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 № 7 з дисципліни
«Технології паралельних та розподілених обчислень»

«Розробка паралельного алгоритму множення матриць з використанням MPI-методів колективного обміну повідомленнями («один-до-багатьох», «багато-до-одного», «багато-до-багатьох») та дослідження його ефективності»»

Виконав(ла) *ІП-01 Корнієнко В.С.* _____
(шифр, прізвище, ім'я, по батькові)

Перевірив(ла) *Стеценко І. В.* _____
(прізвище, ім'я, по батькові)

Київ 2023

Завдання:

1. Ознайомитись з методами колективного обміну повідомленнями типу «один-до-багатьох», «багато-до-одного», «багато-до-багатьох» (див. лекцію та документацію стандарту MPI).
2. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів колективного обміну повідомленнями. **40 балів.**
3. 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 методів обміну повідомленнями «один-до-одного», «один-до-багатьох», «багато-до-одного», «багато-до-багатьох». **60 балів.**

1. **Ознайомитись з методами колективного обміну повідомленнями типу «один-до-багатьох», «багато-до-одного», «багато-до-багатьох» MPI**

- **Один-до-багатьох (One-to-Many):**
  Цей метод дозволяє одному процесу відправити повідомлення до кількох інших процесів. Це може бути корисно, наприклад, коли один процес виконує обчислення та надсилає результати іншим процесам.

- **Багато-до-одного (Many-to-One):**
  Цей метод дозволяє кільком процесам надіслати повідомлення одному конкретному процесу. Це може бути корисно, коли кілька процесів збирають результати своїх обчислень в одному процесі для подальшої обробки.

- **Багато-до-багатьох (Many-to-Many):**
  Цей метод дозволяє обмінюватись повідомленнями між кожною парою процесів. Кожен процес може надіслати повідомлення іншому процесу, і отримати повідомлення від нього.

2. Реалізувати алгоритм паралельного множення матриць з використанням розподілених обчислень в MPI з використанням методів колективного обміну повідомленнями.

Для даного завдання був створений клас CollectiveMPI, який реалізує множення матриць з використанням методів колективного обміну повідомленнями.

Даний клас містить конструктор, який має в собі вхідні аргументи програми

```
public class CollectiveMPI implements IMatrixMultiplicationAlgorithm {
    1 usage
    private static final int MASTER_ID = 0;
    1 usage
    private static final int INT_32_BYTE_SIZE = 4;
    2 usages
    private final String[] args;

    1 usage  valerii.korniienko
    public CollectiveMPI(String[] args) {
        this.args = args;
    }
```

У методі **multiply**() спочатку ініціалізується MPI за допомогою MPI.Init(args), далі отримується кількість процесів tasksCount та ідентифікатор поточного процесу taskID.

```
@Override
public Result multiply(Matrix matrixA, Matrix matrixB) {
    try{
        long startTime = System.currentTimeMillis();

        MPI.Init(args);

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

Далі метод calculateBytes() обчислює розміри підматриць для кожного процесу, які будуть розподілені між ними, а метод calculateOffsets() обчислює зсуви, які вказують на початкові позиції для розподілу даних.

```
int[] bytes = calculateBytes(matrixA, matrixB, tasksCount);
int[] offsets = calculateOffsets(tasksCount, bytes);
```

Потім Матриці A та B перетворюються у байтові масиви, кожен процес отримує свою підматрицю A за допомогою операції Scatterv(), яка розподіляє дані між процесами згідно з розмірами та зсувами.

```
int taskBytes = bytes[taskID];
byte[] subMatrixBytes = new byte[taskBytes];
byte[] resBytes = new byte[matrixA.getRowsCount() * matrixB.getColumnsCount() * INT_32_BYTE_SIZE];

MPI.COMM_WORLD.Scatterv(matrixAByteBuffer, sendoffset: 0, bytes, offsets, MPI.BYTE,
        subMatrixBytes, recvoffset: 0, taskBytes, MPI.BYTE, root: 0);

MPI.COMM_WORLD.Bcast(matrixBByteBuffer, offset: 0, matrixBByteBuffer.length, MPI.BYTE, root: 0);
```

Потім, матриця B розсилається всім процесам за допомогою операції Bcast().

