Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт

Комп‘ютерного практикуму № 6 з дисципліни
«Технології паралельних та розподілених обчислень»

«Розробка паралельного алгоритму множення матриць з використанням MPI-методів обміну повідомленнями «один-до-одного» та дослідження його ефективності»

Виконав(ла) ІП-01 Галько М.В. _____
(шифр, прізвище, ім'я,

Перевірив(ла) Степенко І. В. _____
(прізвище, ім'я, по

Київ 2023

1. Завдання:

1. Ознайомитись з методами блокуючого та неблокуючого обміну повідомленнями типу point-to-point (див. лекцію та документацію стандарту MPI).

2. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів блокуючого обміну повідомленнями (лістинг 1). **30 балів**.

3. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів неблокуючого обміну повідомленнями. **30 балів.**

4. Дослідити ефективність розподіленого обчислення алгоритму множення матриць при збільшенні розміру матриць та при збільшенні кількості вузлів, на яких здійснюється запуск програми. Порівняйте ефективність алгоритму при використанні блокуючих та неблокуючих методів обміну повідомленнями. **40 балів.**

2. Хід роботи

Для реалізації алгоритму паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів блокуючого обміну повідомленнями я створила клас BlockingMPI, в якому в методі main() і буде виконуватись операція множення.

Метод main виконує наступні операції:

1. У методі main() спочатку генеруються випадкові матриці matrixA та matrixB певного розміру.
2. Ініціалізується MPI за допомогою MPI.Init(args).
3. Отримуються кількість процесів та ідентифікатор поточного процесу.
4. Розраховується кількість робочих процесів.
5. Якщо кількість процесів менше 2, то програма припиняє роботу за допомогою MPI.COMM_WORLD.Abort(1) та exit(1).

6. Якщо поточний процес є головним (taskID == MASTER_ID), виконується головний процес, який викликає метод masterProcess().
    a) Головний процес розподіляє роботу між робочими процесами. Він обчислює, які рядки матриці A будуть оброблятися кожним робочим процесом і надсилає їм відповідні початкові та кінцеві індекси рядків.
    b) Головний процес також надсилає робочим процесам відповідні підматриці з матриць A та B за допомогою операції MPI.COMM_WORLD.Send().
    c) Після цього головний процес очікує отримання результатів від робочих процесів за допомогою операції MPI.COMM_WORLD.Recv().
    d) Отримані підматриці результату збираються в кінцеву матрицю результату resultMatrix.
7. Якщо поточний процес є робочим процесом (не головний), виконується робочий процес, який викликає метод workerProcess().
    a) Робочий процес отримує від головного процесу початкові та кінцеві індекси рядків, які він буде обробляти, за допомогою операції Recv().
    b) Робочий процес також отримує підматриці A та B за допомогою операції Recv().
    c) Після отримання даних робочий процес виконує множення своєї підматриці на матрицю B і отримує підматрицю результату resultMatrix.
    d) Результат надсилається головному процесу за допомогою операції Send().
8. Після завершення роботи всіх процесів, викликається MPI.Finalize() для завершення роботи MPI.

Програма виводить час виконання, розмір матриці та кількість робочих процесів на головний процес.

Приклад роботи програми для блокуючого MPI:

```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
MPJ Express (0.44) is started in the multicore configuration
BlockingMPI
Total time: 4827 ms
Matrix size: 1500x1500
Workers count: 7

Process finished with exit code 0
```

Для реалізації алгоритму множення матриць з використанням неблокуючого обміну повідомленнями був створений клас NonBlockingMPI, який аналогічно до класу BlockinMPI, має метод main() в якому в свою чергу виконується операція множення.

