**Міністерство освіти і науки України**

**Національний технічний університет України**

**«Київський політехнічний інститут імені Ігоря Сікорського»**

**Факультет інформатики та обчислювальної техніки**

**Кафедра обчислювальної техніки**

**Лабораторна робота №2**

з дисципліни «Програмне забезпечення високопродуктивних комп’ютерних систем»

з теми «Програмування потоків»

**Виконав:** **Перевірив:**

студент групи ІМ-21 Корочкін О. В.

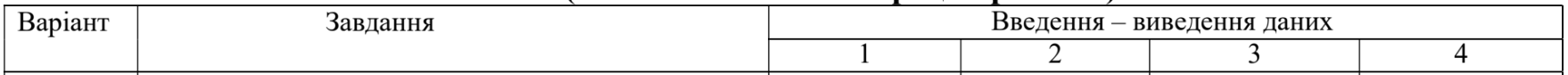
Галай Андрій Вікторович

**Київ 2025**

**Завдання**

Розробити паралельний алгоритм рішення математичної задачі, виявити спільні ресурси для потоків. Описати алгоритм кожного потоку (Т1 – Т4 ) з визначенням критичних ділянок (КД) і точок синхронізації (Wij , Sij). Розробити структурну схему взаємодії задач, де застосувати всі вказані засоби взаємодії процесів. Розробити програму (обов’язкові “шапка” (header) , коментарі), виконати налагодження програми. Отримати правильні результати обчислень. Провести тестування програми для різних значень N (1000 і більше) і кількості процесорів (ядер) 1 та 4 з метою отримання часу виконання програми Час1 та Час4 визначення коефіцієнту прискорення Кп = Час1/Чаc4.

**Завдання за варіантом (Варіант 5)**

****

****

Формула для обчислення:  
 МА= MD\*MC \*d + min(Z)\*MX\*p

Вхідні дані:

- d, p - скаляри;

- MD, MC, MX - матриці;

- Z - вектор;

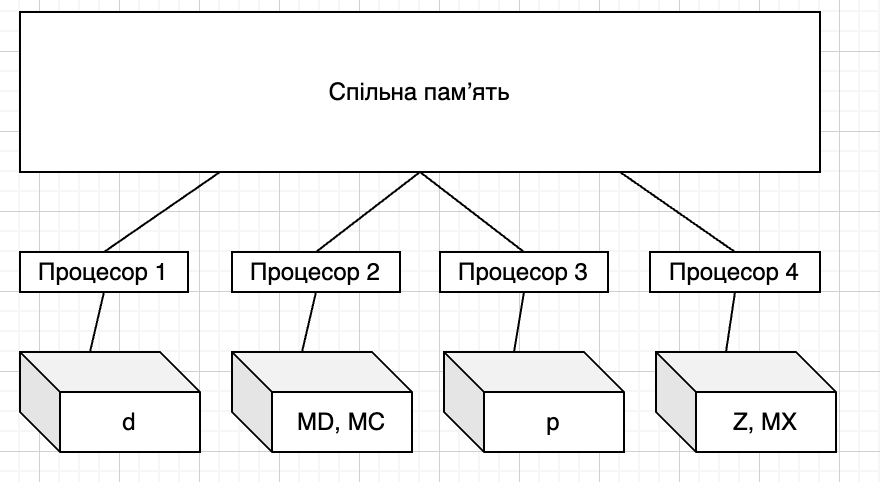
Вихідні дані:

- MA - матриця результату.

