capacidad, de manera que se puedan aprovechar las ventajas de cada nivel.

## En el contexto de una multiplicación de matrices, una implementación eficiente utilizando la jerarquía de memoria podría consistir en almacenar los elementos de las matrices en los niveles más rápidos de la jerarquía (por ejemplo, en la caché L1 o L2), y utilizar técnicas de localidad espacial y temporal para minimizar los accesos a memoria principal.

## Una forma de aprovechar la jerarquía de memoria para la multiplicación de matrices es utilizar la técnica de "bloqueo" o "particionamiento". Esta técnica consiste en dividir las matrices en bloques de tamaño adecuado, de manera que cada bloque quepa en la caché, y luego realizar la multiplicación por bloques.

## En resumen, el método de jerarquía de memoria puede ser utilizado en la multiplicación de matrices mediante el uso de técnicas como el bloqueo o particionamiento, que permiten reducir la cantidad de accesos a memoria principal y aprovechar la velocidad de los niveles más rápidos de la jerarquía.

## puedes hacer que A1 haga eso con las matrices B1,B2,B3 del nodo 0

## Sí, para que A1 realice la multiplicación de las matrices B1, B2 y B3 del nodo 0, se podría modificar el método getResult() de la siguiente manera:

## css

## @Override

## public float[][] getResult() throws RemoteException {

## int row = A1.length;

## int col = B1[0].length;

## float[][] result = new float[row][col];

## 

## // Multiplicar la sección de B1, B2, B3 con la matriz A1

## for (int i = 0; i < row; i++) {

## for (int j = 0; j < col; j++) {

## result[i][j] = calculateElement(i, j, B1[i], B2[i], B3[i]);

## }

## }

## 

## return result;

## }

## private float calculateElement(int row, int col, float[] B1\_row, float[] B2\_row, float[] B3\_row) {

## float element = 0;

## for (int i = 0; i < 3; i++) {

## element += A1[row][i] \* B1\_row[i];

## element += A1[row][i+3] \* B2\_row[i];

## element += A1[row][i+6] \* B3\_row[i];

## }

## return element;

## }

## En este caso, se está asumiendo que la matriz A1 tiene 9 columnas, correspondientes a las tres secciones de la matriz B del nodo 0. En el método calculateElement(), se realiza la multiplicación punto a punto de la fila i-ésima de A1 con cada una de las secciones de la matriz B del nodo 0, y se suman los resultados para obtener el elemento i,j de la matriz resultado.

## ok pero tengo otra duda en esta parte especifica que me piden Se deberá dividir las matrices A y BT en nueve partes iguales, por tanto la matriz C estará dividida en 81 partes:

## Si se está dividiendo la matriz A y la matriz B en nueve partes iguales, entonces cada parte de la matriz A se multiplicará por cada una de las partes de la matriz B. Esto resultará en 81 partes de la matriz C (9x9), cada una de las cuales es el resultado de multiplicar una parte de A por una parte de B.

## En otras palabras, cada nodo recibirá una parte de la matriz A y tres partes diferentes de la matriz B, y cada nodo calculará una parte correspondiente de la matriz C utilizando la parte de A y cada una de las tres partes de B que recibió. Después de que todos los nodos hayan calculado sus partes correspondientes de la matriz C, se deberán combinar para obtener la matriz C completa.

## puedes hacer eso para PClass?

## 2 / 2

## Claro, aquí te muestro un ejemplo de cómo podrías implementar la clase PClass:

## java

## public class PClass implements Runnable {

## private int id;

## private float[][] A1;

## private float[][] B1;

## private float[][] B2;

## private float[][] B3;

## private float[][] B4;

## private float[][] B5;

## private float[][] B6;

## private float[][] B7;

## private float[][] B8;

## private float[][] B9;

## private Interface server;

## 

## public PClass(int id, Interface server) {

## this.id = id;

## this.server = server;

## }

## 

## public void run() {

## try {

## if (id == 0) {

## server.sendMatrixA(A1);

## server.sendMatrixB(B1);

## server.sendMatrixB(B2);

## server.sendMatrixB(B3);

## float[][] result = server.getResult();

## // procesar resultado

## }

## else if (id == 1) {

## server.sendMatrixA(A1);

## server.sendMatrixB(B4);

## server.sendMatrixB(B5);

## server.sendMatrixB(B6);

## float[][] result = server.getResult();

## // procesar resultado

## }

## else if (id == 2) {

## server.sendMatrixA(A1);

## server.sendMatrixB(B7);

## server.sendMatrixB(B8);

## server.sendMatrixB(B9);

## float[][] result = server.getResult();

## // procesar resultado

## }

## else {

## System.out.println("Id de nodo inválido");

## }

## } catch (RemoteException e) {

## e.printStackTrace();

## }

## }

## 

## public void setA(float[][] A) {

## this.A1 = A;

## }

## 

## public void setB1(float[][] B1) {

## this.B1 = B1;

## }

## 

## public void setB2(float[][] B2) {

## this.B2 = B2;

## }

## 

## public void setB3(float[][] B3) {

## this.B3 = B3;

## }

## 

## public void setB4(float[][] B4) {

## this.B4 = B4;

## }

## 

## public void setB5(float[][] B5) {

## this.B5 = B5;

## }

## 

## public void setB6(float[][] B6) {

## this.B6 = B6;

## }

## 

## public void setB7(float[][] B7) {

## this.B7 = B7;

## }

## 

## public void setB8(float[][] B8) {

## this.B8 = B8;

## }

## 

## public void setB9(float[][] B9) {

## this.B9 = B9;

## }

## }

## En este ejemplo, PClass implementa la interfaz Runnable, lo que permite que cada instancia de PClass sea ejecutada como un hilo independiente. Cada instancia de PClass tiene un identificador id que indica a qué nodo de procesamiento corresponde, y se le pasa una referencia a la interfaz remota server que permite la comunicación con el servidor.

## El método run implementa la lógica de cada nodo de procesamiento, donde se envían las matrices correspondientes al servidor y se espera por el resultado. El resultado se procesa después de recibirlo del servidor.

## Los métodos setA y setB1 a setB9 se utilizan para establecer las matrices A y B correspondientes en cada instancia de PClass. Estos métodos son llamados

## este es mi PClass puedes agregar eso en esto import java.rmi.RemoteException; import java.rmi.server.UnicastRemoteObject; public class PClass extends UnicastRemoteObject implements Interface { private static final long serialVersionUID = 1L; private float[][] A1, A2, A3, A4, A5, A6, A7, A8, A9; private float[][] B1, B2, B3, B4, B5, B6, B7, B8, B9; private int node; public PClass(int node) throws RemoteException { super(); this.node = node; } public void borrarMatrices() throws RemoteException { A1 = null; A2 = null; A3 = null; A4 = null; A5 = null; A6 = null; A7 = null; A8 = null; A9 = null; B1 = null; B2 = null; B3 = null; B4 = null; B5 = null; B6 = null; B7 = null; B8 = null; B9 = null; System.out.println("Matrices borradas"); } @Override public void sendMatrixA(float[][] A) throws RemoteException { if (node == 0) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; System.out.println("Matriz A1, A2, A3 recibida"); } } else if (node == 1) { if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; System.out.println("Matriz A4, A5, A6 recibida"); } } else if (node == 2) { if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; System.out.println("Matriz A7, A8, A9 recibida"); } } } @Override public void sendMatrixB(float[][] B) throws RemoteException { if (node == 0) { if (B1 == null) { B1 = B; } else if (B2 == null) { B2 = B; } else if (B3 == null) { B3 = B; System.out.println("Matriz B1, B2, B3 recibida"); } } else if (node == 1) { if (B4 == null) { B4 = B; } else if (B5 == null) { B5 = B; } else if (B6 == null) { B6 = B; System.out.println("Matriz B4, B5, B6 recibida"); } } else if (node == 2) { if (B7 == null) { B7 = B; } else if (B8 == null) { B8 = B; } else if (B9 == null) { B9 = B; System.out.println("Matriz B7, B8, B9 recibida"); } } } private void mostrarMatriz(float[][] matriz) { int rows = matriz.length; int cols = matriz[0].length; for (int i = 0; i < rows; i++) { for (int j = 0; j < cols; j++) { System.out.print(matriz[i][j] + "\t"); } System.out.println(); } } }

## Sí, puedes agregar el método para multiplicar las matrices A1 y B1, B2 y B3 del nodo 0 en la clase PClass de la siguiente manera:

## css

## public float[][] multiply() throws RemoteException {

## float[][] C1 = new float[A1.length/3][B1[0].length/3];

## for (int i = 0; i < A1.length/3; i++) {

## for (int j = 0; j < B1[0].length/3; j++) {

## for (int k = 0; k < A1[0].length/3; k++) {

## C1[i][j] += A1[i][k] \* B1[k][j];

## }

## }

## }

## float[][] C2 = new float[A1.length/3][B1[0].length/3];

## for (int i = 0; i < A1.length/3; i++) {

## for (int j = B1[0].length/3; j < 2\*B1[0].length/3; j++) {

## for (int k = 0; k < A1[0].length/3; k++) {

## C2[i][j-B1[0].length/3] += A1[i][k] \* B2[k][j-B1[0].length/3];

## }

## }

## }

## float[][] C3 = new float[A1.length/3][B1[0].length/3];

## for (int i = 0; i < A1.length/3; i++) {

## for (int j = 2\*B1[0].length/3; j < B1[0].length; j++) {

## for (int k = 0; k < A1[0].length/3; k++) {

## C3[i][j-2\*B1[0].length/3] += A1[i][k] \* B3[k][j-2\*B1[0].length/3];

## }

## }

## }

## float[][] C = new float[A1.length][B1[0].length];

## for (int i = 0; i < A1.length/3; i++) {

## for (int j = 0; j < B1[0].length/3; j++) {

## C[i][j] = C1[i][j];

## }

## }

## for (int i = 0; i < A1.length/3; i++) {

## for (int j = B1[0].length/3; j < 2\*B1[0].length/3; j++) {

## C[i][j] = C2[i][j-B1[0].length/3];

## }

## }

## for (int i = 0; i < A1.length/3; i++) {

## for (int j = 2\*B1[0].length/3; j < B1[0].length; j++) {

## C[i][j] = C3[i][j-2\*B1[0].length/3];

## }

## }

## return C;

## }

## Este método multiplica las matrices A1 y B1, B2 y B3 del nodo 0 y devuelve la matriz resultante C1. Luego, coloca los elementos de C1, C2 y C3 en la matriz C y devuelve C. Puedes llamar a este método desde otro nodo de la siguiente manera:

## graphql

## Interface p0 = (Interface) Naming.lookup("rmi://

## y para esto Utilizando RMI obtener las matrices C1, C2, ... C27 invocando el método multiplica\_matrices() el cual ejecutará en el nodo 0

## Puedes agregar el siguiente método en la clase PClass:

## scss

## @Override

## public float[][] multiplica\_matrices() throws RemoteException {

## if (node != 0) {

## return null; // Solo el nodo 0 tiene la matriz completa

## }

## float[][] C1 = multiplicarMatrices(A1, B1);

## float[][] C2 = multiplicarMatrices(A1, B2);

## float[][] C3 = multiplicarMatrices(A1, B3);

## float[][] C4 = multiplicarMatrices(A2, B1);

## float[][] C5 = multiplicarMatrices(A2, B2);

## float[][] C6 = multiplicarMatrices(A2, B3);

## float[][] C7 = multiplicarMatrices(A3, B1);

## float[][] C8 = multiplicarMatrices(A3, B2);

## float[][] C9 = multiplicarMatrices(A3, B3);

## float[][] C10 = multiplicarMatrices(A4, B1);

## float[][] C11 = multiplicarMatrices(A4, B2);

## float[][] C12 = multiplicarMatrices(A4, B3);

## float[][] C13 = multiplicarMatrices(A5, B1);

## float[][] C14 = multiplicarMatrices(A5, B2);

## float[][] C15 = multiplicarMatrices(A5, B3);

## float[][] C16 = multiplicarMatrices(A6, B1);

## float[][] C17 = multiplicarMatrices(A6, B2);

## float[][] C18 = multiplicarMatrices(A6, B3);

## float[][] C19 = multiplicarMatrices(A7, B1);

## float[][] C20 = multiplicarMatrices(A7, B2);

## float[][] C21 = multiplicarMatrices(A7, B3);

## float[][] C22 = multiplicarMatrices(A8, B1);

## float[][] C23 = multiplicarMatrices(A8, B2);

## float[][] C24 = multiplicarMatrices(A8, B3);

## float[][] C25 = multiplicarMatrices(A9, B1);

## float[][] C26 = multiplicarMatrices(A9, B2);

## float[][] C27 = multiplicarMatrices(A9, B3);

## 

## return concatenarMatrices(concatenarMatrices(C1, C2, C3),

## concatenarMatrices(C4, C5, C6),

## concatenarMatrices(C7, C8, C9),

## concatenarMatrices(C10, C11, C12),

## concatenarMatrices(C13, C14, C15),

## concatenarMatrices(C16, C17, C18),

## concatenarMatrices(C19, C20, C21),

## concatenarMatrices(C22, C23, C24),

## concatenarMatrices(C25, C26, C27));

## }

## // Funciones auxiliares para multiplicar y concatenar matrices

## private float[][] multiplicarMatrices(float[][] A, float[][] B) {

## int n = A.length;

## int m = A[0].length;

## int p = B[0].length;

## float[][] C = new float[n][p];

## for (int i = 0; i < n; i++) {

## for (int j = 0; j < p; j++) {

## for (int k = 0; k < m; k++) {

## C[i][j] += A[i][k] \* B[k][j];

## }

## }

## }

## return C;

## }

## private float

## continua con el codigo // Funciones auxiliares para multiplicar y concatenar matrices private float[][] multiplicarMatrices(float[][] A, float[][] B) { int n = A.length; int m = A[0].length; int p = B[0].length; float[][] C = new float[n][p]; for (int i = 0; i < n; i++) { for (int j = 0; j < p; j++) { for (int k = 0; k < m; k++) { C[i][j] += A[i][k] \* B[k][j]; } } } return C; }

## Esta función auxiliar multiplicarMatrices(A, B) toma dos matrices A y B y devuelve su producto de matrices C, donde C[i][j] es el producto punto del i-ésimo renglón de A con el j-ésimo columna de B. Esta función utiliza un algoritmo básico de multiplicación de matrices utilizando tres ciclos for anidados para calcular el producto punto.

## Aquí está el código completo:

## css

## private float[][] multiplicarMatrices(float[][] A, float[][] B) {

## int n = A.length;

## int m = A[0].length;

## int p = B[0].length;

## float[][] C = new float[n][p];

## for (int i = 0; i < n; i++) {

## for (int j = 0; j < p; j++) {

## for (int k = 0; k < m; k++) {

## C[i][j] += A[i][k] \* B[k][j];

## }

## }

## }

## return C;

## }

## Puedes llamar a esta función para multiplicar dos matrices en cualquier parte del código que lo necesite.

## bueno ahora que el RMIClient envie las matrices a RMIServer como le hago para que el RMIServer haga la multiplicacion y se la envie al RMIClient

## Para que el RMIServer realice la multiplicación de matrices y envíe el resultado al RMIClient, necesitas agregar algunos cambios en el código.

## Primero, crea un método en la clase PClass llamado multiplicaMatrices() que tome como argumentos los nombres de las matrices A y B a multiplicar (por ejemplo, A1, B2) y que retorne la matriz resultante. Dentro de este método, llama a la función multiplicarMatrices() que ya has definido anteriormente para realizar la multiplicación.

## Luego, en la clase RMIServer, agrega un método llamado multiplica\_matrices() que se encargará de llamar al método multiplicaMatrices() en la instancia de PClass y retornar el resultado. Aquí es donde se realizará la multiplicación de todas las combinaciones de matrices A y B.

## Finalmente, en la clase RMIClient, puedes llamar al método multiplica\_matrices() en la instancia de Interface en el nodo 0 y recibir la matriz resultante en el cliente.