```
MPI.COMM_WORLD.Bcast(matrixBByteBuffer, offset: 0, matrixBByteBuffer.length, MPI.BYTE, root: 0);
```

Кожен процес виконує множення своєї підматриці на матрицю B, результати множення збираються в буфер resBytes за допомогою операції Gatherv(), яка збирає дані з різних процесів згідно з розмірами та зсувами.

```
byte[] multiplicationResultBuffer = performMatrixMultiplication(matrixA.getRowsCount(),
        matrixB.getColumnsCount(), matrixBByteBuffer, taskBytes, subMatrixBytes)
        .toByteBuffer();

MPI.COMM_WORLD.Gatherv(multiplicationResultBuffer, sendoffset: 0, multiplicationResultBuffer.length,
        MPI.BYTE, resBytes, recvoffset: 0, bytes, offsets, MPI.BYTE, root: 0);
```

Якщо поточний процес - головний (з ID 0), то результат з буфера resBytes використовується для створення об'єкта Result, який містить матрицю результату та час виконання.

```
if (taskID == MASTER_ID) {
    Matrix resultMatrix = MatrixHelper.createMatrixFromBuffer(resBytes, matrixA.getRowsCount(),
            matrixB.getColumnsCount());

    return new Result(resultMatrix, totalTime: System.currentTimeMillis() - startTime);
}
```

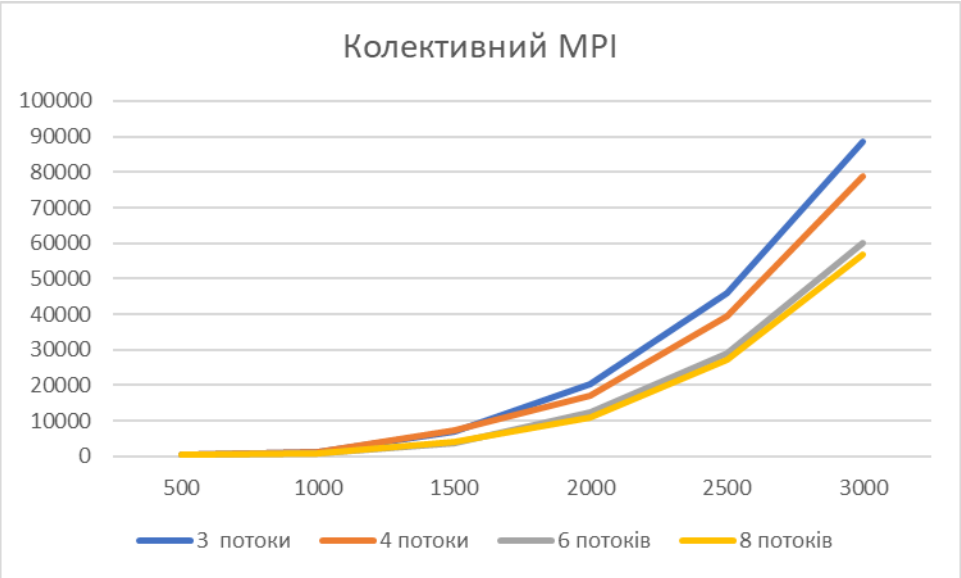
Нарешті, MPI завершує роботу за допомогою MPI.Finalize().