**Мова програмування:** Java

**Засоби взаємодії потоків:** семафори, критичні секції, атомарні змінні, бар’єри.

**Структура ПКС:** Система зі спільною пам’яттю включає чотири процесори та чотири пристрої введення-виведення:



**Виконання лабораторної роботи**

**Етап 1. Розробка алгоритму паралельного обчислення.**

1. zi = min(Z), i = 1..P
2. z = min(z, zi) - КД1
3. MAн = (MD \* MCн) \* d + z \* MXн \* p - (КД2 для копіювання d, КД3 - для z, КД4 - для p)

Спільні ресурси:

z - спільний ресурс, доступ до якого потрібно контролювати

z, d, p - спільний ресурс, копіювати у критичних ділянках

MD - спільний ресурс, який не змінюється під час доступу потоків до нього, тому доступ до нього можна не контролювати

**Етап 2. Розробка алгоритму потоків із з визначенням критичних ділянок (КД) і точок синхронізації (Wij , Sij).**

*Потік Т1*

1. Уведення d
2. Сигнал задачам T2, T3, T4 про введення d - S2,1; S3,1; S4,1
3. Чекати на введення даних у задачах T2, T3, T4 - W2,1; W3,1; W4,1
4. Обчислення z1 = min(Zн)
5. Обчислення z = min(z, z1) - КД
6. Сигнал задачі T2, T3, T4 про завершення обчислення z - S2,2; S3,2; S4,2
7. Чекати завершення обчислення z в задачі T2, T3, T4 - W2,2; W3,2; W4,2
8. Копіювати d1 = d - КД
9. Копіювати z1 = z - КД
10. Копіювати p1 = p - КД
11. Обчислення MAн = (MD \* MCн) \* d1 + z1 \* MXн \* p1
12. Чекати завершення обчислення MAн в T2, T3, T4 - W2,3; W3,3; W4,3
13. Виведення МА

*Потік Т2*

1. Уведення MD, MC
2. Сигнал задачам T1, T3, T4 про введення MD, MC - S1,1; S3,1; S4,1
3. Чекати на введення даних у задачах T1, T3, T4 - W1,1; W3,1; W4,1
4. Обчислення z2 = min(Zн)
5. Обчислення z = min(z, z2) - КД
6. Сигнал задачі T1, T3, T4 про завершення обчислення z - S1,2; S3,2; S4,2
7. Чекати завершення обчислення z в задачі T1, T3, T4 - W1,2; W3,2; W4,2
8. Копіювати d2 = d - КД
9. Копіювати z2 = z - КД
10. Копіювати p2 = p - КД
11. Обчислення MAн = (MD \* MCн) \* d2 + z2 \* MXн \* p2
12. Сигнал T1 про завершення обчислення MAн - S1,3