## Aquí te muestro el código modificado:

## En la clase PClass:

## java

## public float[][] multiplicarMatrices(float[][] A, float[][] B) {

## // Código de la función multiplicarMatrices()

## // ...

## }

## public float[][] multiplicaMatrices(String aName, String bName) throws RemoteException {

## float[][] A = null, B = null;

## switch (aName) {

## case "A1": A = A1; break;

## case "A2": A = A2; break;

## // Agrega los otros casos para las matrices A

## }

## switch (bName) {

## case "B1": B = B1; break;

## case "B2": B = B2; break;

## // Agrega los otros casos para las matrices B

## }

## if (A == null || B == null) {

## return null; // Maneja el caso de matrices no encontradas

## }

## return multiplicarMatrices(A, B);

## }

## En la clase RMIServer:

## java

## public float[][] multiplica\_matrices() throws RemoteException {

## Interface p = (Interface) Naming.lookup("//localhost/P");

## float[][] C1 = p.multiplicaMatrices("A1", "B1");

## float[][] C2 = p.multiplicaMatrices("A1", "B2");

## float[][] C3 = p.multiplicaMatrices("A1", "B3");

## float[][] C4 = p.multiplicaMatrices("A2", "B1");

## float[][] C5 = p.multiplicaMatrices("A2", "B2");

## float[][] C6 = p.multiplicaMatrices("A2", "B3");

## float[][] C7 = p.multiplicaMatrices("A3", "B1");

## float[][] C8 = p.multiplicaMatrices("A3", "B2");

## float[][] C9 = p.multiplicaMatrices("A3", "B3");

## float[][] C10 = p.multiplicaMatrices("A4", "B1");

## float[][] C11 = p.multiplicaMatrices("A4", "B2");

## float[][] C12 = p.multiplicaMatrices("A4", "B3");

## float[][] C13 = p.multiplicaMatrices("A5", "B1");

## float[][] C14 = p.multiplicaMatrices("A5

## donde agregaria eso en mi RMIServer case 0: System.out.println("Nodo 0"); try { int port = 1099; // Puerto por defecto String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(node); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break;

## 2 / 2

## El código que permite al RMIServer recibir las matrices y realizar la multiplicación debería estar en la clase PClass (que se está vinculando con el registro RMI en el código que proporcionaste). En la implementación del método multiplica\_matrices() en PClass, deberías tener el siguiente código para realizar la multiplicación y devolver el resultado al cliente:

## java

## public float[][] multiplica\_matrices(float[][] matrices) throws RemoteException {

## float[][] result = matrices[0]; // Inicializa con la primera matriz

## for (int i = 1; i < matrices.length; i++) {

## result = multiplicarMatrices(result, matrices[i]); // Multiplica por cada una de las matrices

## }

## return result;

## }

## Este método recibe la matriz que contiene todas las matrices a multiplicar (que ha sido enviada por el cliente), utiliza la función multiplicarMatrices() (que ya definiste previamente) para realizar la multiplicación y devuelve el resultado al cliente. Puedes llamar a esta función en el RMIClient para obtener la matriz resultante después de la multiplicación.

## solamente quiero que se multiplique la A1 y B1 y se guarde en C1 porfavor solamente eso import java.rmi.RemoteException; import java.rmi.server.UnicastRemoteObject; public class PClass extends UnicastRemoteObject implements Interface { private static final long serialVersionUID = 1L; private float[][] A1, A2, A3, A4, A5, A6, A7, A8, A9; private float[][] B1, B2, B3, B4, B5, B6, B7, B8, B9; private int node; public PClass(int node) throws RemoteException { super(); this.node = node; } public void borrarMatrices() throws RemoteException { A1 = null; A2 = null; A3 = null; A4 = null; A5 = null; A6 = null; A7 = null; A8 = null; A9 = null; B1 = null; B2 = null; B3 = null; B4 = null; B5 = null; B6 = null; B7 = null; B8 = null; B9 = null; System.out.println("Matrices borradas"); } @Override public void sendMatrixA(float[][] A) throws RemoteException { if (node == 0) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; System.out.println("Matriz A1, A2, A3 recibida"); mostrarMatriz(A1); } } else if (node == 1) { if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; System.out.println("Matriz A4, A5, A6 recibida"); } } else if (node == 2) { if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; System.out.println("Matriz A7, A8, A9 recibida"); } } } @Override public void sendMatrixB(float[][] B) throws RemoteException { if (node == 0) { if (B1 == null) { B1 = B; } else if (B2 == null) { B2 = B; } else if (B3 == null) { B3 = B; System.out.println("Matriz B1, B2, B3 recibida"); mostrarMatriz(B1); } } else if (node == 1) { if (B4 == null) { B4 = B; } else if (B5 == null) { B5 = B; } else if (B6 == null) { B6 = B; System.out.println("Matriz B4, B5, B6 recibida"); } } else if (node == 2) { if (B7 == null) { B7 = B; } else if (B8 == null) { B8 = B; } else if (B9 == null) { B9 = B; System.out.println("Matriz B7, B8, B9 recibida"); } } } private void mostrarMatriz(float[][] matriz) { int rows = matriz.length; int cols = matriz[0].length; for (int i = 0; i < rows; i++) { for (int j = 0; j < cols; j++) { System.out.print(matriz[i][j] + "\t"); } System.out.println(); } } }

## Para realizar la multiplicación de la matriz A1 y la matriz B1 y almacenar el resultado en la matriz C1, puedes crear un método adicional en la clase PClass que realice la multiplicación de estas dos matrices. Por ejemplo:

## scss

## public float[][] multiplicarMatrices() throws RemoteException {

## if (A1 == null || B1 == null) {

## throw new RemoteException("No se han recibido todas las matrices A y B necesarias");

## }

## int rowsA = A1.length;

## int colsA = A1[0].length;

## int rowsB = B1.length;

## int colsB = B1[0].length;

## if (colsA != rowsB) {

## throw new RemoteException("Las matrices A y B no tienen dimensiones compatibles para multiplicar");

## }

## float[][] C1 = new float[rowsA][colsB];

## for (int i = 0; i < rowsA; i++) {

## for (int j = 0; j < colsB; j++) {

## for (int k = 0; k < colsA; k++) {

## C1[i][j] += A1[i][k] \* B1[k][j];

## }

## }

## }

## return C1;

## }

## Este método verifica que las matrices A1 y B1 se hayan recibido y que tengan dimensiones compatibles para la multiplicación. Luego, realiza la multiplicación de las matrices y devuelve la matriz resultante C1.

## Luego, en tu RMIServer, puedes agregar una nueva opción en el switch statement que invoque el método multiplicarMatrices() y envíe el resultado de vuelta al RMIClient. Por ejemplo:

## csharp

## case 1:

## System.out.println("Nodo 1");

## try {

## Interface server = (Interface) Naming.lookup("rmi://localhost:1099/MatrixMultiplier");

## float[][] C1 = server.multiplicarMatrices();

## System.out.println("Matriz C1 calculada");

## server.sendMatrixC(C1);

## System.out.println("Matriz C1 enviada");

## } catch (Exception e) {

## System.err.println("Error: " + e.getMessage());

## }

## break;

## Este código busca el objeto remoto en el registro RMI, invoca el método multiplicarMatrices() en el servidor para calcular la matriz C1, y luego llama al método sendMatrixC() en el objeto remoto para enviar la matriz C1 de vuelta al RMIClient. Nota que deberás agregar un nuevo método sendMatrixC() a la interfaz Interface y a la clase PClass para permitir el envío de la matriz C1.

## agrega la parte de RMIServer en mi codigo case 0: System.out.println("Nodo 0"); try { int port = 1099; // Puerto por defecto String ipAddress = "localhost"; LocateRegistry.createRegistry(port); PClass multiplier = new PClass(node); Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier); System.out.println("Servidor RMI listo en " + ipAddress + ":" + port); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } break;

## Claro, aquí te agrego el código para configurar el servidor RMI en la opción 0 de tu switch:

## csharp

## case 0:

## System.out.println("Nodo 0");

## try {

## int port = 1099; // Puerto por defecto

## String ipAddress = "localhost";

## LocateRegistry.createRegistry(port);

## PClass multiplier = new PClass(node);

## Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier);

## System.out.println("Servidor RMI listo en " + ipAddress + ":" + port);

## } catch (Exception e) {

## System.err.println("Error: " + e.getMessage());

## }

## break;

## En este código, primero se define el puerto (1099) y la dirección IP del servidor (localhost). Luego se crea un registro RMI en el puerto especificado usando el método LocateRegistry.createRegistry(). Después se instancia un objeto PClass y se registra con el nombre "MatrixMultiplier" en el registro RMI usando el método Naming.rebind(). Finalmente, se imprime un mensaje indicando que el servidor RMI está listo y en ejecución.

## agrega esto float[][] C1 = server.multiplicarMatrices(); System.out.println("Matriz C1 calculada"); server.sendMatrixC(C1); System.out.println("Matriz C1 enviada");

## Asumiendo que server es la instancia del objeto remoto que implementa la interfaz Interface y que ya se han recibido las matrices A y B en los métodos sendMatrixA() y sendMatrixB() respectivamente, puede agregar estas líneas dentro del case 0 en su código:

## csharp

## case 0:

## System.out.println("Nodo 0");

## try {

## int port = 1099; // Puerto por defecto

## String ipAddress = "localhost";

## LocateRegistry.createRegistry(port);

## PClass multiplier = new PClass(node);

## Naming.rebind("rmi://" + ipAddress + ":" + port + "/MatrixMultiplier", multiplier);

## System.out.println("Servidor RMI listo en " + ipAddress + ":" + port);

## 

## // Realizar la multiplicación de matrices y enviar el resultado

## float[][] C1 = server.multiplicarMatrices();

## System.out.println("Matriz C1 calculada");

## server.sendMatrixC(C1);

## System.out.println("Matriz C1 enviada");

## 

## } catch (Exception e) {

## System.err.println("Error: " + e.getMessage());

## }

## break;

## Espero que esto ayude.

## este esmi INterface import java.rmi.Remote; import java.rmi.RemoteException; public interface Interface extends Remote { void sendMatrixA(float[][] A) throws RemoteException; void sendMatrixB(float[][] B) throws RemoteException; void borrarMatrices() throws RemoteException; }

## El código que me proporcionaste parece estar incompleto, pero aquí está cómo podría verse con la adición de la línea que mencionaste:

## java

## import java.rmi.Remote;

## import java.rmi.RemoteException;

## public interface Interface extends Remote {

## void sendMatrixA(float[][] A) throws RemoteException;

## void sendMatrixB(float[][] B) throws RemoteException;

## void sendMatrixC(float[][] C) throws RemoteException;

## void borrarMatrices() throws RemoteException;

## }

## Agregué un nuevo método sendMatrixC para permitir que los nodos envíen la matriz resultante de la multiplicación de matrices de vuelta al nodo 0. Asegúrate de que la implementación de PClass incluya el método sendMatrixC.

## como seria sendMatrixC en PClass te mando mi codigo de PClass import java.rmi.RemoteException; import java.rmi.server.UnicastRemoteObject; public class PClass extends UnicastRemoteObject implements Interface { private static final long serialVersionUID = 1L; private float[][] A1, A2, A3, A4, A5, A6, A7, A8, A9; private float[][] B1, B2, B3, B4, B5, B6, B7, B8, B9; private int node; public PClass(int node) throws RemoteException { super(); this.node = node; } public void borrarMatrices() throws RemoteException { A1 = null; A2 = null; A3 = null; A4 = null; A5 = null; A6 = null; A7 = null; A8 = null; A9 = null; B1 = null; B2 = null; B3 = null; B4 = null; B5 = null; B6 = null; B7 = null; B8 = null; B9 = null; System.out.println("Matrices borradas"); } @Override public void sendMatrixA(float[][] A) throws RemoteException { if (node == 0) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; System.out.println("Matriz A1, A2, A3 recibida"); mostrarMatriz(A1); } } else if (node == 1) { if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; System.out.println("Matriz A4, A5, A6 recibida"); } } else if (node == 2) { if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; System.out.println("Matriz A7, A8, A9 recibida"); } } } @Override public void sendMatrixB(float[][] B) throws RemoteException { if (node == 0) { if (B1 == null) { B1 = B; } else if (B2 == null) { B2 = B; } else if (B3 == null) { B3 = B; System.out.println("Matriz B1, B2, B3 recibida"); mostrarMatriz(B1); } } else if (node == 1) { if (B4 == null) { B4 = B; } else if (B5 == null) { B5 = B; } else if (B6 == null) { B6 = B; System.out.println("Matriz B4, B5, B6 recibida"); } } else if (node == 2) { if (B7 == null) { B7 = B; } else if (B8 == null) { B8 = B; } else if (B9 == null) { B9 = B; System.out.println("Matriz B7, B8, B9 recibida"); } } } private void mostrarMatriz(float[][] matriz) { int rows = matriz.length; int cols = matriz[0].length; for (int i = 0; i < rows; i++) { for (int j = 0; j < cols; j++) { System.out.print(matriz[i][j] + "\t"); } System.out.println(); } } public float[][] multiplicarMatrices() throws RemoteException { if (A1 == null || B1 == null) { throw new RemoteException("No se han recibido todas las matrices A y B necesarias"); } int rowsA = A1.length; int colsA = A1[0].length; int rowsB = B1.length; int colsB = B1[0].length; if (colsA != rowsB) { throw new RemoteException("Las matrices A y B no tienen dimensiones compatibles para multiplicar"); } float[][] C1 = new float[rowsA][colsB]; for (int i = 0; i < rowsA; i++) { for (int j = 0; j < colsB; j++) { for (int k = 0; k < colsA; k++) { C1[i][j] += A1[i][k] \* B1[k][j]; } } } mostrarMatriz(C1); return C1; } }

## 2 / 2

## Para añadir el método sendMatrixC a la clase PClass puedes hacer lo siguiente:

## java

## @Override

## public void sendMatrixC(float[][] C) throws RemoteException {

## // implementa la lógica de recepción de la matriz C según las necesidades de tu aplicación

## }

## Luego, en el cliente que invoca los métodos de PClass, puedes llamar al método sendMatrixC de la siguiente manera:

## java

## interfaceObj.sendMatrixC(matrizC);

## Donde interfaceObj es una instancia de la interfaz Interface que apunta a un objeto remoto de tipo PClass, y matrizC es la matriz que deseas enviar. Asegúrate de que tanto el cliente como el servidor estén de acuerdo sobre el orden y la forma en que se deben enviar las matrices A, B y C para que la aplicación funcione correctamente.

## ok ahora quiero hacer un ciclo for dentro de esto, en donde C1 cambie a C2 y asi cada vez que suceda el ciclo for float[][] C1 = new float[rowsA][colsB]; for (int i = 0; i < rowsA; i++) { for (int j = 0; j < colsB; j++) { for (int k = 0; k < colsA; k++) { C1[i][j] += A1[i][k] \* B1[k][j]; } } } mostrarMatriz(C1);

## 2 / 2

## Para hacer que la matriz de salida cambie en cada iteración del ciclo for, puedes declarar una matriz de salida adicional y alternar su uso en cada iteración. Por ejemplo:

## scss

## float[][] C1 = new float[rowsA][colsB];

## float[][] C2 = new float[rowsA][colsB];

## boolean useC1 = true;

## for (int iter = 0; iter < numIterations; iter++) {

## // Perform matrix multiplication

## if (useC1) {

## for (int i = 0; i < rowsA; i++) {

## for (int j = 0; j < colsB; j++) {

## for (int k = 0; k < colsA; k++) {

## C1[i][j] += A1[i][k] \* B1[k][j];

## }

## }

## }

## mostrarMatriz(C1);

## useC1 = false;

## } else {

## for (int i = 0; i < rowsA; i++) {

## for (int j = 0; j < colsB; j++) {

## for (int k = 0; k < colsA; k++) {

## C2[i][j] += A1[i][k] \* B1[k][j];

## }

## }

## }

## mostrarMatriz(C2);

## useC1 = true;

## }

## }

## En este ejemplo, hemos agregado una nueva matriz C2 y una variable booleana useC1 que indica si debemos usar C1 o C2 en la iteración actual del ciclo for. Dentro del ciclo for, comprobamos el valor de useC1 y usamos la matriz correspondiente en el cálculo de la multiplicación de la matriz y en la llamada a la función mostrarMatriz. Después de cada iteración, alternamos el valor de useC1 para que el siguiente ciclo for use la otra matriz.

