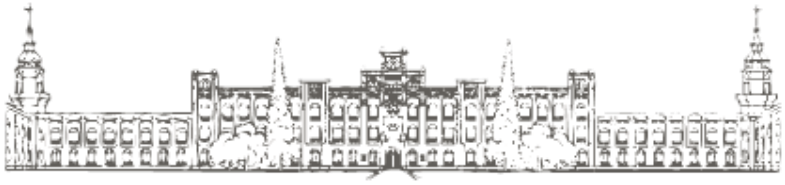

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»
КАФЕДРА ІНФОРМАТИКИ ТА ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Звіт

до лабораторного практикуму 2

«Розробка паралельних алгоритмів множення матриць та дослідження їх ефективності»

з дисципліни

«Технології паралельних обчислень. Курсова робота»

Виконав:
студент групи ІП-01
Пашковський Євгеній

Київ – 2023

Завдання

1. Реалізуйте стрічковий алгоритм множення матриць. Результат множення записуйте в об’єкт класу Result.
2. Реалізуйте алгоритм Фокса множення матриць.
3. Виконайте експерименти, варіюючи розмірність матриць, які перемножуються, для обох алгоритмів, та реєструючи час виконання алгоритму. Порівняйте результати дослідження ефективності обох алгоритмів.
4. Виконайте експерименти, варіюючи кількість потоків, що використовується для паралельного множення матриць, та реєструючи час виконання. Порівняйте результати дослідження ефективності обох алгоритмів.

Хід виконання

Стрічковий алгоритм множення матриць:

```
public Result multiplyStripe(IntegerMatrix integerMatrix) {
    final long startTime = System.currentTimeMillis();

    if (!this.validateMatrixForProduct(integerMatrix)) throw new
RuntimeException("Matrix has bad size");

    int n = this.getHeight();
    int m = this.getWidth();

    int threadsCount = Runtime.getRuntime().availableProcessors();

    ExecutorService threadPool = Executors.newFixedThreadPool(threadsCount);

    ColumnStripeClock columnStripeClock = new ColumnStripeClock(integerMatrix);

    Future<int[]>[] futures = new Future[n];
    for (int j = 0; j < n; j++) {
        Future<int[]> rowFuture = threadPool.submit(new
RowStripeCallable(this.array[j], columnStripeClock));
        futures[j] = rowFuture;
    }

    int[][] newMatrixArray = new int[n][m];

    for (int i = 0; i < n; i++) {
        try {
            newMatrixArray[i] = futures[i].get();
        } catch (InterruptedException | ExecutionException e) {
            throw new RuntimeException(e);
        }
    }

    threadPool.close();

    IntegerMatrix resultIntegerMatrix = new IntegerMatrix(newMatrixArray);

    final long endTime = System.currentTimeMillis();

    return new Result(resultIntegerMatrix, endTime - startTime);
}

import java.util.concurrent.Callable;

public class RowStripeCallable implements Callable<int[]> {

    private final int[] row;

    private final ColumnStripeClock columnStripeClock;

    public RowStripeCallable(int[] row, ColumnStripeClock columnStripeClock) {
        this.row = row;
        this.columnStripeClock = columnStripeClock;
    }

    @Override
    public int[] call() throws Exception {
        int id = columnStripeClock.register();
        int[] resultRow = new int[row.length];

        int i = -1;
        for (int m = 0; m < row.length; m++) {
            i = columnStripeClock.getIndex(id);
            int[] column = columnStripeClock.getColumn(id);

            for (int k = 0; k < row.length; k++) {
                resultRow[i] += row[k] * column[k];
```

```
            }

            columnStripeClock.shift(id);
        }

        return resultRow;
    }
}


import java.util.HashMap;
import java.util.Map;

public class ColumnStripeClock {
    private final int[][] columns;
    private final Map<Integer, Integer> idToIndexMap = new HashMap<>();

    public ColumnStripeClock(IntegerMatrix integerMatrix) {
        this.columns = integerMatrix.getTransponedMatrix().getArray();
    }

    public synchronized int register() {
        int id = idToIndexMap.size();
        idToIndexMap.put(id, id);
        return id;
    }

    public synchronized int getIndex(int id) throws Exception {
        int i = idToIndexMap.get(id);

        if (i < 0) throw new Exception("Unregistered value");
        return i;
    }

    public synchronized int[] getColumn(int id) throws Exception {
        return columns[this.getIndex(id)];
    }

    public synchronized void shift(int id) {
        int index = idToIndexMap.get(id);
        idToIndexMap.put(id, index == columns.length - 1 ? 0 : index + 1);
    }
}
```