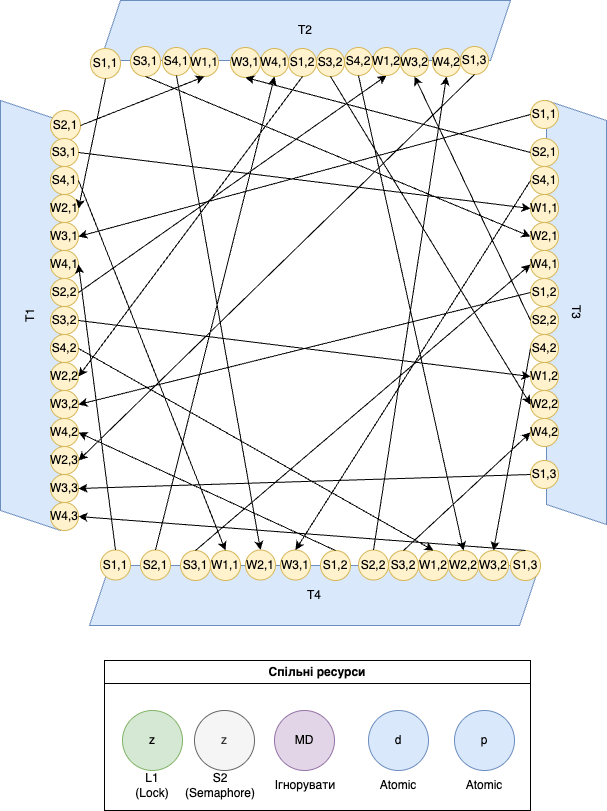
*Потік Т3*

1. Уведення p
2. Сигнал задачам T1, T2, T4 про введення p - S1,1; S2,1; S4,1
3. Чекати на введення даних у задачах T1, T2, T4 - W1,1; W2,1; W4,1
4. Обчислення z3 = min(Zн)
5. Обчислення z = min(z, z3) - КД
6. Сигнал задачі T1, T2, T4 про завершення обчислення z - S1,2; S2,2; S4,2
7. Чекати завершення обчислення z в задачі T1, T2, T4 - W1,2; W2,2; W4,2
8. Копіювати d3 = d - КД
9. Копіювати z3 = z - КД
10. Копіювати p3 = p - КД
11. Обчислення MAн = (MD \* MCн) \* d3+ z3 \* MXн \* p3
12. Сигнал T1 про завершення обчислення MAн - S1,3

*Потік Т4*

1. Уведення Z, MX
2. Сигнал задачам T1, T2, T3 про введення Z, MX - S1,1; S2,1; S3,1
3. Чекати на введення даних у задачах T1, T2, T3 - W1,1; W2,1; W3,1
4. Обчислення z4 = min(Zн)
5. Обчислення z = min(z, z4) - КД
6. Сигнал задачі T1, T2, T3 про завершення обчислення z - S1,2; S2,2; S3,2
7. Чекати завершення обчислення z в задачі T1, T2, T3 - W1,2; W2,2; W3,2
8. Копіювати d4 = d - КД
9. Копіювати z4 = z - КД
10. Копіювати p4 = p - КД
11. Обчислення MAн = (MD \* MCн) \* d4 + z4 \* MXн \* p4
12. Сигнал T1 про завершення обчислення MAн - S1,3

**Етап 3. Розробка структурної схеми взаємодії потоків**



Використані позначення:

* S**j,k -** точка синхронізації, що повʼязана із потоком **j, k** - порядковий номер взаємодії цього потоку та потоку Тj. S**j,k** повʼязана із сигналом до потоку Тj про подію, яка відбулася.
* W**j,k** - точка синхронізації, яка пов’язана з очікуванням на подію, де **j** – номер потоку, у якій відбудеться подія; **k** – порядковий номер взаємодії цього потоку та потоку Тj.

Після аналізу схеми було вирішено:

* Використати AtomicInteger для атомарного отримання d та p
* Замінити S2,1; S3,1; S4,1; S1,1; S1,2; S2,2; S3,2; S4,2; S1,3 на CyclicBarrier CL1, перевикориставши його тричі для сигналізації потокам про закінчення введення або обрахунків.
* Для синхронізації доступу до z було використати семафор та Lock.

**Етап 4. Розробка програми**

Програма написана мовою Java і використовує різні механізми синхронізації потоків, зокрема семафори, критичні секції (локи), атомарні змінні та бар'єри.

### **Структура програми:**

* **Main** – головний клас, який відповідає за ініціалізацію та запуск усіх потоків. Тут також задаються загальні параметри, такі як розмірність N, імена потоків.
* **Data** – допоміжний клас, що містить статичні методи для виконання основних обчислень та обробки даних у потоках. Він також зберігає спільні ресурси, які використовуються всіма потоками, та реалізує операції множення, додавання, сортування, пошуку мінімум у векторі, обчислення заданої формули.
* **MainThread** – базовий клас, від якого успадковуються всі потоки.
* **Size** – утилітарний клас, який використовується для розрахунків, зберігаючи індекси початку та кінця елементів масиву, матриці, що беруть участь в обчисленнях.
* **T1, T2, T3, T4** – класи, що успадковуються від **MainThread** і реалізують алгоритм згідно з розподілом завдань між потоками та використанням механізмів синхронізації, які були розроблені на **Етапах 2 та 3**.