## También hemos agregado una nueva variable numIterations que controla cuántas iteraciones del ciclo for se realizarán. Puedes ajustar este valor según tus necesidades.

## habra una forma rapida para declarar del C1 hasta el C27 sin que sea de forma manual float[][] C1 = new float[rowsA][colsB];

## Sí, se puede usar un arreglo de arreglos para almacenar todas las matrices C de forma dinámica. Por ejemplo, se puede crear un arreglo de longitud 27, donde cada elemento del arreglo sea un arreglo de dos dimensiones que representa una matriz C:

## css

## float[][][] matricesC = new float[27][rowsA][colsB];

## Luego, se puede iterar sobre el arreglo de matrices C y realizar el producto punto entre A y B para cada matriz C:

## scss

## for (int c = 0; c < matricesC.length; c++) {

## for (int i = 0; i < rowsA; i++) {

## for (int j = 0; j < colsB; j++) {

## for (int k = 0; k < colsA; k++) {

## matricesC[c][i][j] += A1[i][k] \* B1[k][j];

## }

## }

## }

## mostrarMatriz(matricesC[c]);

## }

## En este ejemplo, se utiliza la variable c para iterar sobre las matrices C en el arreglo de matrices matricesC. La función mostrarMatriz se llama para cada matriz C calculada en el ciclo for.

## repara este error public float[][] multiplicarMatrices0() throws RemoteException { int rowsA = A1.length; int colsA = A1[0].length; int colsB = B1[0].length; float[][][] matricesC = new float[2][rowsA][colsB]; for (int c = 0; c < matricesC.length; c++) { for (int i = 0; i < rowsA; i++) { for (int j = 0; j < colsB; j++) { for (int k = 0; k < colsA; k++) { matricesC[c][i][j] += A1[i][k] \* B1[k][j]; } } } mostrarMatriz(matricesC[c]); } return matricesC; }

## El error en el código se debe a que estás tratando de devolver una matriz tridimensional (float[][][] matricesC) cuando la firma del método indica que debería devolver una matriz bidimensional (float[][]).

## Si deseas devolver varias matrices C, puedes declarar un arreglo de matrices float[][] C con la cantidad deseada de elementos y luego asignar las matrices calculadas a cada elemento del arreglo dentro del bucle for. Por ejemplo:

## scss

## public float[][][] multiplicarMatrices0() throws RemoteException {

## int rowsA = A1.length;

## int colsA = A1[0].length;

## int colsB = B1[0].length;

## int numMatricesC = 2; // Cambiar según la cantidad deseada de matrices C

## float[][][] matricesC = new float[numMatricesC][rowsA][colsB];

## 

## for (int c = 0; c < numMatricesC; c++) {

## float[][] C = new float[rowsA][colsB];

## for (int i = 0; i < rowsA; i++) {

## for (int j = 0; j < colsB; j++) {

## for (int k = 0; k < colsA; k++) {

## C[i][j] += A1[i][k] \* B1[k][j];

## }

## }

## }

## matricesC[c] = C;

## mostrarMatriz(C);

## }

## 

## return matricesC;

## }

## Este código declararía un arreglo float[][][] matricesC con dos elementos, y dentro del bucle for crearía una nueva matriz float[][] C en cada iteración para almacenar el resultado de la multiplicación de matrices. Luego, la matriz C se asigna al elemento correspondiente en el arreglo matricesC.

## 2 / 2

## necesito hacer lo mismo con B1 y A1 que cuando se multiplique pase a multiplicar A1 con B2

## 2 / 2

## Para hacer lo que estás pidiendo, debes modificar el método multiplicarMatrices0() para que realice la multiplicación de A1 y B1 en la primera iteración, y en las iteraciones siguientes, realice la multiplicación de A1 y B2, y así sucesivamente.

## Aquí te muestro cómo puedes modificar el método para lograr esto:

## css

## public float[][][] multiplicarMatrices0() throws RemoteException {

## int rowsA = A1.length;

## int colsA = A1[0].length;

## int colsB = B1[0].length;

## int numMatrices = 3; // Cambiar esto al número deseado de matrices

## float[][][] matricesC = new float[numMatrices][rowsA][colsB];

## matricesC[0] = multiplicarMatrices(A1, B1); // Realizar la primera multiplicación

## for (int c = 1; c < numMatrices; c++) {

## float[][] currentB = (c % 2 == 0) ? B1 : B2; // Alternar entre B1 y B2

## matricesC[c] = multiplicarMatrices(A1, currentB); // Multiplicar A1 con la matriz B actual

## mostrarMatriz(matricesC[c]);

## }

## 

## return matricesC;

## }

## private float[][] multiplicarMatrices(float[][] A, float[][] B) {

## int rowsA = A.length;

## int colsA = A[0].length;

## int colsB = B[0].length;

## float[][] C = new float[rowsA][colsB];

## for (int i = 0; i < rowsA; i++) {

## for (int j = 0; j < colsB; j++) {

## for (int k = 0; k < colsA; k++) {

## C[i][j] += A[i][k] \* B[k][j];

## }

## }

## }

## return C;

## }

## Lo que he hecho aquí es agregar una variable numMatrices que puedes modificar para indicar cuántas matrices quieres calcular. También he agregado una nueva variable currentB que se alternará entre B1 y B2 en cada iteración, y he llamado a la función multiplicarMatrices() para realizar la multiplicación de A1 con la matriz currentB. La función multiplicarMatrices() es una función auxiliar que realiza la multiplicación de dos matrices y devuelve la matriz resultante.

## Espero que esto te ayude.

## 3 / 3

## esto que siginifica (c % 2 == 0)

## La expresión (c % 2 == 0) verifica si el valor de la variable c es un número par.

## El operador % se utiliza para obtener el resto de la división entre dos números, en este caso, el resto de la división de c entre 2. Si el resto es cero, significa que c es divisible por 2 y por lo tanto es un número par.

## En el código que me mostraste anteriormente, se utiliza esta expresión para determinar si se está calculando una matriz en una posición par o impar dentro del arreglo matricesC.

## si ahora quiero que despues de B2 se multiplique ahora A1 con B3

## Para multiplicar A1 con B3 después de multiplicar A1 con B2 en cada iteración del ciclo, puedes agregar otro ciclo for dentro del ciclo existente que recorra los diferentes valores de B. A continuación, te proporciono un ejemplo de cómo podrías hacerlo:

## scss

## public float[][][] multiplicarMatrices() throws RemoteException {

## int rowsA = A1.length;

## int colsA = A1[0].length;

## int colsB = B1[0].length;

## float[][][] matricesC = new float[3][rowsA][colsB];

## float[][] B = new float[colsA][colsB];

## for (int c = 0; c < matricesC.length; c++) {

## if (c == 0) {

## B = B1;

## } else if (c == 1) {

## B = B2;

## } else {

## B = B3;

## }

## for (int i = 0; i < rowsA; i++) {

## for (int j = 0; j < colsB; j++) {

## for (int k = 0; k < colsA; k++) {

## matricesC[c][i][j] += A1[i][k] \* B[k][j];

## }

## }

## }

## mostrarMatriz(matricesC[c]);

## }

## return matricesC;

## }

## En este ejemplo, he creado una nueva matriz B que se usará para almacenar la matriz de multiplicación correspondiente en cada iteración. En el ciclo principal, utilizo una declaración if-else para determinar qué matriz B se utilizará en cada iteración. La matriz B se inicializa a B1 en la primera iteración, B2 en la segunda iteración y B3 en la tercera y última iteración. Luego, se realiza la multiplicación de matrices para cada iteración del ciclo for y se almacena el resultado en la matriz matricesC.

## ahora lo mismo pero con A

## Para hacer lo mismo con A, podrías usar un enfoque similar, utilizando una variable entera para rastrear qué matriz se debe utilizar en cada iteración del bucle for:

## java

## public float[][] multiplicarMatrices() {

## int rowsA = A1.length;

## int colsA = A1[0].length;

## int colsB = B1[0].length;

## float[][][] matricesC = new float[3][rowsA][colsB];

## int matrizAIndex = 0; // índice de la matriz A que se utilizará en la multiplicación

## for (int c = 0; c < matricesC.length; c++) {

## int matrizBIndex = c % 3; // índice de la matriz B que se utilizará en la multiplicación

## for (int i = 0; i < rowsA; i++) {

## for (int j = 0; j < colsB; j++) {

## for (int k = 0; k < colsA; k++) {

## matricesC[c][i][j] += A[matrizAIndex][i][k] \* B[matrizBIndex][k][j];

## }

## }

## }

## 

## mostrarMatriz(matricesC[c]);

## // Actualizar el índice de la matriz A para la siguiente iteración del bucle for

## if (c % 3 == 2) {

## matrizAIndex++;

## }

## }

## 

## return matricesC[matricesC.length-1];

## }

## En este caso, el índice de la matriz A se actualiza después de cada tres iteraciones del bucle for, ya que hay tres matrices en la matriz A que se deben utilizar en la multiplicación.

## quiero modificar esta parte en donde se haga una matriz primero de 27 partes pero se este multiplicando por una matriz A dividida en 9 partes igual que la matriz B public float[][][] multiplicarMatrices0() throws RemoteException { int rowsA = A1.length; int colsA = A1[0].length; int colsB = B1[0].length; float[][][] matricesC = new float[27][rowsA][colsB]; float[][] B = new float[colsA][colsB]; for (int c = 0; c < matricesC.length; c++) { if (c == 0) { B = B1; } else if (c == 1) { B = B2; } else { B = B3; } for (int i = 0; i < rowsA; i++) { for (int j = 0; j < colsB; j++) { for (int k = 0; k < colsA; k++) { matricesC[c][i][j] += A1[i][k] \* B[k][j]; } } } mostrarMatriz(matricesC[c]); } return matricesC; }

## 2 / 2

## Si entiendo bien, deseas dividir la matriz A1 en 9 partes iguales y luego multiplicar cada una de estas partes con cada una de las 3 matrices B1, B2 y B3. En este caso, podrías dividir la matriz A1 en 9 partes utilizando un bu

## en el siguiente codigo quiero que A1 se multiplique con B1 y despues avance a B2 asi hasta B9 porfavor public float[][][] multiplicarMatrices0() throws RemoteException { int rowsA = A1.length; int colsA = A1[0].length; int colsB = B1[0].length; float[][][] matricesC = new float[27][rowsA][colsB]; float[][] B = new float[colsA][colsB]; for (int c = 0; c < matricesC.length; c++) { if (c == 0) { B = B1; } else if (c == 1) { B = B2; } else { B = B3; } for (int i = 0; i < rowsA; i++) { for (int j = 0; j < colsB; j++) { for (int k = 0; k < colsA; k++) { matricesC[c][i][j] += A1[i][k] \* B[k][j]; } } } mostrarMatriz(matricesC[c]); } return matricesC; }

## Para lograr multiplicar A1 con B1, y luego A1 con B2, y así sucesivamente hasta B9, se debe cambiar la lógica de la variable B en el ciclo for. En lugar de usar condicionales if-else, se puede usar el operador módulo (%) para iterar sobre las matrices B1 a B9 en un ciclo for interno. El código se vería así:

## css

## public float[][][] multiplicarMatrices0() throws RemoteException {

## int rowsA = A1.length;

## int colsA = A1[0].length;

## int colsB = B1[0].length;

## float[][][] matricesC = new float[27][rowsA][colsB];

## float[][] B = new float[colsA][colsB];

## for (int c = 0; c < matricesC.length; c++) {

## B = evaluarB(c % 9 + 1);

## for (int i = 0; i < rowsA; i++) {

## for (int j = 0; j < colsB; j++) {

## for (int k = 0; k < colsA; k++) {

## matricesC[c][i][j] += A1[i][k] \* B[k][j];

## }

## }

## }

## mostrarMatriz(matricesC[c]);

## }

## return matricesC;

## }

## public float[][] evaluarB(int n) {

## switch (n) {

## case 1:

## return B1;

## case 2:

## return B2;

## case 3:

## return B3;

## case 4:

## return B4;

## case 5:

## return B5;

## case 6:

## return B6;

## case 7:

## return B7;

## case 8:

## return B8;

## case 9:

## return B9;

## default:

## return null;

## }

## }

## En este caso, el método evaluarB devuelve la matriz correspondiente a cada valor de n del 1 al 9. La línea B = evaluarB(c % 9 + 1) se encarga de seleccionar la matriz adecuada en cada iteración del ciclo for principal, gracias al operador módulo que va cambiando el valor de n en cada iteración.

## en el siguiente codigo quiero que A1 se multiplique con B1 y despues avance a B2 asi hasta B9 porfavor public float[][][] multiplicarMatrices0() throws RemoteException { int rowsA = A1.length; int colsA = A1[0].length; int colsB = B1[0].length; float[][][] matricesC = new float[9][rowsA][colsB]; float[][] B = new float[colsA][colsB]; for (int c = 0; c < matricesC.length; c++) { if (c == 0) { B = B1; } else if (c == 1) { B = B2; } else { B = B3; } for (int i = 0; i < rowsA; i++) { for (int j = 0; j < colsB; j++) { for (int k = 0; k < colsA; k++) { matricesC[c][i][j] += A1[i][k] \* B[k][j]; } } } mostrarMatriz(matricesC[c]); } return matricesC; }

## Para multiplicar A1 con B1, B2, ..., B9 en ese orden, puedes hacer lo siguiente:

## Definir una matriz B que contenga todas las matrices B1, B2, ..., B9 en ese orden. Puedes hacerlo de la siguiente manera:

