Національний Технічний Університет України

«Київський Політехнічний Інститут імені Ігоря Сікорського» Інститут Прикладного Системного Аналізу

Кафедра Системного Проектування

Паралельні обчислення

Лабораторна робота №1

**Роботу виконав:** Нікітін Б. Д.

Група: ДА­81

**Перевірив:**

Яременко В. С.

Київ – 2021

# Мета роботи

Розробка і реалізація паралельного алгоритму для задач із паралелізмом да­ них.

# Завдання

1. Розробити послідовну та багатопоточну програми які реалізують варіант індивідуального завдання (мова програмування обирається студентом).
2. Порівняти правильність виконання, порівнявши послідовний та паралельний розв’язки.
3. Виміряти час розрахунку для послідовного та паралельного розв’язків при різних значеннях *SIZE* та *NUMBER*\_*THREADS* для власного варіанту, знайти значення при яких кожен з розв’язків (послідовний чи паралельний) буде виконуватися швидше за інший, зробити таблицю та графічно представити результат.

# Варіант 21



# Програмна реалізація (Thread Tests – main class)

**ThreadTests.java**

import java.util.Arrays;  
  
public class ThreadTests {  
 public static int[] *SIZE* = {100, 200, 500, 1000, 2000};  
 public static int[] *NUNMBER\_THREADS* = {3, 8};  
 public static int *NUMBER\_TESTS* = 100;  
  
 public static void main(String [] args ) throws InterruptedException {  
 long[] workTime = new long[*NUMBER\_TESTS*];  
 for (int size: *SIZE*){  
 double[][] matrixSerial = new double[size][size];  
 double[][] matrixParallel = new double[size][size];  
 double[][] resultMatrixSerial = new double[size][size];  
 double[][] resultMatrixParallel = new double[size][size];  
 *generateMatrix*(matrixSerial, matrixParallel, size);  
 System.*out*.println("Matrix size : " + size + "x" + size);  
 for (int i=0; i< *NUMBER\_TESTS*; i++){  
 ThreadSingle single = new ThreadSingle(matrixSerial, size);  
 single.run();  
 workTime[i] = single.getTime();  
 if (!Arrays.*deepEquals*(resultMatrixSerial, single.getMatrix())){  
 if (i != 0) System.*out*.println(i + " test of single thread program made wrong matrix");  
 resultMatrixSerial = single.getMatrix();  
 }  
 }  
 System.*out*.println("1 thread work time : " + Arrays.*stream*(workTime).sum() / *NUMBER\_TESTS*);  
 for (int num: *NUNMBER\_THREADS*){  
 for (int i=0; i< *NUMBER\_TESTS*; i++) {  
 ThreadMultiply multiply = new ThreadMultiply(matrixParallel, size, num);  
 multiply.run();  
 workTime[i] = multiply.getTime();  
 if (!Arrays.*deepEquals*(resultMatrixParallel, multiply.getMatrix())){  
 if (i != 0) System.*out*.println(i + " test of " + num + " threads program made wrong matrix");  
 resultMatrixParallel = multiply.getMatrix();  
 }  
 }  
 if (!Arrays.*deepEquals*(resultMatrixParallel, resultMatrixSerial)){  
 System.*out*.println(num + " threads test program made wrong matrix");  
 }  
 System.*out*.println(num + " threads work time : " + Arrays.*stream*(workTime).sum() / *NUMBER\_TESTS*);  
 }  
 }  
 }  
  
 static void generateMatrix(double[][] matrixA, double[][] matrixB, int size){  
 int rand\_min=1;  
 int rand\_max=42;  
 for(int i =0; i<size; i++){//початкове заповнення матриць випадковими величинами з зазначеного проміжку  
 for(int j =0; j<size; j++){  
 matrixA[i][j] = rand\_min + (int) (Math.*random*() \* rand\_max);  
 matrixB[i][j] = matrixA[i][j];  
 }  
 }  
 }  
}

**ThreadSingle.java**

import java.util.Arrays;  
  
public class ThreadSingle {  
 double[][] matrix;  
 int size;  
 long workTime = 0;  
  
 public ThreadSingle(double[][] matrix, int size) {  
 this.matrix = Arrays.*stream*(matrix)  
 .map((double[] row) -> row.clone())  
 .toArray((length) -> new double[length][]);  
 this.size = size;  
 }  
  
 public long getTime(){  
 return workTime;  
 }  
  
 public double[][] getMatrix(){  
 return matrix;  
 }  
  
 public void run(){  
 long startTime;  
 long finishTime;  
 startTime = System.*nanoTime*();  
 int maxIndex;  
 for( int i=0; i< this.size; i++){  
 maxIndex = i;  
 for( int j=0; j< this.size; j++){ // шукаємо індекс максимального елемента  
 maxIndex = this.matrix[i][j] > this.matrix[i][maxIndex] ? j : maxIndex;  
 }  
 if (maxIndex != i){ // якщо максимальний елемент не знаходиться на діагоналі, переміщуємо його туди  
 double swap = this.matrix[i][i];  
 this.matrix[i][i] = this.matrix[i][maxIndex];  
 this.matrix[i][maxIndex] = swap;  
 }  
 }  
 finishTime = System.*nanoTime*();  
 this.workTime = (finishTime - startTime) / 1000;  
 }  
}