Алгоритм фокса:

```
public Result multiplyFox(IntegerMatrix integerMatrix, int blockSize) {
    final long startTime = System.currentTimeMillis();

    if (!this.validateMatrixForProduct(integerMatrix)) throw new
RuntimeException("Matrix has bad size");

    BlockMatrix blockMatrixA = this.toBlockMatrix(blockSize);
    BlockMatrix blockMatrixB = integerMatrix.toBlockMatrix(blockSize);

    int n = blockMatrixA.getHeight();
    int m = blockMatrixA.getWidth();

    IntegerMatrix[][] blockMatrixARows = blockMatrixA.getArray();
    IntegerMatrix[][] blockMatrixBColumns =
blockMatrixB.getTransponedMatrix().getArray();

    int threadsCount = Runtime.getRuntime().availableProcessors();
    ExecutorService threadPool = Executors.newFixedThreadPool(threadsCount);

    Future<IntegerMatrix>[][] futureMatrixCArray = new Future[n][m];

    for (int j = 0; j < n; j++) {
        for (int i = 0; i < m; i++) {
            futureMatrixCArray[j][i] = threadPool.submit(new
```

```
FoxCallable(blockMatrixARows[j], blockMatrixBColumns[i]));
        }
    }

    IntegerMatrix[][] blockMatrixCArray = new IntegerMatrix[n][m];

    for (int j = 0; j < n; j++) {
        for (int i = 0; i < m; i++) {
            try {
                blockMatrixCArray[j][i] = futureMatrixCArray[j][i].get();
            } catch (InterruptedException | ExecutionException e) {
                throw new RuntimeException(e);
            }
        }
    }

    threadPool.close();

    IntegerMatrix resultIntegerMatrix = IntegerMatrix.fromBlockMatrix(new
BlockMatrix(blockMatrixCArray));

    final long endTime = System.currentTimeMillis();

    return new Result(resultIntegerMatrix, endTime - startTime);
}

import java.util.concurrent.Callable;

public class FoxCallable implements Callable<IntegerMatrix> {
    private final IntegerMatrix[] row;
    private final IntegerMatrix[] column;

    public FoxCallable(IntegerMatrix[] row, IntegerMatrix[] column) {
        this.row = row;
        this.column = column;
    }

    @Override
    public IntegerMatrix call() {
        int blockSize = row[0].getWidth();

        IntegerMatrix ij = IntegerMatrix.getZeroMatrix(blockSize, blockSize);

        for (int k = 0; k < row.length; k++) {
            ij = ij.add(row[k].multiplySingleThread(column[k]).getMatrix());
        }

        return ij;
    }
}
```

Порівняння роботи алгоритмів

Тестувалося на пристрої з такими характеристиками:

- Процесор Intel Core i7-8550U. Базова частота 1.8-2.0ГГц, але фактично може досягати 4ГГц у пікових навантаженнях. Містить 4 фізичних ядра та 8 логічних процесорів.
- 16Гб оперативної пам’яті з частотою 2400МГц
- ОС Windows 10.

Стрічковий алгоритм:

Matrix Size	Serial algorithm	2 processors		4 processors		8 processors	
		Time	Speed Up	Time	Speed Up	Time	Speed Up
500	0,1854	0,0738	2,5122	0,0686	2,7026	0,071	2,6113
1000	1,7052	0,3694	4,6161	0,2902	5,8759	0,4786	3,5629
1500	30,7836	1,4416	21,3537	0,8734	35,2457	1,7164	17,935
2000	98,9058	3,031	32,6314	2,1046	46,9951	3,6416	27,16

Алгоритм Фокса для розміру блоку 10:

Matrix Size	Serial algorithm	2 processors		4 processors		8 processors	
		Time	Speed Up	Time	Speed Up	Time	Speed Up
500	0,1854	0,194	0,9557	0,1302	1,424	0,1266	1,4646
1000	1,7052	1,2644	1,3486	1,0348	1,6479	1,0898	1,5647
1500	30,7836	5,0232	6,1282	3,9316	7,8298	3,444	8,9383
2000	98,9058	11,4662	8,6259	8,9682	11,0285	7,457	13,2635

Висновки

Виконуючи роботу комп’ютерного практикуму, було реалізовано та досліджено ефективність таких паралельних алгоритмів множення матриць як стрічковий алгоритм та алгоритм Фокса. Навчився виконувати експерименти над паралельними алгоритмами та порівнювати їх між собою та з аналогічним послідовним алгоритмом.