МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

ІНСТИТУТ ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ

Кафедра системного проектування

**ЛАБОРАТОРНА РОБОТА №1**

з дисципліни: «ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ»

Виконав:

студент 3 курсу

групи ДА-81

Головін Б. О.

Варіант 5

Київ – 2021

***Лабораторна робота 1***

***Мета роботи:***

Розробка і реалізація паралельного алгоритму для задач із паралелізмом даних.

***Завдання*** Варіант 5

Створити вектор з N>=10000 елементами з випадкових чисел. Знайти максимальний та мінімальний елементи векторів.

***Короткі теоретичні відомості***

Багатозадачність в Java реалізує модель системи зі спільною пам'яттю. Основною обчислювальною одиницею є так званий «потік». Програма на мові Java може складатись з декількох (не менше одного) потоків, які виконуються паралельно і мають спільний адресний простір.

Будь-яка програма за замовчуванням вже має один потік - так званий головний потік (Main Thread), з якого починається виконання програми. В процесі свого функціонування програма може ініціювати створення нових потоків та завершувати існуючі. Програма вважається «живою» поки живий хоча б один її потік.

Створити потік можна **двома способами**:

1. Створити клас, що наслідує клас Thread і перевизначити метод run().

2. Створити екземпляр класу Thread і передати йому як параметр щось, що реалізує інтерфейс Runnable, що містить один метод - run().

Обидва методи приводять до однакового результату і можуть використовуватися на рівні залежності від зручності використання того чи іншого способу в умовах конкретної задачі.

Для того щоб виконати код, який міститься в методі **run()** у паралельному потоці потрібно викликати метод **start()** класу Thread, який ініціює створення потоку та починає виконання відповідного коду.

Класс Thread має цілу низку методів для роботи з потоками, які будуть детально розглянуті в лабораторній роботі №2. Для виконання ж даної роботи буде достатньо розглянути лише деякі з них:

- **start**() - як вже говорилось, створює новий потік та запускає на виконання відповідний код, який міститься в методі **run()**.

- **destroy**() - знищує потік без очищення даних, що до нього відносяться.

- **join**() - зупиняє виконання потоку, який викликав даний метод до завершення виконання потоку, для якого було викликано join. Іншими словами join змушує один потік чекати завершення обчислень в іншому. Даний метод використовується, наприклад, при організації паралельних обчислень, коли після виконання роботи у всіх потоках потрібно провести агрегацію результатів та сформувати остаточну відповідь. Так як потік автоматично завершується по закінченню набору інструкцій, які він мав виконати, потрібно якось змусити потік-агрегатор дочекатись завершення робочих потоків, щоб отримати результати роботи, для чого і доцільно використати join().

**Лістинг програми**

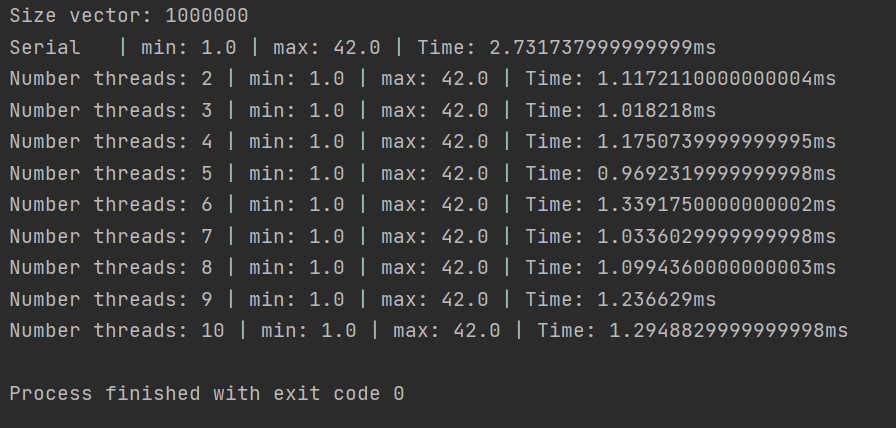
***threadCalculate.java***

class threadCalculate extends Thread {  
  
 double[] vector;  
 int startIndex;  
 int endIndex;  
 double min;  
 double max;  
  
 public threadCalculate(double[] vector, int startIndex, int endIndex) { //конструктор класу, приймає дані для обчислень  
 this.vector = vector;  
 this.startIndex = startIndex;  
 this.endIndex = endIndex;  
 }  
  
 public double getMin() {  
 return min;  
 }  
  
 public double getMax() {  
 return max;  
 }  
  
 @Override  
 public void run() { //обрахунки, що здійснюватимуться в зазначеному потоці  
 min = vector[startIndex];  
 max = vector[startIndex];  
 for (int i = startIndex + 1; i < endIndex; i++) {  
 if (vector[i] > max)  
 max = vector[i];  
 if (vector[i] < min)  
 min = vector[i];  
 }  
 }  
}