```
        }finally {
            MPI.Finalize();
        }
    }
```

3. **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 методів обміну повідомленнями «один-до-одного», «один-до-багатьох», «багато-до-одного», «багато-до-багатьох».**

Маємо такі результати:

Розмірність матриці	3 потоки	4 потоки	6 потоків	8 потоків
500	305	344	275	263
1000	1241	1209	702	649
1500	7017	7198	3557	3964
2000	20357	17059	12168	10833
2500	45897	39627	29038	27180
3000	88724	78860	59924	56664



Аналізуючи результати множення матриць за допомогою колективного MPI та порівнюючи його з блокуючим і неблокуючим MPI, можна зробити наступні спостереження:

Загальною тенденцією є те, що колективний MPI показує значно більші значення часу виконання порівняно з блокуючим та неблокуючим MPI. Це свідчить про те, що колективний підхід може бути менш ефективним для множення матриць у порівнянні з іншими підходами.

Колективний MPI показує кращі результати порівняно з блокуючим MPI при малих розмірах матриць (наприклад, розмірність 500) та невеликій

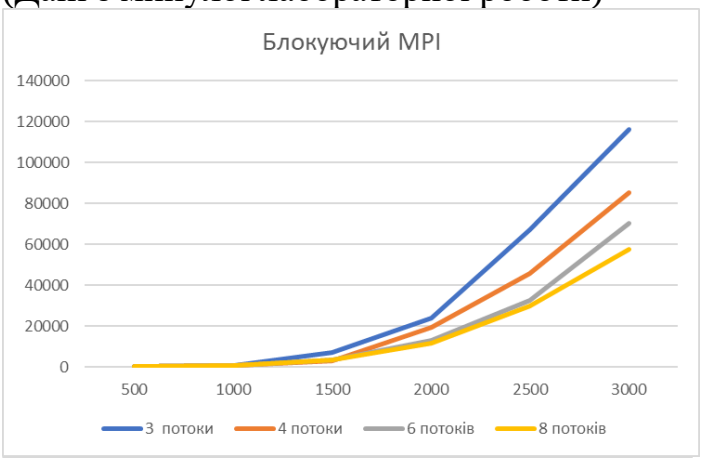
кількості потоків. Однак, при збільшенні розмірності матриць і кількості потоків блокуючий MPI може стати швидшим.

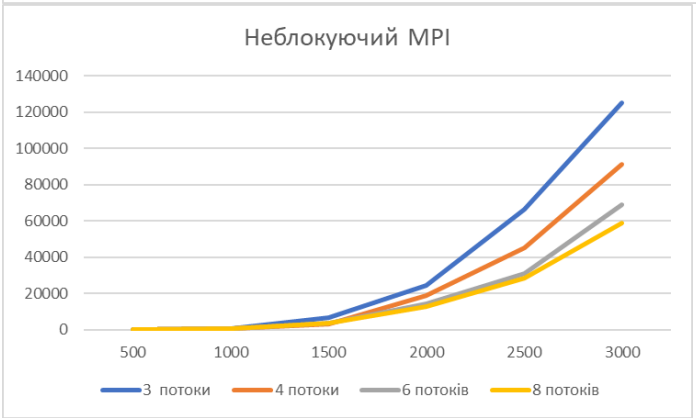
У порівнянні з неблокуючим MPI, колективний MPI зазвичай показує гірші або такі самі результати в усіх комбінаціях розмірності матриць та кількості потоків. Це може бути пов'язано з більшою комунікаційною накладною колективного підходу, яка впливає на швидкодію виконання.

Колективний MPI може мати деяку перевагу в тих випадках, коли розмірність матриць менша, а кількість потоків досить велика. Однак, при збільшенні розмірності матриць і кількості потоків його ефективність зменшується.

На підставі цих спостережень можна зробити висновок, що колективний MPI може бути менш ефективним для множення матриць у порівнянні з блокуючим і неблокуючим MPI, особливо при великих розмірностях матриць та більшій кількості потоків. Результати можуть залежати від конкретного завдання та параметрів обчислювальної системи.

(Дані з минулої лабораторної роботи)





Лістинг коду:

CollectiveMPI.java

```
import mpi.MPI;

public class CollectiveMPI implements
IMatrixMultiplicationAlgorithm {
    private static final int MASTER_ID = 0;
    private static final int INT_32_BYTE_SIZE = 4;
    private final String[] args;

    public CollectiveMPI(String[] args) {
        this.args = args;
    }

    @Override
    public Result multiply(Matrix matrixA, Matrix matrixB) {
        try{
            long startTime = System.currentTimeMillis();

            MPI.Init(args);

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

            int[] bytes = calculateBytes(matrixA, matrixB,
tasksCount);
            int[] offsets = calculateOffsets(tasksCount,
bytes);

            byte[] matrixAByteBuffer = matrixA.toByteBuffer();
            byte[] matrixBByteBuffer = matrixB.toByteBuffer();

            int taskBytes = bytes[taskID];
            byte[] subMatrixBytes = new byte[taskBytes];
            byte[] resBytes = new byte[matrixA.getRowsCount()
* matrixB.getColumnsCount() * INT_32_BYTE_SIZE];

            MPI.COMM_WORLD.Scatterv(matrixAByteBuffer, 0,
bytes, offsets, MPI.BYTE,
                    subMatrixBytes, 0, taskBytes, MPI.BYTE,
0);