Метод main() виконує наступні операції:

1. У методі main() спочатку генеруються випадкові матриці matrixA та matrixB розміром 1000x1000.
2. Ініціалізується MPI за допомогою MPI.Init(args).
3. Отримуються кількість процесів countTasks та ідентифікатор поточного процесу taskID.
4. Розраховується кількість робочих процесів workers (кількість процесів, виключаючи головний).
5. Якщо кількість процесів менше 2, то програма припиняє роботу за допомогою MPI.COMM_WORLD.Abort(1) та exit(1).
6. Якщо поточний процес є головним (taskID == MASTER_ID), виконується головний процес, який викликає метод masterProcess().

    a) Головний процес розподіляє роботу між робочими процесами. Він обчислює, які рядки матриці A будуть оброблятися кожним робочим процесом і надсилає їм відповідні початкові та кінцеві індекси рядків.
    b) Головний процес також надсилає робочим процесам відповідні підматриці з матриць A та B за допомогою неблокуючих операцій Isend().
    c) Після цього головний процес отримує результати від робочих процесів за допомогою операцій Irecv().

d) Отримані підматриці результату збираються в кінцеву матрицю результату resultMatrix.

7. Якщо поточний процес є робочим процесом (не головний), виконується робочий процес, який викликає метод workerProcess().

   a) Робочий процес отримує від головного процесу початкові та кінцеві індекси рядків, які він буде обробляти, за допомогою неблокуючих операцій Irecv().
   b) Він також отримує відповідні підматриці A та B за допомогою неблокуючих операцій Irecv().
   c) Після отримання даних робочий процес виконує множення своєї підматриці на матрицю B і отримує підматрицю результату resultMatrix.
   d) Результат надсилається головному процесу за допомогою неблокуючої операції Isend().

8. Після завершення роботи всіх процесів викликається MPI.Finalize().

Приклад роботи програми для неблокуючого MPI:

```
"C:\Program Files\Java\jdk-20\bin\java.exe" ...
MPJ Express (0.44) is started in the multicore configuration
NonBlockingMPI
Total time: 3490 ms
Matrix size: 1500x1500
Workers count: 7

Process finished with exit code 0
```

В ході дослідження ефективності використання блокуючого та неблокуючого MPI було проведено ряд тестів з різними розмірностями матриць та різною кількістю робочих процесів. Для всіх тестів було проведена достатня кількість спроб, щоб вважати їх результати правдивими

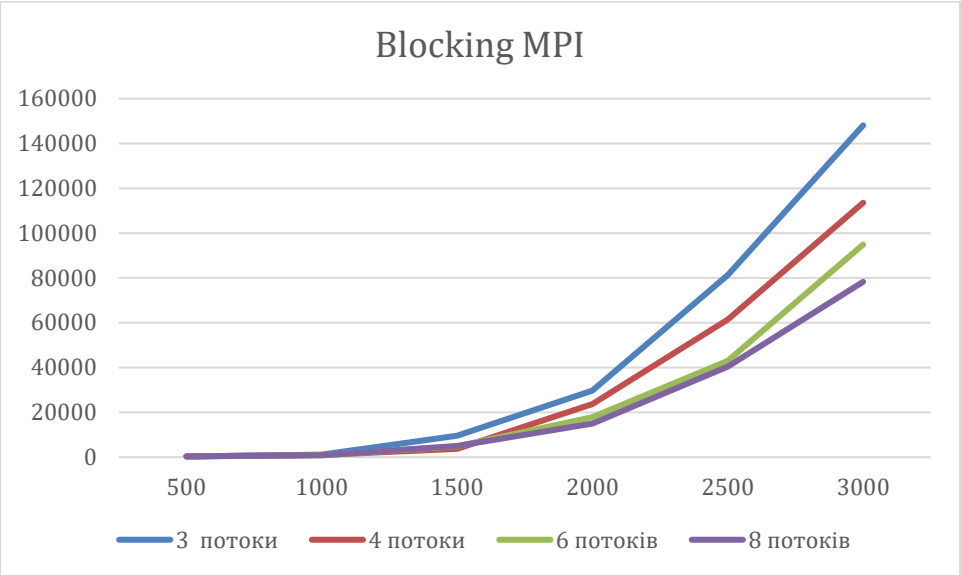
Були отримані наступні дані:
Для блокуючого MPI