## css

##  float[][][] B = new float[][][] {B1, B2, B3, B4, B5, B6, B7, B8, B9};

##  Definir una matriz matricesC de tamaño 9, ya que se están multiplicando A1 con 9 matrices B.

## css

##  float[][][] matricesC = new float[9][rowsA][colsB];

##  Realizar la multiplicación de A1 con cada una de las matrices de B y almacenar los resultados en matricesC.

## scss

##  for (int c = 0; c < matricesC.length; c++) {

## float[][] currentB = B[c];

## for (int i = 0; i < rowsA; i++) {

## for (int j = 0; j < colsB; j++) {

## for (int k = 0; k < colsA; k++) {

## matricesC[c][i][j] += A1[i][k] \* currentB[k][j];

## }

## }

## }

## mostrarMatriz(matricesC[c]);

## }

##  Devolver matricesC.

## kotlin

## return matricesC;

## El código completo quedaría así:

## scss

## public float[][][] multiplicarMatrices0() throws RemoteException {

## int rowsA = A1.length;

## int colsA = A1[0].length;

## int colsB = B1[0].length;

## float[][][] B = new float[][][] {B1, B2, B3, B4, B5, B6, B7, B8, B9};

## float[][][] matricesC = new float[9][rowsA][colsB];

## for (int c = 0; c < matricesC.length; c++) {

## float[][] currentB = B[c];

## for (int i = 0; i < rowsA; i++) {

## for (int j = 0; j < colsB; j++) {

## for (int k = 0; k < colsA; k++) {

## matricesC[c][i][j] += A1[i][k] \* currentB[k][j];

## }

## }

## }

## mostrarMatriz(matricesC[c]);

## }

## return matricesC;

## }

## ayudame necesito enviarle las matrices A\_part y B\_part a mi RMIServer que tiene 2 nodos del 0 al 2 que estaran montados en azure import java.rmi.Naming; import java.util.Scanner; public class RMIClient { private static final String[] SERVERS = {"localhost", "localhost", "localhost"}; private static final int []PORT = {1099,1234,5678}; private static final String SERVICE\_NAME = "MatrixMultiplier"; public static void main(String[] args) { Scanner scanner = new Scanner(System.in); try { Interface[] multipliers = new Interface[SERVERS.length]; for (int i = 0; i < SERVERS.length; i++) { multipliers[i] = (Interface) Naming.lookup(String.format("rmi://%s:%d/%s", SERVERS[i], PORT[i], SERVICE\_NAME)); } multipliers[0].borrarMatrices(); multipliers[1].borrarMatrices(); multipliers[2].borrarMatrices(); System.out.println("Ingrese la opcion 1 o 2"); int opcion = scanner.nextInt(); if (opcion == 1) { try { int N = 9; int M = 4; float[][] A = new float[N][M]; float[][] B = new float[M][N]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } int numParts = 9; int partSize = N / numParts; for (int k = 0; k < numParts; k++) { float[][] A\_part = new float[partSize][M]; float[][] B\_part = new float[M][partSize]; int startRow = k \* partSize; for (int i = startRow; i < startRow + partSize; i++) { for (int j = 0; j < M; j++) { A\_part[i - startRow][j] = A[i][j]; B\_part[j][i - startRow] = B[j][i]; } } if (k < 3) { multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0 multipliers[0].sendMatrixB(B\_part); } else if (k < 6) { multipliers[1].sendMatrixA(A\_part); // enviar al nodo 1 multipliers[1].sendMatrixB(B\_part); } else { multipliers[2].sendMatrixA(A\_part); // enviar al nodo 2 multipliers[2].sendMatrixB(B\_part); } } int m = A.length; int n = B[0].length; int o = B.length; float[][] result = new float[m][n]; for (int i = 0; i < m; i++) { for (int j = 0; j < n; j++) { for (int k = 0; k < o; k++) { result[i][j] += A[i][k] \* B[k][j]; } } } // Mostrar matriz C System.out.println("Matriz C resultante:"); for (int i = 0; i < N; i++) { for (int j = 0; j < N; j++) { System.out.print(result[i][j] + " "); } System.out.println(); } } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } }else if (opcion == 2) { System.out.println("PROXIMAMENTE."); } else { System.out.println("Opcion invalida."); } } catch (Exception e) { System.err.println("Error: " + e.getMessage()); e.printStackTrace(); } } }

## Para enviar las matrices A\_part y B\_part a tu RMIServer, debes llamar a los métodos sendMatrixA() y sendMatrixB() de la interfaz remota correspondiente en cada nodo. Estos métodos deben recibir como parámetros las matrices A\_part y B\_part respectivamente.

## En tu código, ya tienes la lógica para determinar a qué nodo enviar cada matriz. Puedes utilizar esta lógica para llamar a los métodos correspondientes en cada iteración del ciclo.

## Aquí te muestro cómo puedes modificar tu código para enviar las matrices A\_part y B\_part a tu RMIServer:

## java

## for (int k = 0; k < numParts; k++) {

## float[][] A\_part = new float[partSize][M];

## float[][] B\_part = new float[M][partSize];

## int startRow = k \* partSize;

## for (int i = startRow; i < startRow + partSize; i++) {

## for (int j = 0; j < M; j++) {

## A\_part[i - startRow][j] = A[i][j];

## B\_part[j][i - startRow] = B[j][i];

## }

## }

## if (k < 3) {

## multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0

## multipliers[0].sendMatrixB(B\_part);

## } else if (k < 6) {

## multipliers[1].sendMatrixA(A\_part); // enviar al nodo 1

## multipliers[1].sendMatrixB(B\_part);

## } else {

## multipliers[2].sendMatrixA(A\_part); // enviar al nodo 2

## multipliers[2].sendMatrixB(B\_part);

## }

## }

## Ten en cuenta que los métodos sendMatrixA() y sendMatrixB() deben estar definidos en la interfaz remota correspondiente.

## ahora en PClass que los 2 nodos reciban lo mismo public void sendMatrixA(float[][] A) throws RemoteException { if (node == 0) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; } else if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; } else if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; } System.out.println("Nodo 0 : Matriz A1, A2, A3, A4, A5, A6, A7, A8, A9 recibida"); } else if (node == 1) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; } else if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; } else if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; } System.out.println("Nodo 1 : Matriz A1, A2, A3, A4, A5, A6, A7, A8, A9 recibida"); } else if (node == 2) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; } else if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; } else if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; } System.out.println("Nodo 2 : Matriz A1, A2, A3, A4, A5, A6, A7, A8, A9 recibida"); } }

## 3 / 3

## Para que ambos nodos reciban la misma matriz, puedes hacer lo siguiente:

## Agrega un parámetro extra a la función sendMatrixA que indique el índice del nodo que está enviando la matriz.

## Agrega un nuevo método broadcastMatrixA que tome una matriz A como entrada y llame al método sendMatrixA para cada nodo.

## Modifica el método sendMatrixA para que solo se asigne la matriz A recibida si no se ha asignado anteriormente en ninguno de los nodos.

## Aquí te muestro cómo podrías hacer estas modificaciones:

## java

## // Nuevo parámetro "senderNode" que indica el índice del nodo que está enviando la matriz

## public void sendMatrixA(float[][] A, int senderNode) throws RemoteException {

## if (A1 == null && senderNode != 0) {

## A1 = A;

## }

## else if (A2 == null && senderNode != 1) {

## A2 = A;

## }

## else if (A3 == null && senderNode != 2) {

## A3 = A;

## }

## else if (A4 == null) {

## A4 = A;

## }

## else if (A5 == null) {

## A5 = A;

## 

## quota todos los if if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; } else if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; } else if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; } System.out.println("Nodo 0 : Matriz A1, A2, A3, A4, A5, A6, A7, A8, A9 recibida");

## Los bloques de código que aparecen en los tres nodos son idénticos, por lo que se pueden agrupar y simplificar el código. Podría reemplazar todas las condiciones por un bucle for que recorra un arreglo de matrices de tamaño 9. De esta manera, se reduciría la cantidad de líneas de código y se haría más legible.

## Aquí hay un ejemplo de cómo se podría refactorizar el código:

## csharp

## public void sendMatrixA(float[][] A) throws RemoteException {

## for (int i = 0; i < 9; i++) {

## if (AArray[i] == null) {

## AArray[i] = A;

## System.out.println("Nodo " + node + " : Matriz A" + (i+1) + " recibida");

## break;

## }

## }

## }

## Donde AArray es un arreglo de matrices de tamaño 9 y node es la variable que representa el nodo actual. De esta manera, el código se simplifica y se hace más fácil de leer y mantener.

## quiero que las matrices A y B se dividan en 9 partes iguales y esta se envien a cada uno de los nodos Interface[] multipliers = new Interface[SERVERS.length]; for (int i = 0; i < SERVERS.length; i++) { multipliers[i] = (Interface) Naming.lookup(String.format("rmi://%s:%d/%s", SERVERS[i], PORT[i], SERVICE\_NAME)); } multipliers[0].borrarMatrices(); multipliers[1].borrarMatrices(); multipliers[2].borrarMatrices(); System.out.println("Ingrese la opcion 1 o 2"); int opcion = scanner.nextInt(); if (opcion == 1) { try { int N = 9; int M = 4; float[][] A = new float[N][M]; float[][] B = new float[M][N]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } int numParts = 9; int partSize = N / numParts; for (int k = 0; k < numParts; k++) { float[][] A\_part = new float[partSize][M]; float[][] B\_part = new float[M][partSize]; int startRow = k \* partSize; for (int i = startRow; i < startRow + partSize; i++) { for (int j = 0; j < M; j++) { A\_part[i - startRow][j] = A[i][j]; B\_part[j][i - startRow] = B[j][i]; } } if (k < 3) { multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0 multipliers[0].sendMatrixB(B\_part); } else if (k < 6) { multipliers[1].sendMatrixA(A\_part); // enviar al nodo 1 multipliers[1].sendMatrixB(B\_part); } else { multipliers[2].sendMatrixA(A\_part); // enviar al nodo 2 multipliers[2].sendMatrixB(B\_part); } }

## El código que proporcionas ya realiza la división de las matrices A y B en 9 partes iguales y las envía a los nodos correspondientes utilizando la interfaz de RMI.

## En particular, el código crea dos matrices A y B de tamaño 9x4 y 4x9, respectivamente, y las llena con valores. Luego, se define el número de partes en el que se dividirán las matrices (en este caso, 9) y el tamaño de cada parte.

## El código utiliza un bucle para iterar a través de las partes de las matrices y crear una submatriz de A y B para cada parte. Luego, utiliza una serie de condicionales para determinar a qué nodo se debe enviar cada submatriz, según el número de parte.

## Por ejemplo, las partes 0-2 se envían al nodo 0, las partes 3-5 se envían al nodo 1 y las partes 6-8 se envían al nodo 2. Finalmente, cada submatriz se envía a su nodo correspondiente utilizando la interfaz de RMI.

## En resumen, el código proporcionado ya realiza la división de las matrices A y B en 9 partes iguales y las envía a los nodos correspondientes utilizando RMI.

## quiero que el nodo 0 reciba las partes del 0 al 8

## Si deseas que el nodo 0 reciba todas las partes de las matrices A y B, debes cambiar la parte del código que determina a qué nodo se envía cada submatriz.

## En lugar de utilizar la lógica actual que divide las partes de la matriz en grupos de tres y los envía a nodos diferentes, puedes simplemente enviar cada submatriz al nodo 0 en cada iteración del bucle.

## Aquí está el código modificado para enviar todas las submatrices al nodo 0:

## scss

## for (int k = 0; k < numParts; k++) {

## float[][] A\_part = new float[partSize][M];

## float[][] B\_part = new float[M][partSize];

## int startRow = k \* partSize;

## for (int i = startRow; i < startRow + partSize; i++) {

## for (int j = 0; j < M; j++) {

## A\_part[i - startRow][j] = A[i][j];

## B\_part[j][i - startRow] = B[j][i];

## }

## }

## multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0

## multipliers[0].sendMatrixB(B\_part);

## }

## Con este cambio, el nodo 0 recibirá todas las partes de las matrices A y B.

## como seria la funcion sendMatrixA en PClass ?

## La función sendMatrixA en la clase PClass debería tener la misma firma que en la interfaz remota Interface. Es decir, la función debe tener el siguiente encabezado:

## java

## public void sendMatrixA(float[][] matrix) throws RemoteException;

## Además, la implementación de la función sendMatrixA en la clase PClass debe utilizar el método UnicastRemoteObject.exportObject para exportar el objeto remoto, y el método Naming.rebind para registrar el objeto remoto en el registro de RMI.

## La función sendMatrixA también debe tener el código para recibir la matriz matrix y almacenarla en una variable o estructura de datos dentro del objeto remoto PClass. Aquí está un ejemplo de cómo podría ser la implementación de la función sendMatrixA en la clase PClass:

## java

## import java.rmi.Naming;

## import java.rmi.RemoteException;

## import java.rmi.server.UnicastRemoteObject;

## public class PClass extends UnicastRemoteObject implements Interface {

## private float[][] matrixA; // variable para almacenar la matriz A

## public PClass() throws RemoteException {

## super();

## }

## @Override

## public void sendMatrixA(float[][] matrix) throws RemoteException {

## matrixA = matrix; // almacenar la matriz A recibida en la variable matrixA

## }

## public static void main(String[] args) {

## try {

## PClass obj = new PClass();

## Naming.rebind("//localhost/Interface", obj);

## System.out.println("Servidor listo.");

## } catch (Exception e) {

## System.err.println("Excepcion en Servidor: " + e.getMessage());

## e.printStackTrace();

## }

## }

## }

## En este ejemplo, la clase PClass implementa la interfaz remota Interface y define una variable matrixA para almacenar la matriz A. La función sendMatrixA simplemente asigna la matriz recibida a la variable matrixA.

## El método main crea un objeto PClass, lo exporta con el método UnicastRemoteObject.exportObject y lo registra en el registro RMI con el método Naming.rebind. Cuando se llama a Naming.rebind, el objeto remoto obj estará disponible para que los clientes lo utilicen.

## Espero que esto te ayude a implementar la función sendMatrixA en la clase PClass.

## se supone que cada nodo debe de recibir las mismas partes pero siento que no lo hace puedes checarlo en mi codigo public void sendMatrixA(float[][] A) throws RemoteException { if (node == 0) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; } else if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; } else if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; } System.out.println("Nodo 0 : Matriz A1, A2, A3, A4, A5, A6, A7, A8, A9 recibida"); } else if (node == 1) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; } else if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; } else if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; } System.out.println("Nodo 1 : Matriz A1, A2, A3, A4, A5, A6, A7, A8, A9 recibida"); } else if (node == 2) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; } else if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; } else if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; } System.out.println("Nodo 2 : Matriz A1, A2, A3, A4, A5, A6, A7, A8, A9 recibida"); } }

## En tu función sendMatrixA, cada nodo recibe las mismas partes de la matriz A ya que la función sendMatrixA es la misma en cada nodo. Lo que cambia es que cada nodo recibe una parte diferente de la matriz A.

## Por ejemplo, el nodo 0 recibirá la parte de la matriz A que va desde la fila 0 hasta la fila 2 y desde la columna 0 hasta la columna 2. El nodo 1 recibirá la parte de la matriz A que va desde la fila 0 hasta la fila 2 y desde la columna 3 hasta la columna 5. Y así sucesivamente para los demás nodos.

## En tu función sendMatrixA, cada nodo verifica si alguna de sus partes de la matriz A es nula. Si es nula, entonces asigna la parte recibida a esa variable. Si no es nula, significa que ya recibió esa parte de la matriz A y no hace nada.

## Cabe mencionar que esta forma de particionar la matriz A no es la única posible y puede depender del algoritmo que se quiera implementar. Además, esta partición asume que el número de nodos es 9, por lo que en caso de utilizar un número diferente de nodos, la partición tendría que ser modificada.

## quiero que el nodo 0 reciba todas las partes de la matrix A del 1 al 9 en sus diferentes partes public void sendMatrixA(float[][] A) throws RemoteException { if (node == 0) { if (A1 == null) { A1 = A; } else if (A2 == null) { A2 = A; } else if (A3 == null) { A3 = A; } else if (A4 == null) { A4 = A; } else if (A5 == null) { A5 = A; } else if (A6 == null) { A6 = A; } else if (A7 == null) { A7 = A; } else if (A8 == null) { A8 = A; } else if (A9 == null) { A9 = A; } System.out.println("Nodo 0 : Matriz A1, A2, A3, A4, A5, A6, A7, A8, A9 recibida"); }

## Si quieres que el nodo 0 reciba todas las partes de la matriz A del 1 al 9, debes modificar la función sendMatrixA para que maneje los diferentes casos en los que se recibe cada una de estas partes. Puedes hacerlo de la siguiente manera:

## csharp

## public void sendMatrixA(float[][] A) throws RemoteException {

## if (node == 0) {

## if (A1 == null) {

## A1 = A;

## System.out.println("Nodo 0 : Matriz A1 recibida");

## }

## else if (A2 == null) {

## A2 = A;

## System.out.println("Nodo 0 : Matriz A2 recibida");

## }

## else if (A3 == null) {

## A3 = A;

## System.out.println("Nodo 0 : Matriz A3 recibida");

## }

## else if (A4 == null) {

## A4 = A;

## System.out.println("Nodo 0 : Matriz A4 recibida");

## }

## else if (A5 == null) {

## A5 = A;

## System.out.println("Nodo 0 : Matriz A5 recibida");

## }

## else if (A6 == null) {

## A6 = A;

## System.out.println("Nodo 0 : Matriz A6 recibida");

## }

## else if (A7 == null) {

## A7 = A;

## System.out.println("Nodo 0 : Matriz A7 recibida");

## }

## else if (A8 == null) {

## A8 = A;

## System.out.println("Nodo 0 : Matriz A8 recibida");

## }

## else if (A9 == null) {

## A9 = A;

## System.out.println("Nodo 0 : Matriz A9 recibida");

## }

## }

## else if (node == 1) {

## if (A1 == null) {

## A1 = A;

## System.out.println("Nodo 1 : Matriz A1 recibida");

## }

## else if (A2 == null) {

## A2 = A;

## System.out.println("Nodo 1 : Matriz A2 recibida");

## }

## // ...

## }

## // ...

## }

## Con esta modificación, cada vez que se reciba una parte de la matriz A, se verificará en qué parte del arreglo correspondiente debe ser almacenada y se imprimirá un mensaje indicando que la parte correspondiente ha sido recibida.