            MPI.COMM_WORLD.Bcast(matrixBByteBuffer, 0,
matrixBByteBuffer.length, MPI.BYTE, 0);

            byte[] multiplicationResultBuffer =
performMatrixMultiplication(matrixA.getRowsCount(),
                    matrixB.getColumnsCount(),
matrixBByteBuffer, taskBytes, subMatrixBytes)
```

```
                    .toByteBuffer();

            MPI.COMM_WORLD.Gatherv(multiplicationResultBuffer,
0, multiplicationResultBuffer.length,
                    MPI.BYTE, resBytes, 0, bytes, offsets,
MPI.BYTE, 0);

            if (taskID == MASTER_ID) {
                Matrix resultMatrix =
MatrixHelper.createMatrixFromBuffer(resBytes,
matrixA.getRowsCount(),
                        matrixB.getColumnsCount());

                return new Result(resultMatrix,
System.currentTimeMillis() - startTime);
            }

            return null;
        }finally {
            MPI.Finalize();
        }
    }

    private Matrix performMatrixMultiplication(int
matrix1RowsCount, int matrix2ColumnsCount, byte[]
secondMatrixBuffer, int taskBytes, byte[] subMatrixBytes) {
        Matrix subMatrix =
MatrixHelper.createMatrixFromBuffer(subMatrixBytes, taskBytes
/ (INT_32_BYTE_SIZE * matrix2ColumnsCount), matrix1RowsCount);
        Matrix secondMatrix =
MatrixHelper.createMatrixFromBuffer(secondMatrixBuffer,
matrix2ColumnsCount, matrix1RowsCount);

        return subMatrix.multiply(secondMatrix);
    }

    private int[] calculateBytes(Matrix matrix1, Matrix
matrix2, int tasksCount) {
        var rowsForOneWorker = matrix1.getRowsCount() /
tasksCount;
        var extraRows = matrix1.getRowsCount() % tasksCount;

        int[] bytes = new int[tasksCount];
        for (var i = 0; i < tasksCount; i++) {
            if (i != tasksCount - 1) {
                bytes[i] = rowsForOneWorker *
matrix2.getColumnsCount() * INT_32_BYTE_SIZE;
            } else {
                bytes[i] = (rowsForOneWorker + extraRows) *
matrix2.getColumnsCount() * INT_32_BYTE_SIZE;
            }
```

```
        }
        return bytes;
    }

    private int[] calculateOffsets(int tasksCount, int[]
bytes) {
        int[] offsets = new int[tasksCount];
        for (var i = 0; i < offsets.length; i++) {
            if (i == 0) continue;

            offsets[i] = bytes[i - 1] + offsets[i - 1];
        }
        return offsets;
    }
}
```

CollectiveMPIMain.java

```
import mpi.MPI;

public class CollectiveMPIMain {

    public static void main(String[] args){
        int size = 3000;
        Matrix matrix1 =
MatrixHelper.generateRandomMatrix(size);
        Matrix matrix2 =
MatrixHelper.generateRandomMatrix(size);

        CollectiveMPI collectiveMPI = new CollectiveMPI(args);
        Result collectiveMPIResult =
collectiveMPI.multiply(matrix1, matrix2);
        if (collectiveMPIResult == null) {
            return;
        }
        System.out.println("Collective MPI: ");
        System.out.println("Matrix size: " + size);
        System.out.println("Processors count: " +
MPI.COMM_WORLD.Size());
//
System.out.println(collectiveMPIResult.getResultMatrix().equal
s(matrix1.multiply(matrix2)) ?
//                    "Result is Correct" : "Result is
Incorrect");
        System.out.println("Total time: " +
collectiveMPIResult.getTotalTime());
    }
}
```

Shared resources

Matrix.java

```
import java.nio.ByteBuffer;
import java.nio.ByteOrder;

public class Matrix {
    private int[][] matrixData;
    public static final int INT_32_BYTE_SIZE = 4;

    public Matrix(int[][] matrix) {
        this.matrixData = matrix;
    }

    public Matrix(int height, int width){
        this.matrixData = new int[height][width];
    }
    public int[] getRow(int rowIndex){
        return matrixData[rowIndex];
    }
    public int[][] getMatrix() {
        return matrixData;
    }

    public int getRowsCount() {
        return matrixData.length;
    }

    public int getColumnsCount() {
        return matrixData[0].length;
    }

    public int get(int row, int column) {
        return matrixData[row][column];
    }

    public void set(int row, int column, int value) {
        matrixData[row][column] = value;
    }

    public void add(Matrix matrixB) {
        for (int i = 0; i < matrixB.getRowsCount(); i++) {
            for (int j = 0; j < matrixB.getColumnsCount();
j++) {
                matrixData[i][j] += matrixB.get(i, j);
            }
        }
    }

    public boolean equals(Matrix matrix1){
        if (matrixData.length != matrix1.getRowsCount() ||
```