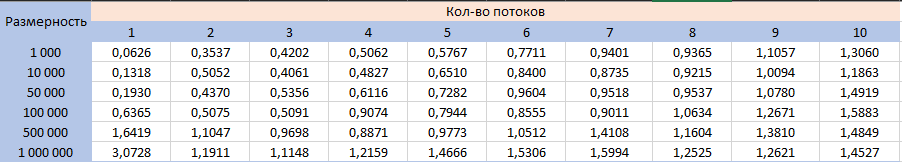
***threadSample.java***

public class threadSample {  
 public static int *SIZE* = 1000000;  
 public static int *NUMBER\_THREADS* = 2;  
  
 public static void main(String[] args) throws InterruptedException {  
  
 double[] vector = new double[*SIZE*];  
 int rand\_min = 1;  
 int rand\_max = 42;  
 for (int i = 0; i < *SIZE*; i++) { //початкове заповнення векторів випадковими величинами з зазначеного проміжку  
 vector[i] = rand\_min + (int) (Math.*random*() \* rand\_max);  
 }  
  
 System.*out*.println("Size vector: " + *SIZE*);  
  
 double startTime, finalTime, totalTime = 0;  
  
 // Serial  
 double maxSerial = 0, minSerial = 0;  
  
 for (int x = 0; x < 100; x++) {  
  
 startTime = System.*nanoTime*();  
 maxSerial = vector[0];  
 minSerial = vector[0];  
 for (int i = 1; i < *SIZE*; i++) {  
 if (vector[i] > maxSerial)  
 maxSerial = vector[i];  
 if (vector[i] < minSerial)  
 minSerial = vector[i];  
 }  
 finalTime = (System.*nanoTime*() - startTime) / 1000000;  
  
 totalTime += finalTime;  
  
 }  
 System.*out*.println("Serial | min: " + minSerial + " | max: " + maxSerial + " | Time: " + totalTime / 100 + "ms");  
  
 // Parallels  
 for (*NUMBER\_THREADS* = 2; *NUMBER\_THREADS*<=10;*NUMBER\_THREADS*++) {  
 totalTime = 0;  
  
 double maxParallel = 0, minParallel = 0;  
  
 for (int x = 0; x < 100; x++) {  
  
 startTime = System.*nanoTime*();  
  
 threadCalculate[] treadArray = new threadCalculate[*NUMBER\_THREADS*];  
  
 for (int i = 0; i < *NUMBER\_THREADS*; i++) { //розбиття на потоки  
 treadArray[i] = new threadCalculate(vector, *SIZE* / *NUMBER\_THREADS* \* i,  
 i == (*NUMBER\_THREADS* - 1) ? *SIZE* : *SIZE* / *NUMBER\_THREADS* \* (i + 1)); //тернарна умовна операція  
 treadArray[i].start();  
 }  
 for (int i = 0; i < *NUMBER\_THREADS*; i++) { //очікування завершення усіх потоків  
 treadArray[i].join();  
 }  
  
 maxParallel = treadArray[0].getMax();  
 minParallel = treadArray[0].getMin();  
 for (int i = 0; i < *NUMBER\_THREADS*; i++) { //збір результатів паралельної роботи  
 if (treadArray[i].getMax() > maxParallel)  
 maxParallel = treadArray[i].getMax();  
 if (treadArray[i].getMin() < minParallel)  
 minParallel = treadArray[i].getMin();  
 }  
  
 finalTime = (System.*nanoTime*() - startTime) / 1000000;  
 if (x != 0)  
 totalTime += finalTime;  
 }  
  
 System.*out*.println("Number threads: " + *NUMBER\_THREADS* + " | min: " + minParallel + " | max: " + maxParallel + " | Time: " + totalTime / 100 + "ms");  
 }  
 }  
}

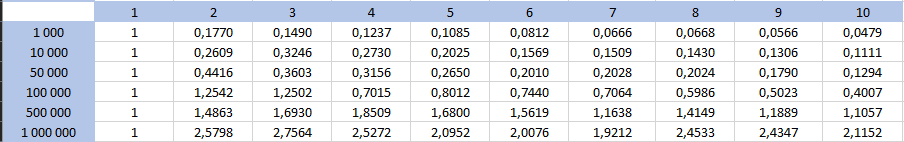
**Результати роботи програми**

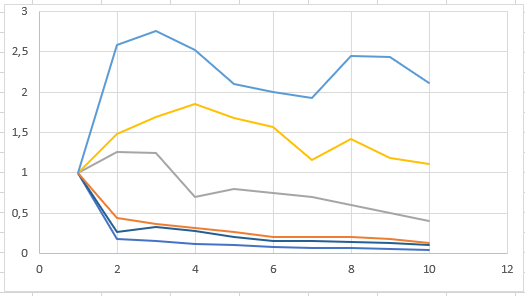
****

*Таблиця швидкості роботи програми в залежності від кількості вхідних даних та кількості потоків*

**

*Таблиця, яка показує, у скільки разів швидше працює програма від послідовної роботи*

****

****

**Висновки**

У даній лабораторній роботі ми ознайомитись з поняттями багатозадачності у Java. А також проводилась розробка і реалізація паралельного алгоритму для задач із паралелізмом даних. За результатами було проаналізовано швидкості при різних кількостях вхідних даних та кількості потоків.