**Main.java**

*/\*\**

*\* Program header: «Програмне забезпечення високопродуктивних компʼютерних систем»*

*\* Lab 2: «Програмування для комп'ютерних систем зі спільною пам'яттю»*

*\* Function:*

*\* - МА= MD\*MC \*d + min(Z)\*MX\*p*

*\* Name: Галай Андрій Вікторович*

*\* Date: 19.03.2025*

*\*/*

package javaBasics;

import java.util.Arrays;

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

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

System.*out*.print("Enter N:");

int N = Integer.*parseInt*(scanner.nextLine());

Data.*setN*(N);

Thread t1 = new Thread(new T1("T1", N, 0));

Thread t2 = new Thread(new T2("T2", N, 1));

Thread t3 = new Thread(new T3("T3", N, 2));

Thread t4 = new Thread(new T4("T4", N, 3));

t1.start();

t2.start();

t4.start();

t3.start();

try {

t1.join();

t2.join();

t3.join();

t4.join();

System.*out*.println(Arrays.*deepToString*(Data.*MA*));

} catch (InterruptedException e) {

throw new RuntimeException(e);

}

}

}

**Data.java**

package javaBasics;

import java.util.Arrays;

import java.util.Random;

import java.util.Scanner;

import java.util.concurrent.CyclicBarrier;

import java.util.concurrent.Semaphore;

import java.util.concurrent.atomic.AtomicInteger;

import java.util.concurrent.locks.ReentrantLock;

public class Data {

private static final Random *random* = new Random();

public static final int *threadCount* = 4;

public static int *N* = 0;

public static AtomicInteger *d* = new AtomicInteger();

public static AtomicInteger *p* = new AtomicInteger();

public static int[] *Z*;

public static int[][] *MC*;

public static int[][] *MD*;

public static int[][] *MA*;

public static int[][] *MX*;

public static int *z* = Integer.*MAX\_VALUE*;

public static final Semaphore *consoleSemaphore* = new Semaphore(1);

public static final Semaphore *S2* = new Semaphore(1);

public static final CyclicBarrier *CL1* = new CyclicBarrier(Data.*threadCount*);

private static final ReentrantLock *L1* = new ReentrantLock();

public static void setN(int N) {

Data.*N* = N;

Data.*MA* = new int[N][N];

}

public static int readScalar() {

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

int result;

while (true) {

String value = scanner.nextLine();

try {

result = Integer.*parseInt*(value);

break;

} catch (NumberFormatException e) {

System.*out*.println("Invalid integer input. Please enter a valid integer.");

}

}

return result;

}

public static int[] readVector(int n) {

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

String vector = scanner.nextLine();

String[] elements = vector.split("\\s+");

int [] res = new int[n];

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

res[i] = Integer.*parseInt*(elements[i]);

}

return res;

}

public static int[][] readMatrix(int n, int m, String threadName) {

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

for (int i = 0; i < n; i++) {

System.*out*.print(threadName + " Row " + (i + 1) + ": ");

int[] row = Data.*readVector*(n);

matrix[i] = row;

}

return matrix;

}

public static int findMin(int[] vector) {

if (vector.length == 0) {

System.*out*.println("Length of vector has to be more that 0");

return -1;

}

int min = vector[0];

for (int i = 1; i < vector.length; i++) {

if (vector[i] < min) {

min = vector[i];

}

}

return min;

}

public static void setMinZ(int zi) {

Data.*L1*.lock();

Data.*z* = Math.*min*(Data.*z*, zi);

Data.*L1*.unlock();

}

public static void readD() {

int temp = Data.*readScalar*();

Data.*d*.set(temp);

}

public static void getRandomD() {

int temp = *getRandomInt*(-1000, 1000);

Data.*d*.set(temp);

}

public static void readP() {

int temp = Data.*readScalar*();

Data.*p*.set(temp);

}

public static void getRandomP() {

int temp = *getRandomInt*(-1000, 1000);

Data.*p*.set(temp);

}

public static void readZ(int n) {

Data.*Z* = Data.*readVector*(n);

}

public static Size getSize(int N, int threadCount, int threadId) {

int itemsCount = N / threadCount; // will be rounded to lower number

int start = threadId \* itemsCount;

boolean isLastThread = (threadCount - 1) == threadId;

int end = start + itemsCount;

if (isLastThread) {

int remainingItemsCount = N % threadCount;

end += remainingItemsCount;

}

return new Size(start, end);