```
matrixData[0].length != matrix1.getColumnsCount()) {
            return false;
        }
        for (int i = 0; i < matrixData.length; i++) {
            for (int j = 0; j < matrix1.getColumnsCount();
j++) {
                if (matrixData[i][j] != matrix1.get(i, j)) {
                    return false;
                }
            }
        }
        return true;
    }

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

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

    public int[] toIntBuffer()
    {
        int [] array = new int[getRowsCount() *
getColumnsCount()];
        int index = 0;
        for (int i = 0; i < getRowsCount(); i++) {
            for (int j = 0; j < getColumnsCount(); j++) {
                array[index] = matrixData[i][j];
                index++;
            }
        }
        return array;
```

```
    }

    public byte[] toByteBuffer() {
        var buffer = ByteBuffer.allocate(getRowsCount() *
getColumnsCount() * INT_32_BYTE_SIZE);
        buffer.order(ByteOrder.nativeOrder());
        var intBuffer = buffer.asIntBuffer();
        for (var ints : matrixData) {
            intBuffer.put(ints);
        }

        return buffer.array();
    }

    public Matrix transpose() {
        int[][] result = new
int[matrixData[0].length][matrixData.length];
        for (int i = 0; i < matrixData.length; i++) {
            for (int j = 0; j < matrixData[0].length; j++) {
                result[j][i] = matrixData[i][j];
            }
        }
        return new Matrix(result);
    }

    public Matrix clone() {
        int[][] result = new
int[matrixData.length][matrixData[0].length];
        for (int i = 0; i < matrixData.length; i++) {
            System.arraycopy(matrixData[i], 0, result[i], 0,
matrixData[0].length);
        }
        return new Matrix(result);
    }

    public void print() {
        for (int i = 0; i < matrixData.length; i++) {
            for (int j = 0; j < matrixData[0].length; j++) {
                System.out.print(matrixData[i][j] + " ");
            }
            System.out.println();
        }
    }

    public Matrix multiply(Matrix matrix2) {
        int[][] result = new
int[matrixData.length][matrix2.getColumnsCount()];
        for (int i = 0; i < matrixData.length; i++) {
            for (int j = 0; j < matrix2.getColumnsCount();
j++) {
                for (int k = 0; k < matrixData[0].length; k++)
```

```
{
                    result[i][j] += matrixData[i][k] *
matrix2.get(k, j);
                }
            }
        }
        return new Matrix(result);
    }
}
```

MatrixHelper.java

```
import java.nio.ByteBuffer;
import java.nio.ByteOrder;

public class MatrixHelper {
    public static Matrix generateRandomMatrix(int width, int
height, int minValue, int maxValue) {
        int[][] result = new int[height][width];
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                result[i][j] = (int) (Math.random() *
(maxValue - minValue)) + minValue;
            }
        }
        return new Matrix(result);
    }

    public static Matrix generateRandomMatrix(int size) {
        return generateRandomMatrix(size, size, 0, 100);
    }

    public static Matrix createMatrixFromBuffer(int[] array,
int rowsCount, int columnsCount) {
        int[][] matrixData = new int[rowsCount][columnsCount];

        int arrayIndex = 0;
        for (int i = 0; i < rowsCount; i++) {
            for (int j = 0; j < columnsCount; j++) {
                matrixData[i][j] = array[arrayIndex];
                arrayIndex++;
            }
        }
        return new Matrix(matrixData);
    }

    public static Matrix createMatrixFromBuffer(byte[] bytes,
int rows, int cols) {
        var buffer = ByteBuffer.wrap(bytes);
        buffer.order(ByteOrder.nativeOrder());
        var array = new int[rows][cols];
```

```
        for (var i = 0; i < rows; i++) {
            for (var j = 0; j < cols; j++) {
                array[i][j] = buffer.getInt();
            }
        }
        return new Matrix(array);
    }
}
```