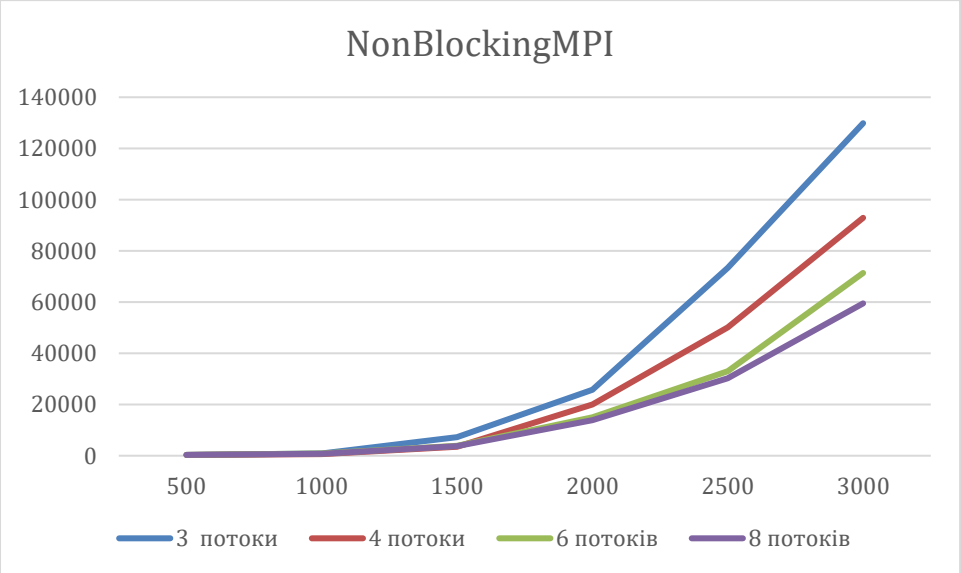
Розмір матриці	3 потоки	4 потоки	6 потоків	8 потоків
500	316	294	356	363
1000	1192	940	842	810
1500	9502	3692	4842	4925
2000	29738	23644	17817	15011
2500	81216	61425	42997	40474
3000	148091	113550	94905	78260

Для неблокуючого MPI

Розмір матриці	3 потоки	4 потоки	6 потоків	8 потоків
500	293	278	321	330
1000	888	664	817	715
1500	7254	3552	3839	3746
2000	25694	19941	14949	13889
2500	73416	50145	32985	30279
3000	129860	92906	71380	59490

Можемо також представити дані графічно:





3. Висновок

У даній роботі проведено порівняльний аналіз роботи блокуючого та неблокуючого підходів до MPI для множення матриць на різних розмірах матриць та з різною кількістю процесорів.

Загальною тенденцією, яку можна спостерігати з результатів, є те, що неблокуючий MPI зазвичай показує кращі часові характеристики порівняно з блокуючим MPI. Це пов'язано з тим, що неблокуючий підхід дозволяє процесорам виконувати інші корисні роботи, поки очікується на завершення операцій обміну даними.

На малих розмірах матриць та з невеликою кількістю процесорів різниця між блокуючим та неблокуючим MPI може бути незначною. Проте, зі збільшенням розміру матриць та кількості процесорів неблокуючий MPI демонструє перевагу, забезпечуючи кращу масштабованість та зменшення часу виконання.

Результати також показують, що ефективність MPI залежить від розмірності матриць та кількості доступних процесорів. На великих розмірах матриць та з більшою кількістю процесорів спостерігається покращення продуктивності, але з певного моменту, збільшення кількості процесорів може призвести до зростання накладних витрат та зниження швидкості обробки.

Загалом, неблокуючий підхід MPI є більш ефективним для множення матриць в розподілених обчисленнях порівняно з блокуючим підходом. Проте, перед використанням неблокуючого підходу, слід враховувати особливості конкретного завдання та характеристики системи, такі як розмір матриць, кількість процесорів та наявні ресурси, для забезпечення оптимальної продуктивності та використання ресурсів.