}

public static int[] getSubArr(int[] arr, int threadCount, int threadId) {

Size size = Data.*getSize*(arr.length, threadCount, threadId);

return Arrays.*copyOfRange*(arr, size.getStart(), size.getEnd());

}

public static void calculateRows(int threadCount, int threadId, int d, int z, int p) {

Size size = Data.*getSize*(Data.*N*, threadCount, threadId);

for (int i = size.getStart(); i < size.getEnd(); i++) {

int[] res = new int[Data.*N*];

for (int j = 0; j < Data.*N*; j++) {

for (int k = 0; k < Data.*N*; k++) {

res[j] += Data.*MD*[k][j] \* Data.*MC*[j][k];

}

System.*out*.println("z = " + z);

System.*out*.println("res[i][j] = " + res[j] + " + " + z \* Data.*MX*[i][j] \* p);

res[j] += z \* Data.*MX*[i][j] \* p;

}

Data.*MA*[i] = res;

}

}

public static int getRandomInt(int min, int max) {

return *random*.nextInt(max - min + 1) + min;

}

public static int[] getRandomArray(int n, int min, int max) {

int[] arr = new int[n];

for (int i = 0; i < n; i++) {

arr[i] = *getRandomInt*(min, max);

}

return arr;

}

public static int[][] getRandomMatrix(int n, int min, int max) {

int[][] matrix = new int[n][n];

for (int i = 0; i < n; i++) {

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

matrix[i][j] = *getRandomInt*(min, max);

}

}

return matrix;

}

}

**MainThread.java**

package javaBasics;

import java.util.concurrent.locks.Condition;

import java.util.concurrent.locks.Lock;

import java.util.concurrent.locks.ReentrantLock;

public class MainThread {

private final String name;

public final int N;

public final int threadId;

private final Lock lock = new ReentrantLock();

private final Condition condition = lock.newCondition();

public MainThread(String name, int N, int threadId) {

this.name = name;

this.N = N;

this.threadId = threadId;

}

public String getName() {

return "[" + name + "]";

}

}

**Size.java**

package javaBasics;

public class Size {

int start;

int end;

public Size(int start, int end) {

this.start = start;

this.end = end;

}

public int getStart() {

return start;

}

public int getEnd() {

return end;

}

}

**T1.java**

package javaBasics;

import java.util.concurrent.BrokenBarrierException;

public class T1 extends MainThread implements Runnable {

public T1(String name, int N, int threadId) {

super(name, N, threadId);

}

@Override

public void run() {

// Введення даних

try {

if (N < 4) {

Data.*consoleSemaphore*.acquire();

System.*out*.print(this.getName() + " Enter d: ");

Data.*readD*();

} else {

Data.*getRandomD*();

}

} catch (InterruptedException e) {

throw new RuntimeException(e);

} finally {

Data.*consoleSemaphore*.release();

}

// Сигнал про закінчення введення даних

try {

Data.*CL1*.await();

} catch (InterruptedException | BrokenBarrierException e) {

throw new RuntimeException(e);

}

int[] subZ = Data.*getSubArr*(Data.*Z*, Data.*threadCount*, this.threadId);

if (subZ.length > 0) {

int minZ = Data.*findMin*(subZ);

// КД1

Data.*setMinZ*(minZ);

}

// Сигнал про закінчення обчислення zi

try {

Data.*CL1*.await();

} catch (InterruptedException | BrokenBarrierException e) {

throw new RuntimeException(e);

}

// КД 2

int d1 = Data.*d*.get();

int z1;

try {

// КД 3

Data.*S2*.acquire();

z1 = Data.*z*;

} catch (InterruptedException e) {

throw new RuntimeException(e);

} finally {

Data.*S2*.release();

}

// КД 4

int p1 = Data.*p*.get();

// Обчислення MAh

Data.*calculateRows*(Data.*threadCount*, this.threadId, d1, z1, p1);

try {

// Повідомлення про закінчення обчислення MAh

Data.*CL1*.await();

} catch (InterruptedException | BrokenBarrierException e) {

throw new RuntimeException(e);