## quiero hacer una funcion en PClass en donde cada vez que sendMatrixA envie una parte de la matrix A al RMIServer este la guarde en A1 y cuando reciba otra parte en A2 asi hasta A9 for (int k = 0; k < numParts; k++) { float[][] A\_part = new float[partSize][M]; float[][] B\_part = new float[M][partSize]; int startRow = k \* partSize; for (int i = startRow; i < startRow + partSize; i++) { for (int j = 0; j < M; j++) { A\_part[i - startRow][j] = A[i][j]; B\_part[j][i - startRow] = B[j][i]; } } multipliers[0].sendMatrixA(A\_part); // enviar al nodo 0

## Para lograr lo que quieres, puedes crear un contador para llevar el control de las partes de la matriz A que ya se han recibido. La función sendMatrixA podría recibir el número de parte que se está enviando y guardarla en el arreglo correspondiente (A1, A2, ..., A9) de acuerdo al número de parte que se está enviando.

## Aquí te dejo un ejemplo de cómo podrías implementar la función sendMatrixA:

## csharp

## private int countPartsReceived = 0;

## public void sendMatrixA(float[][] A, int partNumber) throws RemoteException {

## switch(partNumber) {

## case 1:

## A1 = A;

## break;

## case 2:

## A2 = A;

## break;

## case 3:

## A3 = A;

## break;

## case 4:

## A4 = A;

## break;

## case 5:

## A5 = A;

## break;

## case 6:

## A6 = A;

## break;

## case 7:

## A7 = A;

## break;

## case 8:

## A8 = A;

## break;

## case 9:

## A9 = A;

## break;

## }

## countPartsReceived++;

## System.out.println("Part " + partNumber + " of matrix A received.");

## if (countPartsReceived == 9) {

## // All parts have been received, do something with the complete matrix A

## // ...

## countPartsReceived = 0;

## }

## }

## En el código anterior, countPartsReceived lleva el control del número de partes que ya se han recibido. Cuando se recibe una parte de la matriz A, se guarda en el arreglo correspondiente según el número de parte que se esté recibiendo. Luego se incrementa el contador countPartsReceived y se imprime un mensaje en consola indicando que la parte partNumber de la matriz A ha sido recibida.

## Finalmente, si se han recibido todas las partes de la matriz A, se puede hacer algo con la matriz completa (por ejemplo, multiplicarla por la matriz B, como se hace en tu código original). En este ejemplo, simplemente se reinicia el contador countPartsReceived para que se pueda recibir otra matriz A en el futuro.

## quiero que matricesC1, matricesC2 y matricesC3 esten acomodadas en una matriz de 9x9 en donde la posicion (0,0) sea C1 y para (0,1) sea C2 asi hasta llegar a la posicion (9,9) sea C81 float[][][] matricesC1 = multipliers[0].getMatricesC1(); float[][][] matricesC2 = multipliers[1].getMatricesC2(); float[][][] matricesC3 = multipliers[2].getMatricesC3();

## 2 / 2

## Para acomodar las matrices matricesC1, matricesC2 y matricesC3 en una matriz de 9x9, puedes crear una matriz tridimensional de tamaño 9x9x3 que contenga estas matrices.

## Aquí te muestro cómo puedes hacerlo en Java:

## css

## float[][][] matrices = new float[9][9][3];

## for (int i = 0; i < 3; i++) {

## float[][][] currentMatrices;

## if (i == 0) {

## currentMatrices = matricesC1;

## } else if (i == 1) {

## currentMatrices = matricesC2;

## } else {

## currentMatrices = matricesC3;

## }

## for (int j = 0; j < 3; j++) {

## for (int k = 0; k < 3; k++) {

## matrices[j \* 3 + k][i \* 3] = currentMatrices[j][k];

## }

## }

## }

## Este código crea una matriz tridimensional matrices de tamaño 9x9x3. Luego, utiliza un bucle for para recorrer cada una de las tres matrices matricesC1, matricesC2 y matricesC3. Para cada matriz, utiliza dos bucles for para recorrer sus elementos y colocarlos en la posición correspondiente en la matriz tridimensional matrices.

## La posición en la matriz tridimensional matrices se calcula utilizando los índices j, k e i. El índice j y k se utilizan para recorrer los elementos dentro de cada matriz individual, mientras que el índice i se utiliza para recorrer las tres matrices en su conjunto.

## Para cada matriz individual, la posición inicial en la matriz tridimensional matrices se calcula utilizando la fórmula j \* 3 + k para el índice de fila y i \* 3 para el índice de columna.

## De esta manera, la posición (0,0) en la matriz tridimensional matrices contendrá la matriz matricesC1, la posición (0,1) contendrá la matriz matricesC2, y así sucesivamente hasta que la posición (8,8) contenga la matriz matricesC3.

## haber vamos por partes quiero armar solo la matricesC1 en una matriz C1 de 9 columnas por 4 filas float[][][] matricesC1 = multipliers[0].getMatricesC1();

## 2 / 2

## Claro, sin problema. Si lo que quieres es crear una matriz C1 de 9 columnas por 4 filas y asignar los valores correspondientes de la matriz multipliers[0].getMatricesC1(), puedes hacer lo siguiente:

## css

## float[][][] matricesC1 = multipliers[0].getMatricesC1();

## float[][] C1 = new float[4][9];

## for (int i = 0; i < 4; i++) {

## for (int j = 0; j < 9; j++) {

## C1[i][j] = matricesC1[i][j][0];

## }

## }

## En este código, primero se obtiene la matriz matricesC1 utilizando el método getMatricesC1() del primer elemento de la matriz multipliers. Luego se crea una nueva matriz C1 con las dimensiones deseadas (4 filas por 9 columnas).

## Después, se utiliza un bucle anidado para recorrer los elementos de la matriz matricesC1 e insertarlos en la matriz C1 en el orden correcto. En este caso, se está tomando el primer valor de cada submatriz (matricesC1[i][j][0]) y colocándolo en la posición correspondiente en la matriz C1 (C1[i][j]).

## Una vez que se completa el bucle, la matriz C1 tendrá los valores de matricesC1 y estará lista para su uso.

## ajusta este for int k=0; for(int i=0;i<9;i++){ for(int j=0;j<4;j++){ matrizC[i][j]=matricesC1[k]; k++; } }

## 2 / 2

## El código que has proporcionado no es válido, ya que estás tratando de asignar un valor de tipo float[] a una variable de tipo float. Suponiendo que matrizC es una matriz de tipo float de 9x4 y que matricesC1 es una matriz tridimensional de tipo float[][][] de dimensiones 4x9x1, puedes ajustar el código de la siguiente manera:

## scss

## int k = 0;

## for(int i = 0; i < 9; i++){

## for(int j = 0; j < 4; j++){

## matrizC[i][j] = matricesC1[j][i][0];

## k++;

## }

## }

## En este código, primero se inicializa la variable k a cero. Luego, se utiliza un bucle anidado para recorrer los elementos de la matriz matrizC. En cada iteración, se asigna el valor correspondiente de la matriz matricesC1 a la posición correspondiente de la matriz matrizC.

## En este caso, se utiliza j como índice para recorrer las dimensiones de la matriz matricesC1 y i como índice para recorrer las filas de la matriz matrizC. Además, se toma el valor matricesC1[j][i][0] para acceder al valor de la matriz tridimensional en la posición correcta y se asigna a matrizC[i][j].

## Finalmente, se incrementa la variable k en cada iteración del bucle interno para recorrer los elementos de la matriz matricesC1 en el orden correcto.

## optimiza este codigo System.out.println("Matriz C:"); int g=0; for(int control = 0;control<81;control++){ for(int i=0;i<N;i++){ for(int j=0;j<N;j++){ if(g<27){ C[i][j] = matricesC1[g][0][0]; C[i+3][j] = matricesC2[g][0][0]; C[i+6][j] = matricesC3[g][0][0]; g++; } } } } mostrarMatriz(C);

## Puedes optimizar el código reduciendo los bucles anidados y utilizando aritmética para acceder a los elementos de las matrices de manera más eficiente. Podrías hacer lo siguiente:

## scss

## System.out.println("Matriz C:");

## int g = 0;

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < N; j++) {

## C[i][j] = matricesC1[g / 3][i % 3][j % 3];

## C[i + 3][j] = matricesC2[g / 3][i % 3][j % 3];

## C[i + 6][j] = matricesC3[g / 3][i % 3][j % 3];

## g++;

## }

## }

## mostrarMatriz(C);

## En este código, primero se inicializa la variable g a cero. Luego, se utiliza un bucle anidado para recorrer las posiciones de la matriz C. En cada iteración, se utiliza aritmética modular para calcular los índices de la matriz tridimensional correspondiente y asignar el valor correspondiente a la posición correcta de la matriz C.

## En este caso, se utiliza g/3 para calcular el índice de la matriz tridimensional (ya que se están agrupando tres submatrices de 3x3 en una sola matriz de 9x3), i%3 para calcular el índice de fila dentro de cada submatriz y j%3 para calcular el índice de columna dentro de cada submatriz.

## Finalmente, se incrementa la variable g en cada iteración para recorrer los elementos de las matrices matricesC1, matricesC2 y matricesC3 en el orden correcto. Después de completar el bucle, se llama al método mostrarMatriz para mostrar la matriz C.

## calcula el checksum de la matriz C

## 2 / 2

## Para calcular el checksum de la matriz C, puedes sumar todos los elementos de la matriz y luego tomar el resto de la división de esa suma por un número grande, como 1,000,000,007. Puedes hacerlo de la siguiente manera:

## csharp

## long checksum = 0;

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < N; j++) {

## checksum += C[i][j];

## }

## }

## checksum %= 1000000007;

## System.out.println("Checksum: " + checksum);

## En este código, primero se inicializa la variable checksum a cero. Luego, se utiliza un bucle anidado para recorrer todos los elementos de la matriz C. En cada iteración, se suma el valor del elemento actual a la variable checksum.

## Después de completar el bucle, se toma el resto de la división de checksum por 1000000007 para obtener el valor final del checksum. Finalmente, se imprime el valor del checksum usando System.out.println.

## el checksum debe de ser de tipo double

## En ese caso, puedes modificar el código para utilizar una variable double en lugar de una variable long para almacenar el checksum. Puedes hacerlo de la siguiente manera:

## scss

## double checksum = 0.0;

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < N; j++) {

## checksum += (double) C[i][j];

## }

## }

## checksum %= 1000000007.0;

## System.out.println("Checksum: " + checksum);

## En este código, se cambia la declaración de la variable checksum a double. Además, dentro del bucle, se convierte el valor de cada elemento de C a double antes de sumarlo a checksum, para asegurarse de que la suma se realice como una operación de punto flotante.

## Al final del bucle, se toma el resto de la división de checksum por 1000000007.0 (ahora de tipo double) y se almacena en la variable checksum. Finalmente, se imprime el valor del checksum usando System.out.println.

## necesito calcular el checksum de estas 3 matrices cuadradas como le hago? float[][][] matricesC1 = multipliers[0].getMatricesC1(); float[][][] matricesC2 = multipliers[1].getMatricesC2(); float[][][] matricesC3 = multipliers[2].getMatricesC3();

## Para calcular el checksum de una matriz cuadrada, puedes sumar todos los valores de la matriz y luego tomar el resto de esa suma al dividirla por un número grande. Este número grande suele ser 2^32-1 o 2^64-1, dependiendo de la precisión que se requiera.

## Aquí te dejo un ejemplo de cómo podrías calcular el checksum de cada una de las matrices cuadradas en tu código en Java:

## java

## long checksumC1 = 0;

## long checksumC2 = 0;

## long checksumC3 = 0;

## for (float[][] matriz : matricesC1) {

## for (float[] fila : matriz) {

## for (float valor : fila) {

## checksumC1 += valor;

## }

## }

## }

## for (float[][] matriz : matricesC2) {

## for (float[] fila : matriz) {

## for (float valor : fila) {

## checksumC2 += valor;

## }

## }

## }

## for (float[][] matriz : matricesC3) {

## for (float[] fila : matriz) {

## for (float valor : fila) {

## checksumC3 += valor;

## }

## }

## }

## checksumC1 = checksumC1 % (long)(Math.pow(2, 32) - 1);

## checksumC2 = checksumC2 % (long)(Math.pow(2, 32) - 1);

## checksumC3 = checksumC3 % (long)(Math.pow(2, 32) - 1);

## System.out.println("Checksum C1: " + checksumC1);

## System.out.println("Checksum C2: " + checksumC2);

## System.out.println("Checksum C3: " + checksumC3);

## Este código recorre cada una de las matrices cuadradas y suma todos sus valores en las variables checksumC1, checksumC2 y checksumC3, respectivamente. Luego, se toma el resto de la división de cada checksum por 2^32-1 para obtener el valor final. Por último, se imprimen los valores de los checksums en la consola.

## Es importante tener en cuenta que este método de cálculo del checksum puede no ser adecuado para matrices grandes o muy dispersas, ya que la suma de todos los valores de la matriz puede ser muy grande y llevar a errores de desbordamiento. En esos casos, se pueden utilizar algoritmos de checksum más sofisticados.

## las 3 matrices deben de darme solo 1 checksum porfavor

## Para obtener un solo checksum para las tres matrices, puedes simplemente sumar los checksums individuales de cada matriz y luego calcular el resto de la división de esa suma por un número grande, como se hace en el ejemplo anterior. Aquí te dejo el código para hacerlo:

## java

## long checksumTotal = 0;

## for (float[][] matriz : matricesC1) {

## for (float[] fila : matriz) {

## for (float valor : fila) {

## checksumTotal += valor;

## }

## }

## }

## for (float[][] matriz : matricesC2) {

## for (float[] fila : matriz) {

## for (float valor : fila) {

## checksumTotal += valor;

## }

## }

## }

## for (float[][] matriz : matricesC3) {

## for (float[] fila : matriz) {

## for (float valor : fila) {

## checksumTotal += valor;

## }

## }

## }

## checksumTotal = checksumTotal % (long)(Math.pow(2, 32) - 1);

## System.out.println("Checksum total: " + checksumTotal);

## Este código suma todos los valores de las tres matrices cuadradas en la variable checksumTotal, y luego calcula el resto de la división de esa suma por 2^32-1 para obtener el valor final del checksum total. Por último, se imprime el valor del checksum total en la consola.

## Espero que esto te ayude a resolver tu problema.

## el checksumTotal es de tipo double

## En el código que te proporcioné anteriormente, el tipo de dato utilizado para el checksumTotal es long. Si necesitas utilizar un tipo de dato double para el checksumTotal, puedes hacerlo de la siguiente manera:

## java

## double checksumTotal = 0.0;

## for (float[][] matriz : matricesC1) {

## for (float[] fila : matriz) {

## for (float valor : fila) {

## checksumTotal += valor;

## }

## }

## }

## for (float[][] matriz : matricesC2) {

## for (float[] fila : matriz) {

## for (float valor : fila) {

## checksumTotal += valor;

## }

## }

## }

## for (float[][] matriz : matricesC3) {

## for (float[] fila : matriz) {

## for (float valor : fila) {

## checksumTotal += valor;

## }

## }

## }

## checksumTotal = checksumTotal % (Math.pow(2, 32) - 1);

## System.out.println("Checksum total: " + checksumTotal);

## En este caso, el tipo de dato utilizado para el checksumTotal es double, y la operación para calcular el resto de la división se hace utilizando Math.pow(2, 32) - 1 como divisor, que es también un valor de tipo double. De esta manera, se asegura que el resultado del checksumTotal sea un valor de punto flotante con mayor precisión.