4. Код

```
import mpi.MPI;

import static java.lang.System.exit;

public class BlockingMPI {
    private static final int TAG_MASTER = 1;
    private static final int TAG_WORKER = 2;
    private static final int MASTER_ID = 0;
    public static void main(String[] args) {
        Matrix matrixA = new Matrix(1000, 1000, true);
        Matrix matrixB = new Matrix(1000, 1000, true);

        try {
            long startTime = System.currentTimeMillis();
            int rowsCount = matrixA.getRowsNumber();
            int columnsCount = matrixB.getColumnsNumber();
            Matrix resultMatrix = new Matrix(rowsCount, columnsCount,
false);

            MPI.Init(args);

            int tasksCount = MPI.COMM_WORLD.Size();
            int taskID = MPI.COMM_WORLD.Rank();

            int workersCount = tasksCount - 1;

            if (tasksCount < 2) {
                MPI.COMM_WORLD.Abort(1);
                exit(1);
            }

            if (taskID == MASTER_ID) {
                masterProcess(matrixA, matrixB, resultMatrix,
workersCount);
                System.out.println("Total time: " +
(System.currentTimeMillis() - startTime) + " ms");
                System.out.println("Matrix size: " + rowsCount + "x" +
columnsCount);
                System.out.println("Workers count: " + workersCount);
            } else {
                workerProcess(rowsCount, columnsCount);
            }
        } finally {
            MPI.Finalize();
        }
    }


    private static void workerProcess(int rowsCount, int columnsCount) {
        int[] startRowIndex = new int[1];
        int[] endRowIndex = new int[1];
        MPI.COMM_WORLD.Recv(startRowIndex, 0, 1, MPI.INT, 0, TAG_MASTER);
        MPI.COMM_WORLD.Recv(endRowIndex, 0, 1, MPI.INT, 0, TAG_MASTER);

        int sizeSubMatrix1Buffer = (endRowIndex[0] - startRowIndex[0] + 1)
* columnsCount * Integer.BYTES;
        int sizeMatrix2Buffer = rowsCount * columnsCount * Integer.BYTES;
```

```
        int[] subMatrix1Buffer = new int[sizeSubMatrix1Buffer];
        int[] matrix2Buffer = new int[sizeMatrix2Buffer];
        MPI.COMM_WORLD.Recv(subMatrix1Buffer, 0, sizeSubMatrix1Buffer,
MPI.INT, 0, TAG_MASTER);
        MPI.COMM_WORLD.Recv(matrix2Buffer, 0, sizeMatrix2Buffer, MPI.INT,
0, TAG_MASTER);

        Matrix subMatrix1 = Matrix.fromIntArray(subMatrix1Buffer,
                endRowIndex[0] - startRowIndex[0] + 1, columnsCount);
        Matrix matrix2 = Matrix.fromIntArray(matrix2Buffer, rowsCount,
columnsCount);
        Matrix resultMatrix = subMatrix1.multiplyMatrix(matrix2);

        int[] resultMatrixBuffer = resultMatrix.toIntArray();

        MPI.COMM_WORLD.Send(startRowIndex, 0, 1, MPI.INT, 0, TAG_WORKER);
        MPI.COMM_WORLD.Send(endRowIndex, 0, 1, MPI.INT, 0, TAG_WORKER);
        MPI.COMM_WORLD.Send(resultMatrixBuffer, 0,
resultMatrixBuffer.length, MPI.INT, 0, TAG_WORKER);
    }

    private static void masterProcess(Matrix matrix1, Matrix matrix2,
Matrix resultMatrix, int workers) {
        int rowsForOneWorker = resultMatrix.getRowsNumber() / workers;
        int extraRows = resultMatrix.getRowsNumber() % workers;

        for (int i = 1; i <= workers; i++) {
            int startRowIndex = (i - 1) * rowsForOneWorker;
            int endRowIndex = startRowIndex + rowsForOneWorker - 1;
            if (i == workers) {
                endRowIndex += extraRows;
            }

            Matrix subMatrix1 = matrix1.sliceMatrix(startRowIndex,
endRowIndex, resultMatrix.getColumnsNumber());
            int[] subMatrix1Buffer = subMatrix1.toIntArray();
            int[] matrix2Buffer = matrix2.toIntArray();

            MPI.COMM_WORLD.Send(new int[]{startRowIndex}, 0, 1, MPI.INT, i,
TAG_MASTER);