}

System.*out*.println("Thread " + this.getName() + " finished");

}

}

**T2.java**

package javaBasics;

import java.util.Arrays;

import java.util.concurrent.BrokenBarrierException;

public class T2 extends MainThread implements Runnable {

public T2(String name, int N, int threadId) {

super(name, N, threadId);

}

@Override

public void run() {

// Введення даних

try {

if (N < 4) {

Data.*consoleSemaphore*.acquire();

System.*out*.println(this.getName() + " Enter MC:");

Data.*MC* = Data.*readMatrix*(this.N, this.N, this.getName());

System.*out*.println(this.getName() + " Enter MD:");

Data.*MD* = Data.*readMatrix*(this.N, this.N, this.getName());

} else {

Data.*MC* = Data.*getRandomMatrix*(N, -1000, 1000);

Data.*MD* = Data.*getRandomMatrix*(N, -1000, 1000);

}

} catch (InterruptedException e) {

throw new RuntimeException(e);

} finally {

Data.*consoleSemaphore*.release();

}

// Сигнал про закінчення введення даних

try {

Data.*CL1*.await();

} catch (InterruptedException | BrokenBarrierException e) {

throw new RuntimeException(e);

}

int[] subZ = Data.*getSubArr*(Data.*Z*, Data.*threadCount*, this.threadId);

if (subZ.length > 0) {

int minZ = Data.*findMin*(subZ);

// КД 1

Data.*setMinZ*(minZ);

}

// Сигнал про закінчення обчислення zi

try {

Data.*CL1*.await();

} catch (InterruptedException | BrokenBarrierException e) {

throw new RuntimeException(e);

}

// КД 2

int d2 = Data.*d*.get();

int z2;

try {

// КД 3

Data.*S2*.acquire();

z2 = Data.*z*;

} catch (InterruptedException e) {

throw new RuntimeException(e);

} finally {

Data.*S2*.release();

}

// КД4

int p2 = Data.*p*.get();

// Обчислення MAh

Data.*calculateRows*(Data.*threadCount*, this.threadId, d2, z2, p2);

try {

// Повідомлення про закінчення обчислення MAh

Data.*CL1*.await();

} catch (InterruptedException | BrokenBarrierException e) {

throw new RuntimeException(e);

}

System.*out*.println("Thread " + this.getName() + " finished");

}

}

**T3.java**

package javaBasics;

import java.util.Arrays;

import java.util.concurrent.BrokenBarrierException;

public class T3 extends MainThread implements Runnable {

public T3(String name, int N, int threadId) {

super(name, N, threadId);

}

@Override

public void run() {

// Введення даних

try {

if (N < 4) {

Data.*consoleSemaphore*.acquire();

System.*out*.print(this.getName() + " Enter p: ");

Data.*readP*();

} else {

Data.*getRandomP*();

}

} catch (InterruptedException e) {

throw new RuntimeException(e);

} finally {

Data.*consoleSemaphore*.release();

}

// Сигнал про закінчення введення даних

try {

Data.*CL1*.await();

} catch (InterruptedException | BrokenBarrierException e) {

throw new RuntimeException(e);

}

int[] subZ = Data.*getSubArr*(Data.*Z*, Data.*threadCount*, this.threadId);

if (subZ.length > 0) {

int minZ = Data.*findMin*(subZ);

// КД1

Data.*setMinZ*(minZ);

}

// Сигнал про закінчення обчислення zi

try {

Data.*CL1*.await();

} catch (InterruptedException | BrokenBarrierException e) {

throw new RuntimeException(e);

}

// КД 2

int d3 = Data.*d*.get();

int z3;

try {

// КД 3

Data.*S2*.acquire();

z3 = Data.*z*;

} catch (InterruptedException e) {

throw new RuntimeException(e);

} finally {

Data.*S2*.release();

}

// КД4

int p3 = Data.*p*.get();

// Обчислення MAh

Data.*calculateRows*(Data.*threadCount*, this.threadId, d3, z3, p3);

try {

// Повідомлення про закінчення обчислення MAh

Data.*CL1*.await();