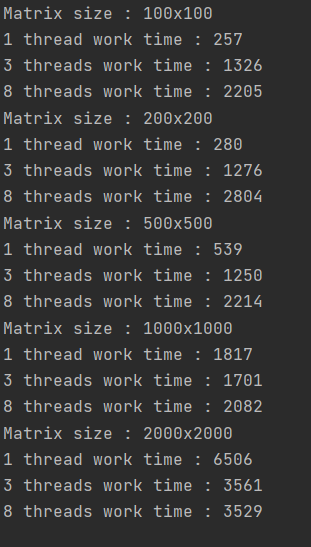
**ThreadMultiply.java**

import java.util.Arrays;  
  
public class ThreadMultiply {  
 int numberThreads;  
 int size;  
 double[][] matrix;  
 long workTime;  
  
 public ThreadMultiply(double[][] matrix, int size, int numberThreads) {  
 this.matrix = Arrays.*stream*(matrix)  
 .map((double[] row) -> row.clone())  
 .toArray((length) -> new double[length][]);  
 this.size = size;  
 this.numberThreads = numberThreads;  
 }  
  
 public long getTime(){  
 return workTime;  
 }  
  
 public double[][] getMatrix(){  
 return matrix;  
 }  
  
 public void run() throws InterruptedException {  
  
  
 long startTime;  
 long finishTime;  
 ThreadCacl[] TreadArray = new ThreadCacl[numberThreads];  
 for(int i = 0; i < size % numberThreads; i++){ //розбиття на потоки  
 TreadArray[i] = new ThreadCacl(matrix ,  
 (size/numberThreads + 1) \* i,  
 (size/numberThreads + 1));  
 }  
 for(int i = size % numberThreads; i < numberThreads; i++){ //розбиття на потоки  
 TreadArray[i] = new ThreadCacl(matrix ,  
 size/numberThreads \* i + size % numberThreads,  
 size/numberThreads);  
 }  
 startTime = System.*nanoTime*();  
 for(int i = 0; i < numberThreads; i++){ //старт потоків  
 TreadArray[i].start();  
 }  
 for(int i = 0; i < numberThreads; i++){ //очікування завершення усіх потоків  
 TreadArray[i].join();  
 }  
 finishTime = System.*nanoTime*();  
 workTime = (finishTime - startTime) / 1000;  
 }  
}

**ThreadCacl.java**

class ThreadCacl extends Thread{  
  
 double[][] matrix;  
 int startIndex;  
 int numOfRows;  
  
 public ThreadCacl(double[][] matrix, int startIndex, int numOfRows) {//конструктор класу, приймає дані для обчислень  
 this.matrix = matrix;  
 this.startIndex = startIndex;  
 this.numOfRows = numOfRows;  
 }  
  
 @Override  
 public void run(){ //обрахунки, що здійснюватимуться в зазначеному потоці  
 int maxIndex;  
 for( int i=startIndex; i< (numOfRows + startIndex); i++){  
 maxIndex = i;  
 for( int j=0; j< matrix[i].length; j++){  
 maxIndex = matrix[i][j] > matrix[i][maxIndex] ? j : maxIndex;  
 }  
 if (maxIndex != i){  
 double swap = matrix[i][i];  
 matrix[i][i] = matrix[i][maxIndex];  
 matrix[i][maxIndex] = swap;  
 }  
 }  
 }  
}

# Результати роботи програми та їх правильність



На скріншоті вище зображено часові дані роботи програми з різними вхідними даними.

Правильність роботи програми перевіряється за допомогою Arrays.deepEqual() порівняння оброблених матриць.

Також я проводжу 100 тестів, щоб часові дані були усереднені.

# Обробка результатів

|  |  |  |  |
| --- | --- | --- | --- |
| **Size** | **1 thread** | **3 threads** | **8 threads** |
| 100 | 257 | 1326 | 2205 |
| 200 | 280 | 1276 | 2804 |
| 500 | 539 | 1250 | 2214 |
| 1000 | 1817 | 1701 | 2082 |
| 2000 | 6506 | 3561 | 3529 |

З таблиці вище ми бачимо, що для матриць розміром менше 1000х1000 паралельний варіант виконання програми не є ефективним. Це може бути через втрати на ініціалізацію потоків, що є більшими ніж затрати на виконання поставленої задачі.

**Висновки**

У даній роботі я розробив програму, що реалізує варіант моєї задачі послідовно і паралельно. Також я порівняв результати, щоб впевнитися, що програми реалізовані правильно. У результаті я побачив, що паралелізм не завжди є ефективним і перед його використанням треба впевнитися у його необхідності.