            MPI.COMM_WORLD.Send(new int[]{endRowIndex}, 0, 1, MPI.INT, i,
TAG_MASTER);
            MPI.COMM_WORLD.Send(subMatrix1Buffer, 0,
subMatrix1Buffer.length, MPI.INT, i, TAG_MASTER);
            MPI.COMM_WORLD.Send(matrix2Buffer, 0, matrix2Buffer.length,
MPI.INT, i, TAG_MASTER);
        }

        for (int i = 1; i <= workers; i++) {
            int[] startRowIndex = new int[1];
            int[] endRowIndex = new int[1];
            MPI.COMM_WORLD.Recv(startRowIndex, 0, 1, MPI.INT, i,
TAG_WORKER);
            MPI.COMM_WORLD.Recv(endRowIndex, 0, 1, MPI.INT, i, TAG_WORKER);

            int countElemsResultBuffer = (endRowIndex[0] - startRowIndex[0]
+ 1) * resultMatrix.getColumnsNumber() * Integer.BYTES;
            int[] resultMatrixBuffer = new int[countElemsResultBuffer];
            MPI.COMM_WORLD.Recv(resultMatrixBuffer, 0,
                    countElemsResultBuffer, MPI.INT, i, TAG_WORKER);
            Matrix subMatrix = Matrix.fromIntArray(resultMatrixBuffer,
```

```
                    endRowIndex[0] - startRowIndex[0] + 1,
resultMatrix.getColumnsNumber());

            resultMatrix.changeSlice(subMatrix, startRowIndex[0],
endRowIndex[0], resultMatrix.getColumnsNumber());
        }
    }
}
```

```
import mpi.MPI;
import mpi.Request;

import static java.lang.System.exit;

public class NonBlockingMPI {
    private static final int TAG_MASTER = 1;
    private static final int TAG_WORKER = 2;
    private static final int MASTER_ID = 0;

    public static void main(String[] args) {
        Matrix matrixA = new Matrix(1000, 1000, true);
        Matrix matrixB = new Matrix(1000, 1000, true);

        try{
            long startTime = System.currentTimeMillis();
            int rowsCount = matrixA.getRowsNumber();
            int columnsCount = matrixB.getColumnsNumber();
            Matrix resultMatrix = new Matrix(rowsCount, columnsCount,
false);

            MPI.Init(args);

            int countTasks = MPI.COMM_WORLD.Size();
            int taskID = MPI.COMM_WORLD.Rank();

            int workers = countTasks - 1;

            if(countTasks < 2){
                MPI.COMM_WORLD.Abort(1);
                exit(1);
            }

            if(taskID == MASTER_ID){
                masterProcess(matrixA, matrixB, resultMatrix, workers);

                System.out.println("Total time: " +
(System.currentTimeMillis() - startTime) + " ms");
                System.out.println("Matrix size: " + rowsCount + "x" +
columnsCount);
                System.out.println("Workers count: " + workers);
            }
            else {
                workerProcess(columnsCount, rowsCount);
            }
        }
        finally {
            MPI.Finalize();
        }
    }
```

```
    private static void workerProcess(int columnsCount, int rowsCount  ) {
        int[] startRowIndex = new int[1];
        int[] endRowIndex = new int[1];
        Request recStartIndex = MPI.COMM_WORLD.Irecv(startRowIndex,0,1,
MPI.INT, 0, TAG_MASTER);
        Request recEndIndex = MPI.COMM_WORLD.Irecv(endRowIndex,0,1,
MPI.INT, 0, TAG_MASTER);
        recStartIndex.Wait();
        recEndIndex.Wait();

        int sizeSubMatrix1Buffer = (endRowIndex[0] - startRowIndex[0] + 1)
* columnsCount;
        int sizeMatrix2Buffer = rowsCount * columnsCount;
        int[] subMatrix1Buffer = new int[sizeSubMatrix1Buffer];
        int[] matrix2Buffer = new int[sizeMatrix2Buffer];
        Request recSubMatrix1 = MPI.COMM_WORLD.Irecv(subMatrix1Buffer,0,
sizeSubMatrix1Buffer,
                MPI.INT,0, TAG_MASTER);
        Request recMatrix2 =
MPI.COMM_WORLD.Irecv(matrix2Buffer,0,sizeMatrix2Buffer, MPI.INT,0,
TAG_MASTER);
        recSubMatrix1.Wait();
        recMatrix2.Wait();