## Espero que esta solución sea útil para ti.

## como realizo esto El cliente RMI deberá utilizar threads para invocar el método remoto multiplica\_matrices() de manera que los servidores RMI calculen los productos en paralelo.

## Para invocar el método remoto multiplica\_matrices() de manera que los servidores RMI calculen los productos en paralelo, puedes utilizar threads en el cliente RMI. El objetivo es dividir las matrices de entrada en subconjuntos y enviar cada subconjunto a un servidor diferente para que realice los cálculos de manera concurrente.

## Aquí te muestro un ejemplo de cómo implementar esto:

## java

## // Definir matrices de entrada (matricesC1, matricesC2 y matricesC3)

## // Crear un array de servidores RMI

## String[] servidores = {"localhost:1099", "localhost:1100", "localhost:1101"};

## // Crear un array de threads para almacenar las tareas a realizar por cada servidor

## Thread[] threads = new Thread[servidores.length];

## // Dividir las matrices de entrada en subconjuntos para cada servidor

## int subconjuntos = servidores.length;

## int tamanioSubconjunto = matricesC1.length / subconjuntos;

## int inicio = 0;

## int fin = tamanioSubconjunto;

## // Para cada servidor, crear un thread que invoque el método remoto para calcular los productos de matrices

## for (int i = 0; i < servidores.length; i++) {

## float[][][] submatricesC1 = Arrays.copyOfRange(matricesC1, inicio, fin);

## float[][][] submatricesC2 = Arrays.copyOfRange(matricesC2, inicio, fin);

## float[][][] submatricesC3 = Arrays.copyOfRange(matricesC3, inicio, fin);

## threads[i] = new Thread(() -> {

## try {

## // Obtener referencia del servidor RMI

## MultiplicadorMatricesRMI multiplicador = (MultiplicadorMatricesRMI) Naming.lookup("//" + servidores[i] + "/MultiplicadorMatricesRMI");

## // Invocar el método remoto para calcular los productos de matrices

## multiplicador.multiplica\_matrices(submatricesC1, submatricesC2, submatricesC3);

## } catch (Exception e) {

## e.printStackTrace();

## }

## });

## // Actualizar los índices para el siguiente subconjunto

## inicio = fin;

## fin = fin + tamanioSubconjunto;

## }

## // Iniciar los threads para cada servidor

## for (Thread thread : threads) {

## thread.start();

## }

## // Esperar a que todos los threads finalicen

## for (Thread thread : threads) {

## thread.join();

## }

## En este ejemplo, se crean subconjuntos threads (uno por cada servidor RMI) y se dividen las matrices de entrada en subconjuntos iguales para cada servidor. Luego, se crea un thread para cada servidor, que se encarga de invocar el método remoto multiplica\_matrices() del servidor correspondiente para calcular los productos de matrices. Finalmente, se inician todos los threads y se espera a que todos finalicen.

## Espero que esta solución te ayude a implementar el cálculo de productos de matrices en paralelo utilizando threads en el cliente RMI.

## te paso mi codigo, espero puedas juntar lo que me dijistes con lo mio import java.rmi.Naming; import java.util.Scanner; public class RMIClient { private static final String[] SERVERS = {"localhost", "localhost", "localhost"}; private static final int []PORT = {1099,1234,5678}; private static final String SERVICE\_NAME = "MatrixMultiplier"; public static void main(String[] args) { Scanner scanner = new Scanner(System.in); try { Interface[] multipliers = new Interface[SERVERS.length]; for (int i = 0; i < SERVERS.length; i++) { multipliers[i] = (Interface) Naming.lookup(String.format("rmi://%s:%d/%s", SERVERS[i], PORT[i], SERVICE\_NAME)); } multipliers[0].borrarMatrices(); multipliers[1].borrarMatrices(); multipliers[2].borrarMatrices(); System.out.println("Ingrese la opcion 1 o 2"); int opcion = scanner.nextInt(); if (opcion == 1) { try { int N = 9; int M = 4; float[][] A = new float[N][M]; float[][] B = new float[M][N]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } int numParts = 9; int partSize = N / numParts; for (int k = 0; k < numParts; k++) { float[][] A\_part = new float[partSize][M]; float[][] B\_part = new float[M][partSize]; int startRow = k \* partSize; for (int i = startRow; i < startRow + partSize; i++) { for (int j = 0; j < M; j++) { A\_part[i - startRow][j] = A[i][j]; B\_part[j][i - startRow] = B[j][i]; } } multipliers[0].sendMatrixA(A\_part,k); // enviar al nodo 0 multipliers[0].sendMatrixB(B\_part,k); multipliers[1].sendMatrixA(A\_part,k); // enviar al nodo 1 multipliers[1].sendMatrixB(B\_part,k); multipliers[2].sendMatrixA(A\_part,k); // enviar al nodo 2 multipliers[2].sendMatrixB(B\_part,k); } float[][][] matricesC1 = multipliers[0].getMatricesC1(); float[][][] matricesC2 = multipliers[1].getMatricesC2(); float[][][] matricesC3 = multipliers[2].getMatricesC3(); float [][] C = new float[N][N]; // Mostrar la matriz C System.out.println("Matriz C:"); int g=0; for(int i=0;i<N;i++){ for(int j=0;j<N;j++){ if(g<27){ C[i][j] = matricesC1[g][0][0]; C[i+3][j] = matricesC2[g][0][0]; C[i+6][j] = matricesC3[g][0][0]; g++; } } } mostrarMatriz(C); double checksum = 0.0; for (int i = 0; i < N; i++) { for (int j = 0; j < N; j++) { checksum += (double) C[i][j]; } } checksum %= 1000000007.0; System.out.println("Checksum: " + checksum); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } }else if (opcion == 2) { try { int N = 900; int M = 400; float[][] A = new float[N][M]; float[][] B = new float[M][N]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } int numParts = 9; int partSize = N / numParts; for (int k = 0; k < numParts; k++) { float[][] A\_part = new float[partSize][M]; float[][] B\_part = new float[M][partSize]; int startRow = k \* partSize; for (int i = startRow; i < startRow + partSize; i++) { for (int j = 0; j < M; j++) { A\_part[i - startRow][j] = A[i][j]; B\_part[j][i - startRow] = B[j][i]; } } multipliers[0].sendMatrixA(A\_part,k); // enviar al nodo 0 multipliers[0].sendMatrixB(B\_part,k); multipliers[1].sendMatrixA(A\_part,k); // enviar al nodo 1 multipliers[1].sendMatrixB(B\_part,k); multipliers[2].sendMatrixA(A\_part,k); // enviar al nodo 2 multipliers[2].sendMatrixB(B\_part,k); } float[][][] matricesC1 = multipliers[0].getMatricesC1(); float[][][] matricesC2 = multipliers[1].getMatricesC2(); float[][][] matricesC3 = multipliers[2].getMatricesC3(); checksum(matricesC1,matricesC2,matricesC3); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); } } else { System.out.println("Opcion invalida."); } } catch (Exception e) { System.err.println("Error: " + e.getMessage()); e.printStackTrace(); } } public static void mostrarMatriz(float[][] matriz) { for (int i = 0; i < matriz.length; i++) { for (int j = 0; j < matriz[0].length; j++) { System.out.print(matriz[i][j]+ " "); } System.out.println(); } System.out.println(); } public static void checksum(float[][][]matriz1,float[][][]matriz2,float[][][]matriz3){ double checksumTotal = 0.0; for (float[][] matriz : matriz1) { for (float[] fila : matriz) { for (float valor : fila) { checksumTotal += valor; } } } for (float[][] matriz : matriz2) { for (float[] fila : matriz) { for (float valor : fila) { checksumTotal += valor; } } } for (float[][] matriz : matriz3) { for (float[] fila : matriz) { for (float valor : fila) { checksumTotal += valor; } } } checksumTotal = checksumTotal % (Math.pow(2, 32) - 1); System.out.println("Checksum total: " + checksumTotal); } }

## Here's the modified code with the changes we discussed:

## java

## import java.rmi.Naming;

## import java.util.Scanner;

## public class RMIClient {

## private static final String[] SERVERS = {"localhost", "localhost", "localhost"};

## private static final int []PORT = {1099,1234,5678};

## private static final String SERVICE\_NAME = "MatrixMultiplier";

## public static void main(String[] args) {

## Scanner scanner = new Scanner(System.in);

## try {

## Interface[] multipliers = new Interface[SERVERS.length];

## for (int i = 0; i < SERVERS.length; i++) {

## multipliers[i] = (Interface) Naming.lookup(String.format("rmi://%s:%d/%s", SERVERS[i], PORT[i], SERVICE\_NAME));

## }

## multipliers[0].borrarMatrices();

## multipliers[1].borrarMatrices();

## multipliers[2].borrarMatrices();

## System.out.println("Ingrese la opcion 1 o 2");

## int opcion = scanner.nextInt();

## if (opcion == 1) {

## try {

## 

## int N = 9;

## int M = 4;

## 

## float[][] A = new float[N][M];

## float[][] B = new float[M][N];

## 

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < M; j++) {

## A[i][j] = 2 \* i + 3 \* j;

## B[j][i] = 3 \* i - 2 \* j;

## }

## }

## 

## int numParts = 9;

## int partSize = N / numParts;

## for (int k = 0; k < numParts; k++) {

## float[][] A\_part = new float[partSize][M];

## float[][] B\_part = new float[M][partSize];

## int startRow = k \* partSize;

## for (int i = startRow; i < startRow + partSize; i++) {

## for (int j = 0; j < M; j++) {

## A\_part[i - startRow][j] = A[i][j];

## B\_part[j][i - startRow] = B[j][i];

## }

## }

## 

## int serverIndex = k % SERVERS.length;

## multipliers[serverIndex].sendMatrixA(A\_part,k);

## multipliers[serverIndex].sendMatrixB(B\_part,k);

## 

## }

## float[][][] matricesC = new float[SERVERS.length][][];

## for (int i = 0; i < SERVERS.length; i++) {

## matricesC[i] = multipliers[i].getMatricesC();

## }

## float [][] C = new float[N][N];

## // Combine matricesC into C

## int g=0;

## for(int i=0;i<N;i+=partSize){

## for(int j=0;j<N;j+=partSize){

## int serverIndex = (i/partSize + j/partSize) % SERVERS.length;

## float[][] C\_part = matricesC[serverIndex][g];

## for(int a=0; a<C\_part.length; a++){

## for(int b=0; b<C\_part[0].length; b++){

## C[i+a][j+b] = C\_part[a][b];

## }

## }

## g++;

## }

## }

## // Mostrar la matriz C

## System.out.println("Matriz C:");

## mostrarMatriz(C);

## double checksum = 0.0;

## for (int i = 0; i < N;

## continua porfavor // Mostrar la matriz C System.out.println("Matriz C:"); mostrarMatriz(C);

## Claro, aquí continúa el código para mostrar la matriz C:

## csharp

## System.out.println("Matriz C:");

## mostrarMatriz(C);

## public static void mostrarMatriz(double[][] matriz) {

## for (int i = 0; i < matriz.length; i++) {

## for (int j = 0; j < matriz[0].length; j++) {

## System.out.print(matriz[i][j] + " ");

## }

## System.out.println();

## }

## }

## La función mostrarMatriz recorre la matriz e imprime cada uno de sus elementos. La variable i representa la fila y la variable j representa la columna. La función System.out.print imprime el valor de la matriz seguido de un espacio en blanco para separar los elementos en la misma fila. Después de imprimir todos los elementos en una fila, la función System.out.println agrega una nueva línea para imprimir la siguiente fila.

## continua porfavor double checksum = 0.0;

## Continuing from where we left off:

## // Calcular checksum de la matriz C double checksum = 0.0; for(int i=0; i<N; i++) { for(int j=0; j<N; j++) { checksum += C[i][j]; } } System.out.println("Checksum de la matriz C: " + checksum);

## En este código estamos calculando el checksum de la matriz C, que es simplemente la suma de todos los elementos de la matriz. Para ello, usamos un bucle anidado que recorre todas las filas y columnas de la matriz, y en cada iteración sumamos el elemento correspondiente al checksum.

## Por último, imprimimos el valor del checksum por consola, para verificar que el cálculo fue realizado correctamente.

## corrige el error porfavor float[][][] matricesC = new float[SERVERS.length][][]; float [][] C = new float[N][N]; // Combine matricesC into C int g=0; for(int i=0;i<N;i+=partSize){ for(int j=0;j<N;j+=partSize){ int serverIndex = (i/partSize + j/partSize) % SERVERS.length; float[][] C\_part = matricesC[serverIndex][g]; for(int a=0; a<C\_part.length; a++){ for(int b=0; b<C\_part[0].length; b++){ C[i+a][j+b] = C\_part[a][b]; } } g++; }

## El error en este código está en la declaración de la matriz matricesC. Actualmente se está creando una matriz tridimensional que no tiene el tamaño correcto para almacenar todas las submatrices de la matriz resultante. Se debe crear una matriz bidimensional con el tamaño adecuado.

## Aquí está el código corregido:

## css

## float[][][] matricesC = new float[SERVERS.length][partCount][partSize\*partSize];

## float[][] C = new float[N][N];

## // Combine matricesC into C

## int g = 0;

## for (int i = 0; i < N; i += partSize) {

## for (int j = 0; j < N; j += partSize) {

## int serverIndex = (i/partSize + j/partSize) % SERVERS.length;

## float[][] C\_part = new float[partSize][partSize];

## for (int a = 0; a < partSize; a++) {

## for (int b = 0; b < partSize; b++) {

## C\_part[a][b] = matricesC[serverIndex][g][a\*partSize+b];

## }

## }

## for (int a = 0; a < C\_part.length; a++) {

## for (int b = 0; b < C\_part[0].length; b++) {

## C[i+a][j+b] = C\_part[a][b];

## }

## }

## g++;

## }

## }

## En este código se ha cambiado la declaración de matricesC para crear una matriz bidimensional en lugar de tridimensional. También se ha añadido la creación de una matriz C\_part dentro del bucle para extraer cada submatriz. Esta matriz se inicializa con los elementos correspondientes de matricesC, y luego se copia a la matriz resultante C.

## como seria lo de los hilos que te pregunte para esta seccion del codigo for (int k = 0; k < numParts; k++) { float[][] A\_part = new float[partSize][M]; float[][] B\_part = new float[M][partSize]; int startRow = k \* partSize; for (int i = startRow; i < startRow + partSize; i++) { for (int j = 0; j < M; j++) { A\_part[i - startRow][j] = A[i][j]; B\_part[j][i - startRow] = B[j][i]; } } multipliers[0].sendMatrixA(A\_part,k); // enviar al nodo 0 multipliers[0].sendMatrixB(B\_part,k); multipliers[1].sendMatrixA(A\_part,k); // enviar al nodo 1 multipliers[1].sendMatrixB(B\_part,k); multipliers[2].sendMatrixA(A\_part,k); // enviar al nodo 2 multipliers[2].sendMatrixB(B\_part,k); } float[][][] matricesC1 = multipliers[0].getMatricesC1(); float[][][] matricesC2 = multipliers[1].getMatricesC2(); float[][][] matricesC3 = multipliers[2].getMatricesC3();

## Una posible solución sería crear un hilo para cada nodo que realice las siguientes acciones:

## Obtener su matriz A y B de la parte correspondiente a su nodo.

## Realizar la multiplicación de matrices y obtener su matriz C correspondiente.

## Almacenar su matriz C en una estructura de datos compartida entre los hilos.

## Para implementar esta solución, se podría crear una clase MatrixMultiplierThread que extienda de la clase Thread y reciba los siguientes parámetros en su constructor:

## La matriz A completa.

## La matriz B completa.

## El tamaño de cada parte (partSize).

## El índice del nodo correspondiente.

## Una estructura de datos compartida (por ejemplo, un arreglo tridimensional) donde se almacenan las matrices C de cada nodo.