} catch (InterruptedException | BrokenBarrierException e) {

throw new RuntimeException(e);

}

System.*out*.println("Thread " + this.getName() + " finished");

}

}

**T4.java**

package javaBasics;

import java.util.Arrays;

import java.util.concurrent.BrokenBarrierException;

public class T4 extends MainThread implements Runnable {

public T4(String name, int N, int threadId) {

super(name, N, threadId);

}

@Override

public void run() {

// Введення даних

try {

if (N < 4) {

Data.*consoleSemaphore*.acquire();

System.*out*.print(this.getName() + " Enter vector Z: ");

Data.*readZ*(this.N);

System.*out*.println(this.getName() + " Enter MX:");

Data.*MX* = Data.*readMatrix*(this.N, this.N, this.getName());

} else {

Data.*Z* = Data.*getRandomArray*(N, -1000, 1000);

Data.*MX* = Data.*getRandomMatrix*(N, -1000, 1000);

}

} catch (InterruptedException e) {

throw new RuntimeException(e);

} finally {

Data.*consoleSemaphore*.release();

}

// Сигнал про закінчення введення даних

try {

Data.*CL1*.await();

} catch (InterruptedException | BrokenBarrierException e) {

throw new RuntimeException(e);

}

int[] subZ = Data.*getSubArr*(Data.*Z*, Data.*threadCount*, this.threadId);

if (subZ.length > 0) {

int minZ = Data.*findMin*(subZ);

// КД1

Data.*setMinZ*(minZ);

}

// Сигнал про закінчення обчислення zi

try {

Data.*CL1*.await();

} catch (InterruptedException | BrokenBarrierException e) {

throw new RuntimeException(e);

}

// КД 2

int d4 = Data.*d*.get();

int z4;

try {

// КД 3

Data.*S2*.acquire();

z4 = Data.*z*;

} catch (InterruptedException e) {

throw new RuntimeException(e);

} finally {

Data.*S2*.release();

}

// КД4

int p4 = Data.*p*.get();

// Обчислення MAh

Data.*calculateRows*(Data.*threadCount*, this.threadId, d4, z4, p4);

try {

// Повідомлення про закінчення обчислення MAh

Data.*CL1*.await();

} catch (InterruptedException | BrokenBarrierException e) {

throw new RuntimeException(e);

}

System.*out*.println("Thread " + this.getName() + " finished");