        Matrix subMatrix1 = Matrix.fromIntArray(subMatrix1Buffer,
                endRowIndex[0] - startRowIndex[0] + 1, columnsCount);
        Matrix matrix2 = Matrix.fromIntArray(matrix2Buffer, rowsCount,
columnsCount);
        Matrix resultMatrix = subMatrix1.multiplyMatrix(matrix2);

        int[] resultMatrixBuff = resultMatrix.toIntArray();

        MPI.COMM_WORLD.Isend(startRowIndex,0, 1, MPI.INT, 0, TAG_WORKER);
        MPI.COMM_WORLD.Isend(endRowIndex,0, 1, MPI.INT, 0, TAG_WORKER);
        MPI.COMM_WORLD.Isend(resultMatrixBuff,0, resultMatrixBuff.length,
MPI.INT, 0, TAG_WORKER);
    }

    private static void masterProcess(Matrix matrix1, Matrix matrix2,
Matrix resultMatrix, int workers) {
        int rowsForOneWorker = resultMatrix.getRowsNumber() / workers;
        int extraRows = resultMatrix.getRowsNumber() % workers;

        for (int i = 1; i <= workers; i++) {
            int startRowIndex = (i-1) * rowsForOneWorker;
            int endRowIndex = startRowIndex + rowsForOneWorker - 1;
            if(i == workers){
                endRowIndex += extraRows;
            }

            Matrix subMatrix1 = matrix1.sliceMatrix(startRowIndex,
endRowIndex, resultMatrix.getColumnsNumber());
            int[] subMatrix1Buff = subMatrix1.toIntArray();
            int[] matrix2Buff = matrix2.toIntArray();

            MPI.COMM_WORLD.Isend(new int[]{startRowIndex}, 0, 1, MPI.INT,
i, TAG_MASTER);
            MPI.COMM_WORLD.Isend(new int[]{endRowIndex}, 0, 1, MPI.INT, i,
TAG_MASTER);
            MPI.COMM_WORLD.Isend(subMatrix1Buff, 0, subMatrix1Buff.length ,
```

```
MPI.INT, i, TAG_MASTER);
            MPI.COMM_WORLD.Isend(matrix2Buff, 0, matrix2Buff.length,
MPI.INT, i, TAG_MASTER);
        }

        for (int i = 1; i <= workers; i++) {
            int[] startRowIndex = new int[1];
            int[] endRowIndex = new int[1];

            Request recStartIndex = MPI.COMM_WORLD.Irecv(startRowIndex, 0,
1, MPI.INT, i, TAG_WORKER);
            Request recEndIndex = MPI.COMM_WORLD.Irecv(endRowIndex, 0, 1,
MPI.INT, i, TAG_WORKER);
            recStartIndex.Wait();
            recEndIndex.Wait();

            int resultBufferElementsCount = (endRowIndex[0] -
startRowIndex[0] + 1) * resultMatrix.getColumnsNumber();
            int[] resultMatrixBuff = new int[resultBufferElementsCount];

            Request recRes = MPI.COMM_WORLD.Irecv(resultMatrixBuff, 0,
resultBufferElementsCount,
                    MPI.INT, i, TAG_WORKER);
            recRes.Wait();

            Matrix subMatrix = Matrix.fromIntArray(resultMatrixBuff,
                    endRowIndex[0] - startRowIndex[0] + 1,
resultMatrix.getColumnsNumber());
            resultMatrix.changeSlice(subMatrix, startRowIndex[0],
endRowIndex[0], resultMatrix.getColumnsNumber());
        }
    }

}
```

```
import java.util.Random;

public class Matrix {
    private final int[][] data;
    private final int rows;
    private final int columns;

    public Matrix(int[][] data) {
        this.data = data;
        this.rows = data.length;
        this.columns = data[0].length;
    }

    public Matrix(int rows, int columns, boolean generateRandom) {
        this.data = generateRandom ? generateMatrix(rows, columns) : new
int[rows][columns];
        this.rows = rows;
        this.columns = columns;
    }

    public static Matrix fromIntArray(int[] resultMatrixBuffer, int i, int
columnsNumber) {
        int[][] data = new int[i][columnsNumber];
        for (int row = 0; row < i; row++) {
            System.arraycopy(resultMatrixBuffer, row * columnsNumber,
```

```
data[row], 0, columnsNumber);
        }
        return new Matrix(data);
    }

    public int getValue(int i, int j) {
        return data[i][j];