## La clase MatrixMultiplierThread tendría un método run() que se encargue de realizar la multiplicación de matrices correspondiente y almacenar la matriz C en la estructura compartida. Además, cada instancia de MatrixMultiplierThread deberá ser iniciada con el método start() para que se ejecute en un hilo separado.

## Luego, en la sección de código que mencionaste, se podría reemplazar el código actual que envía y recibe las matrices A, B y C de cada nodo, por el siguiente:

## java

## // Crear la estructura de datos compartida

## float[][][] matricesC = new float[SERVERS.length][][];

## // Crear los hilos para cada nodo

## MatrixMultiplierThread[] multipliers = new MatrixMultiplierThread[SERVERS.length];

## for (int i = 0; i < SERVERS.length; i++) {

## multipliers[i] = new MatrixMultiplierThread(A, B, partSize, i, matricesC);

## multipliers[i].start();

## }

## // Esperar a que todos los hilos terminen

## for (int i = 0; i < SERVERS.length; i++) {

## try {

## multipliers[i].join();

## } catch (InterruptedException e) {

## e.printStackTrace();

## }

## }

## // Combinar las matrices C de cada nodo en una sola matriz C

## int g = 0;

## for (int i = 0; i < N; i += partSize) {

## for (int j = 0; j < N; j += partSize) {

## int serverIndex = (i / partSize + j / partSize) % SERVERS.length;

## float[][] C\_part = matricesC[serverIndex][g];

## for (int a = 0; a < C\_part.length; a++) {

## for (int b = 0; b < C\_part[0].length; b++) {

## C[i + a][j + b] = C\_part[a][b];

## }

## }

## g++;

## }

## }

## Con esta implementación, se estarían realizando las multiplicaciones de matrices de forma concurrente en hilos separados para cada nodo, lo que podría mejorar el rendimiento de la aplicación.

## te paso como tengo armado SERVERS private static final String[] SERVERS = {"localhost", "localhost", "localhost"}; private static final int []PORT = {1099,1234,5678}; private static final String SERVICE\_NAME = "MatrixMultiplier"; public static void main(String[] args) { Scanner scanner = new Scanner(System.in); try { Interface[] multipliers = new Interface[SERVERS.length]; for (int i = 0; i < SERVERS.length; i++) { multipliers[i] = (Interface) Naming.lookup(String.format("rmi://%s:%d/%s", SERVERS[i], PORT[i], SERVICE\_NAME)); }

## Para agregar hilos al envío de las matrices A\_part y B\_part, puedes crear una clase que extienda la clase Thread y que tome como argumentos la matriz A\_part, la matriz B\_part y el objeto Interface al cual se enviarán las matrices. Por ejemplo:

## csharp

## class MultiplierThread extends Thread {

## private float[][] A\_part;

## private float[][] B\_part;

## private Interface multiplier;

## public MultiplierThread(float[][] A\_part, float[][] B\_part, Interface multiplier) {

## this.A\_part = A\_part;

## this.B\_part = B\_part;

## this.multiplier = multiplier;

## }

## public void run() {

## int k = 0;

## multiplier.sendMatrixA(A\_part, k);

## multiplier.sendMatrixB(B\_part, k);

## }

## }

## Luego, en lugar de llamar a los métodos sendMatrixA y sendMatrixB directamente, puedes crear y ejecutar los hilos para cada objeto Interface:

## scss

## for (int k = 0; k < numParts; k++) {

## float[][] A\_part = new float[partSize][M];

## float[][] B\_part = new float[M][partSize];

## int startRow = k \* partSize;

## for (int i = startRow; i < startRow + partSize; i++) {

## for (int j = 0; j < M; j++) {

## A\_part[i - startRow][j] = A[i][j];

## B\_part[j][i - startRow] = B[j][i];

## }

## }

## Thread thread1 = new MultiplierThread(A\_part, B\_part, multipliers[0]);

## Thread thread2 = new MultiplierThread(A\_part, B\_part, multipliers[1]);

## Thread thread3 = new MultiplierThread(A\_part, B\_part, multipliers[2]);

## thread1.start();

## thread2.start();

## thread3.start();

## }

## De esta manera, se enviarán las matrices A\_part y B\_part a cada objeto Interface en hilos separados, lo que podría mejorar el rendimiento del programa si el envío de las matrices es una operación que consume mucho tiempo.

## y esto en donde lo pongo class MultiplierThread extends Thread { private float[][] A\_part; private float[][] B\_part; private Interface multiplier; public MultiplierThread(float[][] A\_part, float[][] B\_part, Interface multiplier) { this.A\_part = A\_part; this.B\_part = B\_part; this.multiplier = multiplier; } public void run() { int k = 0; multiplier.sendMatrixA(A\_part, k); multiplier.sendMatrixB(B\_part, k); } }

## Podrías utilizar la clase MultiplierThread en el siguiente bloque de código:

## css

## for (int k = 0; k < numParts; k++) {

## float[][] A\_part = new float[partSize][M];

## float[][] B\_part = new float[M][partSize];

## int startRow = k \* partSize;

## for (int i = start

## del RMIClien?

## Sí, puedes definir la clase MultiplierThread en el cliente RMI. Por ejemplo, puedes hacerlo después de la definición de la clase RMIClient. Además, debes instanciar y ejecutar los hilos dentro del bucle for que envía las matrices a los servidores. Por ejemplo:

## scss

## for (int k = 0; k < numParts; k++) {

## float[][] A\_part = new float[partSize][M];

## float[][] B\_part = new float[M][partSize];

## int startRow = k \* partSize;

## for (int i = startRow; i < startRow + partSize; i++) {

## for (int j = 0; j < M; j++) {

## A\_part[i - startRow][j] = A[i][j];

## B\_part[j][i - startRow] = B[j][i];

## }

## }

## for (int i = 0; i < SERVERS.length; i++) {

## MultiplierThread thread = new MultiplierThread(A\_part, B\_part, multipliers[i]);

## thread.start();

## }

## }

## float[][][] matricesC1 = multipliers[0].getMatricesC1();

## float[][][] matricesC2 = multipliers[1].getMatricesC2();

## float[][][] matricesC3 = multipliers[2].getMatricesC3();

## Ten en cuenta que esto creará varios hilos, uno por cada servidor, que enviarán los datos de manera concurrente. Además, necesitarás definir el método getMatricesC1 (o getMatricesC2, o getMatricesC3) en la interfaz Interface.

## que significa esto No enclosing instance of type RMIClient is accessible. Must qualify the allocation with an enclosing instance of type RMIClient (e.g. x.new A() where x is an instance of RMIClient).Java(16777237)

## Este error significa que estás tratando de crear una instancia de una clase interna desde un contexto estático sin una instancia de la clase externa.

## En otras palabras, estás tratando de crear una instancia de la clase MultiplierThread desde el método main de la clase RMIClient, que es estático. Como MultiplierThread es una clase interna de RMIClient, necesitas una instancia de RMIClient para crear una instancia de MultiplierThread.

## Para solucionar este error, debes crear primero una instancia de RMIClient y luego crear una instancia de MultiplierThread utilizando esa instancia. Por ejemplo:

## java

## RMIClient client = new RMIClient();

## MultiplierThread thread = client.new MultiplierThread(A\_part, B\_part, multiplier);

## Alternativamente, puedes hacer que MultiplierThread sea una clase externa o estática para que no requiera una instancia de la clase externa para su creación.

## en java RMI quiero cambiar este arreglo por hilos Interface[] multipliers = new Interface[SERVERS.length]; for (int i = 0; i < SERVERS.length; i++) { multipliers[i] = (Interface) Naming.lookup(String.format("rmi://%s:%d/%s", SERVERS[i], PORT[i], SERVICE\_NAME)); }

## Para cambiar el arreglo multipliers por hilos, podrías crear un arreglo de MultiplierThread y ejecutar cada hilo en paralelo para que envíe los datos a cada servidor. Algo así:

## scss

## MultiplierThread[] threads = new MultiplierThread[SERVERS.length];

## for (int i = 0; i < SERVERS.length; i++) {

## Interface multiplier = (Interface) Naming.lookup(String.format("rmi://%s:%d/%s", SERVERS[i], PORT[i], SERVICE\_NAME));

## threads[i] = new MultiplierThread(A\_part, B\_part, multiplier);

## threads[i].start();

## }

## // Esperar a que todos los hilos terminen

## for (MultiplierThread thread : threads) {

## try {

## thread.join();

## } catch (InterruptedException e) {

## e.printStackTrace();

## }

## }

## En este caso, se crea un arreglo de MultiplierThread, donde cada hilo recibe como parámetros la matriz A, la matriz B y la instancia de la interfaz del servidor correspondiente. Luego se inicia cada hilo con el método start(), para que se ejecuten en paralelo. Por último, se espera a que todos los hilos terminen con el método join().

## y aqui que modifico float[][][] matricesC1 = multipliers[0].getMatricesC1(); float[][][] matricesC2 = multipliers[1].getMatricesC2(); float[][][] matricesC3 = multipliers[2].getMatricesC3();

## Puedes modificar la obtención de las matrices C a través de los objetos Interface que se guardan en el arreglo multipliers. En lugar de llamar a los métodos getMatricesC1, getMatricesC2 y getMatricesC3 en cada objeto Interface, puedes crear un ciclo y ejecutar los métodos getMatricesC de cada objeto en un hilo separado.

## Aquí hay un ejemplo de cómo hacerlo:

## scss

## List<Thread> threads = new ArrayList<>();

## float[][][] matricesC = new float[SERVERS.length][][];

## for (int i = 0; i < SERVERS.length; i++) {

## final int index = i;

## Thread thread = new Thread(() -> matricesC[index] = multipliers[index].getMatricesC());

## threads.add(thread);

## thread.start();

## }

## for (Thread thread : threads) {

## try {

## thread.join();

## } catch (InterruptedException e) {

## e.printStackTrace();

## }

## }

## Este código crea una lista de hilos y luego recorre cada objeto Interface en el arreglo multipliers. Para cada objeto, se crea un nuevo hilo que ejecuta el método getMatricesC y guarda el resultado en la posición correspondiente del arreglo matricesC.

## Una vez que se han creado todos los hilos, el código espera a que todos los hilos terminen ejecutándose utilizando el método join. Finalmente, cuando todos los hilos han terminado, el arreglo matricesC contiene todas las matrices C calculadas en los diferentes servidores.

## agrega hilos solo en esta parte del codigo y que todo lo demas si haga correctamente su funcionamiento Interface[] multipliers = new Interface[SERVERS.length]; for (int i = 0; i < SERVERS.length; i++) { multipliers[i] = (Interface) Naming.lookup(String.format("rmi://%s:%d/%s", SERVERS[i], PORT[i], SERVICE\_NAME)); } multipliers[0].borrarMatrices(); multipliers[1].borrarMatrices(); multipliers[2].borrarMatrices(); try { int N = 900; int M = 400; float[][] A = new float[N][M]; float[][] B = new float[M][N]; for (int i = 0; i < N; i++) { for (int j = 0; j < M; j++) { A[i][j] = 2 \* i + 3 \* j; B[j][i] = 3 \* i - 2 \* j; } } int numParts = 9; int partSize = N / numParts; for (int k = 0; k < numParts; k++) { float[][] A\_part = new float[partSize][M]; float[][] B\_part = new float[M][partSize]; int startRow = k \* partSize; for (int i = startRow; i < startRow + partSize; i++) { for (int j = 0; j < M; j++) { A\_part[i - startRow][j] = A[i][j]; B\_part[j][i - startRow] = B[j][i]; } } multipliers[0].sendMatrixA(A\_part,k); // enviar al nodo 0 multipliers[0].sendMatrixB(B\_part,k); multipliers[1].sendMatrixA(A\_part,k); // enviar al nodo 1 multipliers[1].sendMatrixB(B\_part,k); multipliers[2].sendMatrixA(A\_part,k); // enviar al nodo 2 multipliers[2].sendMatrixB(B\_part,k); } float[][][] matricesC1 = multipliers[0].getMatricesC1(); float[][][] matricesC2 = multipliers[1].getMatricesC2(); float[][][] matricesC3 = multipliers[2].getMatricesC3(); checksum(matricesC1,matricesC2,matricesC3); } catch (Exception e) { System.err.println("Error: " + e.getMessage()); }

## 3 / 3

## Puedes modificar el código de la siguiente manera para utilizar hilos:

## java

## Interface[] multipliers = new Interface[SERVERS.length];

## Thread[] threads = new Thread[SERVERS.length];

## for (int i = 0; i < SERVERS.length; i++) {

## multipliers[i] = (Interface) Naming.lookup(String.format("rmi://%s:%d/%s", SERVERS[i], PORT[i], SERVICE\_NAME));

## }

## multipliers[0].borrarMatrices();

## multipliers[1].borrarMatrices();

## multipliers[2].borrarMatrices();

## try {

## int N = 900;

## int M = 400;

## float[][] A = new float[N][M];

## float[][] B = new float[M][N];

## for (int i = 0; i < N; i++) {

## for (int j = 0; j < M; j++) {

## A[i][j] = 2 \* i + 3 \* j;

## B[j][i] = 3 \* i - 2 \* j;

## }

## }

## int numParts = 9;

## int partSize = N / numParts;

## for (int k = 0; k < numParts; k++) {

## float[][] A\_part = new float[partSize][M];

## float[][] B\_part = new float[M][partSize];

## int startRow = k \* partSize;

## for (int i = startRow; i < startRow + partSize; i++) {

## for (int j = 0; j < M; j++) {

## A\_part[i - startRow][j] = A[i][j];

## B\_part[j][i - startRow] = B[j][i];

## }

## }

## for (int i = 0; i < SERVERS.length; i++) {

## threads[i] = new MultiplierThread(A\_part, B\_part, multipliers[i]);

## threads[i].start();

## }

## for (int i = 0; i < SERVERS.length; i++) {

## threads[i].join();

## }

## }

## float[][][] matricesC1 = multipliers[0].getMatricesC1();

## float[][][] matricesC2 = multipliers[1].getMatricesC2();

## float[][][] matricesC3 = multipliers[2].getMatricesC3();

## checksum(matricesC1, matricesC2, matricesC3);

## } catch (Exception e) {

## System.err.println("Error: " + e.getMessage());

## }

## En este nuevo código, primero se crean los objetos de los servidores en un bucle for, igual que en el código original. Después, se crea un arreglo de hilos y se inicializa con una instancia de la clase MultiplierThread para cada servidor. Cada hilo recibe un objeto de la interfaz, una parte de la matriz A y una parte de la matriz B.

## Luego se inician los hilos con el método start() y se espera a que todos los hilos terminen con el método join(). Este proceso se realiza para cada parte de la matriz. Finalmente, se recuperan las matrices resultantes de cada servidor y se realiza la comprobación del checksum.

## como es la funcion MultiplierThread

## La función MultiplierThread es una clase que extiende la clase Thread y que tiene como objetivo enviar las matrices A\_part y B\_part a través de una instancia de la interfaz Interface mediante la invocación del método sendMatrixA y sendMatrixB. Aquí está el código de la clase MultiplierThread:

## csharp

## class MultiplierThread extends Thread {

## private float[][] A\_part;

## private float[][] B\_part;

## private Interface multiplier;

## private int k;

## public MultiplierThread(float[][] A\_part, float[][] B\_part, Interface multiplier, int k) {

## this.A\_part = A\_part;

## this.B\_part = B\_part;

## this.multiplier = multiplier;

## this.k = k;

## }

## public void run() {

## multiplier.sendMatrixA(A\_part, k);

## multiplier.sendMatrixB(B\_part, k);

## }

## }

## El constructor de esta clase toma tres argumentos: A\_part, B\_part y multiplier. Estos argumentos representan las matrices A y B que se van a enviar y la instancia de la interfaz Interface a través de la cual se van a enviar. También hay un argumento adicional k, que representa el índice de la partición actual.