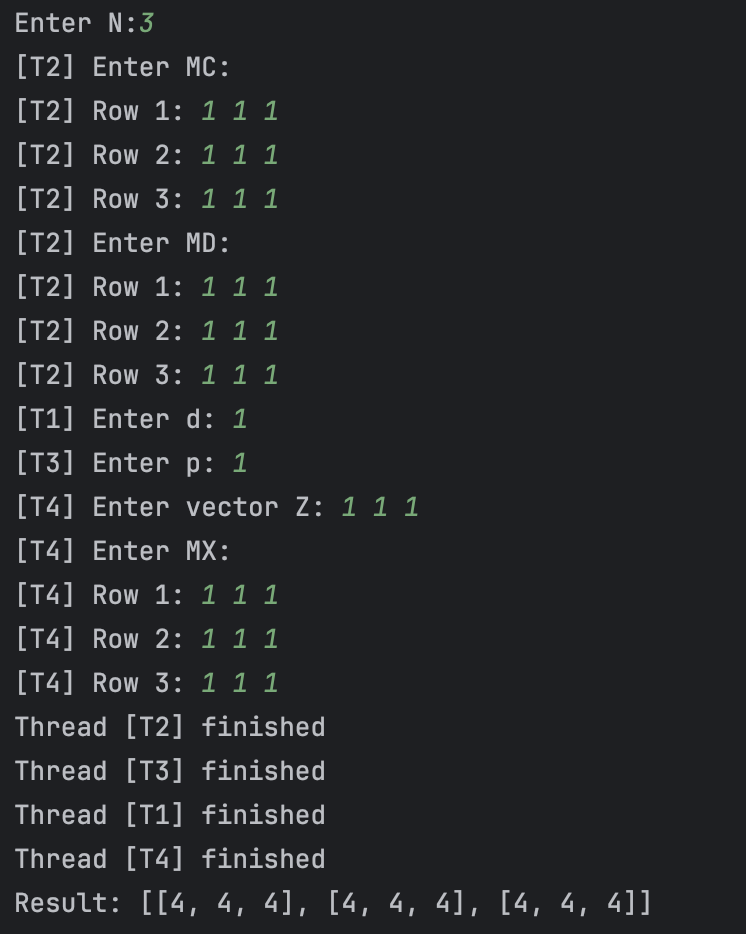
}

}

**Етап 5. Тестування програми для різних значень N**

**Тестування 1**

N=3



Перевірка коректності результатів:



Відповідь програми збігається із результатом отриманим на калькуляторі для однакових вхідних даних.

**Тестування 2**

N=1000

Для виконання лабораторної роботи було використано компʼютер із операційною системою macOS, тому для того, щоб обмежити кількість задіяних ядер процесора було використано Docker.

Було створено .jar файл розробленої програми та запущено програму в Docker контейнері.

Dockerfile

FROM openjdk:latest

COPY out/artifacts/2\_lab\_jar/2-lab.jar app.jar

CMD ["java", "-jar", "app.jar"]

Для створення Docker image було використано команду:

*docker build -t my-java-app .*

Для корегування кількості ядер, які дозволено використовувати докер контейнору, використано прапорець --cpus.

Для тестування програми на одному ядрі використано команду:

*docker run --cpus=1 -it my-java-app*

Для тестування програми на чотирьох ядрах використано команду:

*docker run --cpus=4 -it my-java-app*

Результати тестування

Для N=1000 на одному ядрі t1 = 4437ms.

Для N=1000 на чотирьох ядрах t4 = 961ms.

Коефіцієнт прискорення = КП = 4437 / 961 = 4,62.

**Висновок**

1. Було розроблено алгоритм паралельного обчислення виразу МА= MD\*MC \*d + min(Z)\*MX\*p та визначено критичні ділянки та спільні ресурси, а саме z - одному випадку спільний ресурс, доступ до якого потрібно контролювати, в іншому спільний ресурс, який копіювати у критичних ділянках; d, p - спільний ресурс, який копіювати у критичних ділянках; MD - спільний ресурс, який не змінюється під час доступу потоків до нього, тому доступ до нього можна не контролювати.
2. Було розроблено алгоритм для кожного із потоків із з визначенням критичних ділянок (КД) і точок синхронізації, який у подальшому використовувався для розробки структурної схеми взаємодії потоків та розробки програми: КД1 - обчислення мінімального значення вектора Z, КД2, КД3 та КД4 для копіювання значеня d, z, p відповідно.
3. Було розроблено структурну схему взаємодії потоків, із зображенням засобів синхронізації потоків. Схема містить спільні ресурси, а також потоки, де позначено точки синхронізації (S, W). Структурну схему у подальшому було використано для розробки програми обчислення заданого виразу.
4. Було розроблено програму, що виконує паралельні обчислення заданої формули (МА= MD\*MC \*d + min(Z)\*MX\*p), що складається із допоміжного класу Data, у якому містяться всі засоби синхронізації та дані, клас Main, що служить точкою входу в програму, допоміжний клас Size, що використовувався при обчисленнях, клас MainThread, що є батьківським класом для класів T1, T2, T3, T4, де і було реалізовано алгоритм розроблений у попередніх етапах для кожного із потоків.
5. Було проведено тестування програми для різних значень N. Було перевірено коректність розрахунків програми за допомогою порівняння результатів виконання програми для N = 3 та калькулятором. Було виконано тестування програми для N = 1000 із використанням 1 та 4 ядер. Час обчислень із використанням 4 ядер становив **961ms**, час обчислень із використанням 1 ядра становив **4437ms**. Було визначено коефіцієнт прискорення КП = 4437 / 961 = 4,62.