    }

    public void setValue(int i, int j, int value) {
        data[i][j] = value;
    }

    public int getRowsNumber() {
        return rows;
    }

    public int getColumnsNumber() {
        return columns;
    }

    public int[] getRow(int row) {
        return data[row];
    }

    public int[][] getArrayCopy() {
        int[][] copy = new int[rows][columns];
        for (int row = 0; row < rows; row++) {
            System.arraycopy(data[row], 0, copy[row], 0, columns);
        }
        return copy;
    }

    public Matrix getMatrixCopy() {
        return new Matrix(getArrayCopy());
    }

    public void print() {
        for (int[] row : data) {
            for (int value : row) {
                System.out.print(value + " ");
            }
            System.out.println();
        }
        System.out.println();
    }

    public Matrix getTransposedMatrix() {
        int[][] transposed = new int[columns][rows];
        for (int row = 0; row < columns; row++) {
            for (int col = 0; col < rows; col++) {
                transposed[row][col] = data[col][row];
            }
        }
        return new Matrix(transposed);
    }

    public void addMatrix(Matrix matrix) {
        for (int row = 0; row < rows; row++) {
            for (int col = 0; col < columns; col++) {
                data[row][col] += matrix.getValue(row, col);
```

```
            }
        }
    }

    public Matrix multiplyMatrix(Matrix matrix) {
        int[][] result = new int[rows][matrix.getColumnsNumber()];
        for (int row = 0; row < rows; row++) {
            for (int col = 0; col < matrix.getColumnsNumber(); col++) {
                for (int i = 0; i < columns; i++) {
                    result[row][col] += data[row][i] * matrix.getValue(i,
col);
                }
            }
        }
        return new Matrix(result);
    }

    public Matrix sliceMatrix(int startRowIndex, int endRowIndex, int
columnsCount)
    {
        Matrix subMatrix = new Matrix(endRowIndex - startRowIndex + 1,
columnsCount, false);
        for (int i = startRowIndex; i <= endRowIndex; i++) {
            for (int j = 0; j < columnsCount; j++) {
                subMatrix.setValue(i - startRowIndex, j, data[i][j]);
            }
        }
        return subMatrix;
    }

    public void changeSlice(Matrix matrix, int indexStartRow, int
indexEndRow, int countColumns)
    {
        for (int i = indexStartRow; i <= indexEndRow; i++) {
            for (int j = 0; j < countColumns; j++) {
                data[i][j] = matrix.getValue(i - indexStartRow, j);
            }
        }
    }

    public static boolean compareMatrices(Matrix m1, Matrix m2) {
        if (m1.getRowsNumber() != m2.getRowsNumber() ||
m1.getColumnsNumber() != m2.getColumnsNumber()) {
            return false;
        }
        for (int row = 0; row < m1.getRowsNumber(); row++) {
            for (int col = 0; col < m1.getColumnsNumber(); col++) {
                if (m1.getValue(row, col) != m2.getValue(row, col)) {
                    return false;
                }
            }
        }
        return true;
    }

    private int[][] generateMatrix(int rows, int columns) {
        int[][] matrix = new int[rows][columns];
        Random random = new Random();

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
```

```
                matrix[i][j] = random.nextInt(10);
            }
        }
        return matrix;
    }

    public int[] toIntArray() {
        int[] array = new int[rows * columns];
        int index = 0;
        for (int[] row : data) {
            for (int value : row) {
                array[index++] = value;
            }
        }
        return array;
    }
}
```