## El método run se ejecutará cuando se inicie la instancia de MultiplierThread. Este método invoca los métodos sendMatrixA y sendMatrixB en la instancia de la interfaz Interface pasada como argumento, pasando las matrices A\_part y B\_part como parámetros, junto con el índice de partición actual k.

## como quedaria en la Interface, agregalo en el siguiente codigo import java.rmi.Remote; import java.rmi.RemoteException; public interface Interface extends Remote { void sendMatrixA(float[][] A, int partNumber) throws RemoteException; void sendMatrixB(float[][] B, int par) throws RemoteException; void borrarMatrices() throws RemoteException; float[][][] getMatricesC1() throws RemoteException; float[][][] getMatricesC2() throws RemoteException; float[][][] getMatricesC3() throws RemoteException; }

## Para agregar la funcionalidad del hilo en la interfaz, podrías agregar el método sendMatricesAB que recibiría las matrices A y B y el número de la parte correspondiente, y se encargaría de enviar ambas matrices a través de la llamada a los métodos sendMatrixA y sendMatrixB de la interfaz. El código quedaría así:

## java

## import java.rmi.Remote;

## import java.rmi.RemoteException;

## public interface Interface extends Remote {

## void sendMatrixA(float[][] A, int partNumber) throws RemoteException;

## void sendMatrixB(float[][] B, int par) throws RemoteException;

## void borrarMatrices() throws RemoteException;

## float[][][] getMatricesC1() throws RemoteException;

## float[][][] getMatricesC2() throws RemoteException;

## float[][][] getMatricesC3() throws RemoteException;

## 

## void sendMatricesAB(float[][] A, float[][] B, int partNumber) throws RemoteException;

## }

## Luego, en la implementación de la interfaz en el servidor, tendrías que agregar el cuerpo del método sendMatricesAB y llamar a los métodos correspondientes sendMatrixA y sendMatrixB desde dentro de la ejecución del hilo.

## me marca un error el RMIClient esta linea de codigo threads[i] = new MultiplierThread(A\_part, B\_part, multipliers[i]);

## El error puede ser debido a que la clase MultiplierThread no ha sido importada en el archivo que contiene la clase RMIClient. Asegúrate de tener la siguiente línea de código al inicio del archivo:

## java

## import <ruta\_del\_paquete>.MultiplierThread;

## Reemplaza <ruta\_del\_paquete> con la ruta completa del paquete que contiene la clase MultiplierThread.

## Si la clase MultiplierThread ya ha sido importada, verifica que la clase esté definida en el mismo paquete o que esté en un paquete diferente pero público. Si la clase está en un paquete diferente y no es pública, no podrás acceder a ella desde el código de la clase RMIClient.

## donde puedo agregar esa class si solo quiero que haya 4 archivos RMIServer, RMIClient, PClass y Interface

## Puedes agregar la clase MultiplierThread en el archivo RMIClient.java, ya que esta clase se utiliza específicamente en ese archivo para manejar hilos. Simplemente agrega la definición de la clase MultiplierThread dentro del archivo RMIClient.java, debajo de la definición de la clase RMIClient. Asegúrate de importar las clases necesarias si las estás utilizando dentro de MultiplierThread.

## necesito un hilo para esto multipliers[0].borrarMatrices(); multipliers[1].borrarMatrices(); multipliers[2].borrarMatrices();

## Puedes crear un hilo que ejecute el método borrarMatrices() para cada objeto en el arreglo multipliers. Podría ser algo así:

## scss

## Thread borrarMatricesThread = new Thread(new Runnable() {

## public void run() {

## for (Interface multiplier : multipliers) {

## try {

## multiplier.borrarMatrices();

## } catch (RemoteException e) {

## e.printStackTrace();

## }

## }

## }

## });

## borrarMatricesThread.start();

## Esto creará un hilo que ejecutará el método borrarMatrices() para cada objeto en el arreglo multipliers.

## se podra hacer hilo con esto float[][][] matricesC1 = multipliers[0].getMatricesC1(); float[][][] matricesC2 = multipliers[1].getMatricesC2(); float[][][] matricesC3 = multipliers[2].getMatricesC3(); for (int i = 0; i < SERVERS.length; i++) { threads[i] = new MultiplierThread(multipliers[i]); threads[i].start(); } for (int i = 0; i < SERVERS.length; i++) { threads[i].join(); } checksum(matricesC1,matricesC2,matricesC3);

## Yes, you can create a thread to execute the code that you have provided. The code creates three float 3D arrays, initializes them with the matrices from the "multipliers" array and then creates and starts multiple threads to perform some computations on these matrices. The threads are then joined to wait for them to finish executing before proceeding to call the "checksum" method.

## If you want to create a thread to execute this code, you can simply put the code inside the "run" method of a class that extends the "Thread" class, like this:

## scss

## public class MyThread extends Thread {

## @Override

## public void run() {

## float[][][] matricesC1 = multipliers[0].getMatricesC1();

## float[][][] matricesC2 = multipliers[1].getMatricesC2();

## float[][][] matricesC3 = multipliers[2].getMatricesC3();

## for (int i = 0; i < SERVERS.length; i++) {

## threads[i] = new MultiplierThread(multipliers[i]);

## threads[i].start();

## }

## for (int i = 0; i < SERVERS.length; i++) {

## threads[i].join();

## }

## checksum(matricesC1,matricesC2,matricesC3);

## }

## }

## Then, you can create an instance of the "MyThread" class and start it, like this:

## scss

## MyThread myThread = new MyThread();

## myThread.start();

## This will create a new thread that will execute the code inside the "run" method of the "MyThread" class.

## esto es un hilo y lo que quiero es calcular un solo checksum class MultiplierThread extends Thread { private Interface multiplier; private int k; public MultiplierThread(Interface multiplier,int k) { this.k = k; this.multiplier = multiplier; } public void run() { try { if(k==0){ float[][][] matricesC1 = multiplier.getMatricesC1(); } else if(k==1){ float[][][] matricesC2 = multiplier.getMatricesC2(); }else if(k==2){ float[][][] matricesC3 = multiplier.getMatricesC3(); } } catch (RemoteException e) { // TODO Auto-generated catch block e.printStackTrace(); } } }

## 2 / 2

## It seems that you are trying to calculate a single checksum in a multi-threaded environment using the class "MultiplierThread".

## If that's the case, you can modify the "MultiplierThread" class to calculate the checksum for a given set of matrices. Here's how you can do it:

## csharp

## public class MultiplierThread extends Thread {

## private Interface multiplier;

## private int k;

## private float[][][] matrices;

## public MultiplierThread(Interface multiplier, int k, float[][][] matrices) {

## this.multiplier = multiplier;

## this.k = k;

## this.matrices = matrices;

## }

## public void run() {

## try {

## // Perform the multiplication for the given set of matrices

## multiplier.multiply(matrices);

## } catch (RemoteException e) {

## e.printStackTrace();

## }

## }

## }

## Then, you can create multiple instances of "MultiplierThread" to calculate the checksum for each set of matrices:

## scss

## float[][][] matricesC1 = multipliers[0].getMatricesC1();

## float[][][] matricesC2 = multipliers[1].getMatricesC2();

## float[][][] matricesC3 = multipliers[2].getMatricesC3();

## // Create three threads to calculate the checksum for each set of matrices

## MultiplierThread thread1 = new MultiplierThread(multipliers[0], 0, matricesC1);

## MultiplierThread thread2 = new MultiplierThread(multipliers[1], 1, matricesC2);

## MultiplierThread thread3 = new MultiplierThread(multipliers[2], 2, matricesC3);

## // Start the threads

## thread1.start();

## thread2.start();

## thread3.start();

## // Wait for all the threads to finish

## try {

## thread1.join();

## thread2.join();

## thread3.join();

## } catch (InterruptedException e) {

## e.printStackTrace();

## }

## // Calculate the checksum for the matrices

## checksum(matricesC1, matricesC2, matricesC3);

## This code will create three threads, each of which will calculate the multiplication for one set of matrices. Once all the threads have finished, the "checksum" method will be called to calculate the final checksum for all the matrices.

## quiero crear un hilo que vaya en paralelo que cumpla con esto float[][][] matricesC1 = multipliers[0].getMatricesC1(); float[][][] matricesC2 = multipliers[1].getMatricesC2(); float[][][] matricesC3 = multipliers[2].getMatricesC3();

## To create a thread that runs in parallel and performs the three operations you have mentioned, you can do the following:

## scss

## public class MatrixThread extends Thread {

## private Interface multiplier;

## private int matrixNumber;

## private float[][][] matrices;

## public MatrixThread(Interface multiplier, int matrixNumber) {

## this.multiplier = multiplier;

## this.matrixNumber = matrixNumber;

## }

## public float[][][] getMatrices() {

## return matrices;

## }

## public void run() {

## try {

## if(matrixNumber == 0) {

## matrices = multiplier.getMatricesC1();

## } else if(matrixNumber == 1) {

## matrices = multiplier.getMatricesC2();

## } else if(matrixNumber == 2) {

## matrices = multiplier.getMatricesC3();

## }

## } catch(RemoteException e) {

## e.printStackTrace();

## }

## }

## }

## In this implementation, MatrixThread extends Thread, takes an Interface object as input, and has a matrixNumber parameter to identify which matrix set is being processed. The getMatrices() method is used to return the matrices obtained by the thread.

## Now, to run the thread in parallel, you can create three MatrixThread objects and run them concurrently like this:

## scss

## // Create the threads

## MatrixThread thread1 = new MatrixThread(multipliers[0], 0);

## MatrixThread thread2 = new MatrixThread(multipliers[1], 1);

## MatrixThread thread3 = new MatrixThread(multipliers[2], 2);

## // Start the threads

## thread1.start();

## thread2.start();

## thread3.start();

## // Wait for the threads to finish

## try {

## thread1.join();

## thread2.join();

## thread3.join();

## } catch(InterruptedException e) {

## e.printStackTrace();

## }

## // Get the matrices calculated by the threads

## float[][][] matricesC1 = thread1.getMatrices();

## float[][][] matricesC2 = thread2.getMatrices();

## float[][][] matricesC3 = thread3.getMatrices();

## // Perform the checksum

## checksum(matricesC1, matricesC2, matricesC3);

## This code creates three MatrixThread objects, each with a different matrixNumber value, then starts all three threads simultaneously using start(). After starting the threads, the program waits for them to finish with join(), which ensures that all three threads have completed execution before moving on to the next step. Finally, the three sets of matrices are retrieved by calling getMatrices() on each MatrixThread object, and the checksum function is called on those matrices.

## se podra simplificar todo esto class SendThread extends Thread { private float[][] A\_part; private float[][] B\_part; private Interface multiplier; private int k; public SendThread(float[][] A\_part, float[][] B\_part, Interface multiplier, int k) { this.A\_part = A\_part; this.B\_part = B\_part; this.multiplier = multiplier; this.k = k; } public void run() { try { multiplier.sendMatrixA(A\_part, k); } catch (RemoteException e) { // TODO Auto-generated catch block e.printStackTrace(); } try { multiplier.sendMatrixB(B\_part, k); } catch (RemoteException e) { // TODO Auto-generated catch block e.printStackTrace(); } } } class VaciarMatrices extends Thread { private Interface multiplier; public VaciarMatrices(Interface multiplier) { this.multiplier = multiplier; } public void run() { try { multiplier.borrarMatrices(); } catch (RemoteException e) { // TODO Auto-generated catch block e.printStackTrace(); } } } class MatrixThread extends Thread { private Interface multiplier; private int matrixNumber; private float[][][] matrices; public MatrixThread(Interface multiplier, int matrixNumber) { this.multiplier = multiplier; this.matrixNumber = matrixNumber; } public float[][][] getMatrices() { return matrices; } public void run() { try { if(matrixNumber == 0) { matrices = multiplier.getMatricesC1(); } else if(matrixNumber == 1) { matrices = multiplier.getMatricesC2(); } else if(matrixNumber == 2) { matrices = multiplier.getMatricesC3(); } } catch(RemoteException e) { e.printStackTrace(); } } }

## Sí, se puede simplificar el código. En lugar de tener tres clases diferentes para enviar las matrices, vaciar las matrices y obtener las matrices, se pueden combinar en una sola clase de hilo. Aquí hay una posible implementación:

## scss

## class MatrixThread extends Thread {

## private Interface multiplier;

## private int matrixNumber;

## private float[][][] matrices;

## private float[][] A\_part;

## private float[][] B\_part;

## public MatrixThread(Interface multiplier, int matrixNumber, float[][] A\_part, float[][] B\_part) {

## this.multiplier = multiplier;

## this.matrixNumber = matrixNumber;

## this.A\_part = A\_part;

## this.B\_part = B\_part;

## }

## public float[][][] getMatrices() {

## return matrices;

## }

## public void run() {

## try {

## if (matrixNumber == 0) {

## multiplier.sendMatrixA(A\_part, matrixNumber);

## multiplier.sendMatrixB(B\_part, matrixNumber);

## matrices = multiplier.getMatricesC1();

## } else if (matrixNumber == 1) {

## multiplier.sendMatrixA(A\_part, matrixNumber);

## multiplier.sendMatrixB(B\_part, matrixNumber);

## matrices = multiplier.getMatricesC2();

## } else if (matrixNumber == 2) {

## multiplier.sendMatrixA(A\_part, matrixNumber);

## multiplier.sendMatrixB(B\_part, matrixNumber);

## matrices = multiplier.getMatricesC3();

## }

## multiplier.borrarMatrices();

## } catch (RemoteException e) {

## e.printStackTrace();

## }

## }

## }

## En esta implementación, la clase de hilo MatrixThread envía las matrices A y B, obtiene las matrices C correspondientes y luego vacía las matrices. Esto reduce la cantidad de clases de hilo necesarias y simplifica el código en general.

## como seria esto MatrixThread thread1 = new MatrixThread(multipliers[0], 0);

## La línea MatrixThread thread1 = new MatrixThread(multipliers[0], 0); crea un objeto de la clase MatrixThread que se ejecutará en el hilo actual. El constructor de MatrixThread toma dos argumentos: un objeto Interface llamado multiplier y un número entero llamado matrixNumber.

## En este caso, se crea un objeto MatrixThread llamado thread1 con el primer objeto Interface en el arreglo multipliers (multipliers[0]) y el número 0. Esto significa que el hilo ejecutará el método run() de la clase MatrixThread para obtener las matrices de la multiplicación de matrices correspondientes a la entrada C1.

## AZURE:

## Creacion del nodo 1:

## 1.- Abrimos AZURE y damos click al boton de crear.

## 2.- Seleccionamos UBUNTU 18.08 y el nombre del servidor

## 3.-Agregamos usuario y contraseña

## 4.- Le damos a disco y aquí escogemos el tipo de disco HDD

## 5.- En la opcion de Disco y Administracion no movemos nada hasta la ventana de Monitoreo. Y Seleccionamos Desactivar la funcion

## 6.- Avanzamos hasta Revisar y Crear y le damos a la opcion de Crear y esperamos hasta que nos diga que todo esta en orden

## PRUEBAS:

Conclusiones

## Para concluir puedo decir que esta Tarea fue mas sencilla que las anteriores, gracias al uso del chatGPT que ahora con mas experiencia en su uso y saber con que palabras dar una consulta correcta se logro el codigo y su correcta ejecucion en esta Tarea, tambien quiero aclarar que AZURE ya con los 4 programas anteriores se pudo manipular y hacer uso correcto y mas sencillo para la comunicación entre las redes ademas de que ahora no me tarde tanto en la conexión y por su puesto en las pruebas realizadas estas si lograban enviar correctamente la informacion de un servidor a otro.

.