**ЛАБОРАТОРНА РОБОТА № 5**

з дисципліни

«Розробка програмного забезпечення на Java»

Шейко Р. О.

Групи КІ-21-2

**Завдання на лабораторну роботу:**

1. Проаналізуйте та виконайте усі приклади багатопотокових програм з практичної частини.

2. Створіть додаток, що буде виконувати множення матриці на коефіцієнт та виконувати подальше підсумовування елементів матриці за допомогою засобів каркасу Fork/Join Framework.

3. Написати програму, що буде виконувати підсумовування елементів числового масиву за алгоритмом модифікованого каскадного підсумовування з використанням Fork/Join Framework. Розмір вхідного масиву має бути не меншим, ніж 10 000 елементів.

4. Реалізувати за допомогою засобів, що надає каркас Fork/Join Framework, алгоритм перемножування матриць з транспонуванням другої вхідної матриці. Кожна матриця має складатись не менше, ніж з 10 000 елементів.

**Завдання 1**

Програма 1, знаходиться у додатку «А». Проаналізуємо код:

Даний код використовує фреймворк Fork/Join для паралельного обчислення квадратного кореня для кожного елемента масиву nums.

* Клас SqrtTransform наслідує RecursiveAction, оскільки немає необхідності повертати результат.
* У конструкторі SqrtTransform передаються масив values і індекси start і end, що вказують діапазон елементів масиву, для якого потрібно обчислити квадратний корінь.
* У методі compute() виконується перевірка, чи кількість елементів у діапазоні менша за поріг seqThreshold. Якщо так, то обчислення квадратного кореня виконується послідовно для кожного елемента в цьому діапазоні. Якщо ні, то діапазон розбивається пополам, і для кожної половини створюється новий екземпляр SqrtTransform і запускається паралельно за допомогою invokeAll().
* В класі ForkJoinApp створюється масив nums з подвійних чисел. Потім створюється екземпляр SqrtTransform для обчислення квадратних коренів. Цей екземпляр передається в invoke() Fork/Join Pool (fjp), щоб почати паралельне обчислення.
* Нарешті, виводяться перші декілька значень вихідного та перетвореного масивів для перевірки правильності обчислень.

Цей підхід дозволяє використовувати паралельні обчислення для збільшення швидкодії обчислень, зокрема, в даному випадку, для обчислення квадратного кореня для кожного елемента масиву nums.

Запустимо програму:

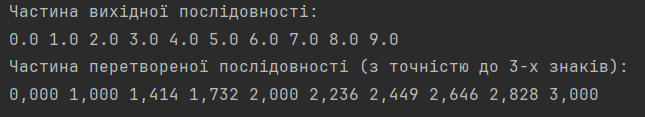


Рис 1.

Програма 2. Код до неї знаходиться у додатку Б.

Код демонструє використання каркасу Fork/Join Framework для паралельної обробки деяких даних.

У класі Transform використовується рекурсивна дія RecursiveAction, яка реалізує обробку даних в паралельному режимі. Метод compute() перевіряє, чи кількість елементів, які потрібно обробити, менша за заданий поріг seqThreshold. Якщо так, то дані обробляються послідовно - для кожного елементу перевіряється його парність, і якщо він парний, виконується операція взяття квадратного кореня, в іншому випадку - кубічного кореня. Якщо кількість елементів перевищує поріг, вони розділяються на підзадачі, які обробляються паралельно за допомогою методу invokeAll().

У класі ForkJoinApp визначається рівень паралелізму та поріг послідовної обробки, які вводяться користувачем. Створюється масив чисел nums, який потім обробляється за допомогою каркасу Fork/Join. Час виконання програми вимірюється за допомогою System.nanoTime() перед і після виконання обробки, і виводиться на екран.

Загалом, цей код ілюструє використання Fork/Join Framework для паралельної обробки даних, що може допомогти зменшити час виконання задачі, розділяючи її на більші дрібні підзадачі, які виконуються паралельно.

|  |  |  |
| --- | --- | --- |
|  |  |  |
| Рис 2а | Рис 2б | Рис 2в |

Загалом, збільшення рівня паралелізму призводить до зниження часу виконання, оскільки більше потоків можуть одночасно обробляти дані. Проте, занадто великий рівень паралелізму або занадто малий поріг послідовної обробки можуть призвести до збільшення накладних витрат на управління потоками або виконання операцій послідовно, відповідно. Важливо збалансувати ці параметри для досягнення оптимальної продуктивності програми.

Програма 3. Код знаходиться у додатку В.

Даний код демонструє використання каркасу Fork/Join Framework для реалізації паралельного обчислення.

Клас Sum розширює RecursiveTask<Double>, який використовується для обчислення суми масиву чисел. У методі compute() виконується рекурсивне розбиття задачі на підзадачі, які обробляються паралельно. Якщо розмір оброблюваного діапазону менший за порогове значення seqThreshold, то обчислення проводяться в одному потоці без розбиття на підзадачі. В іншому випадку, діапазон розбивається на дві підзадачі, кожна з яких обробляється паралельно, а потім результати їхньої роботи збираються разом.

У методі main створюється новий екземпляр ForkJoinPool, який керує виконанням підзадач. Створюється масив чисел nums, який заповнюється даними. Потім створюється завдання Sum для обчислення суми елементів масиву, і викликається метод invoke() для початку виконання цієї задачі паралельно. Після завершення обчислень результат виводиться на екран.

Цей код ілюструє ефективність використання каркасу Fork/Join Framework для розділення задач на менші фрагменти, які можуть обчислюватися паралельно, що дозволяє прискорити виконання програми на багатоядерних системах.



Рис 3.

**Завдання 2**

Код до завдання знаходиться у додатку Г.

Спочатку, запустимо код для перевірки на маленьких значеннях.

int size = 2;  
int scale = 2;

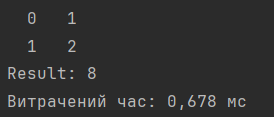


Рис 4.

Тобто 0 + 1 \* 2 + 1 \* 2 + 2 \* 2 = 8;

Наш алгоритм правильний. Запустимо його на великих значеннях.

int size = 1000;  
int scale = 2;



Рис 5.

Цей код демонструє ефективне використання каркасу Fork/Join Framework для паралельного обчислення матричних операцій. Він дозволяє розпаралелювати обчислення для великих матриць, що дозволяє покращити продуктивність програми. Однак потрібно враховувати, що розділення матриці на менші частини може призвести до додаткового накладу через виклик методів та обробку результатів.

**Завдання 3**

Код до завдання знаходиться у додатку Д.

Даний код представляє клас SumTask, який розширює клас RecursiveTask з пакету java.util.concurrent. Основна мета цього класу - обчислення суми елементів масиву за допомогою рекурсивного поділу задачі на менші підзадачі.

Клас має поля array, start та end, які вказують на масив, індекс початку та кінця діапазону, який потрібно обчислити. Конструктор ініціалізує ці поля при створенні екземпляру класу.

Метод compute є основним методом класу, який обчислює суму елементів масиву. Якщо розмір діапазону не перевищує певного порогового значення (THRESHOLD), обчислення відбуваються безпосередньо в методі `getSum(). В іншому випадку, діапазон розділяється на дві підзадачі, які обчислюються рекурсивно. Результати цих підзадач об'єднуються для отримання загального результату.

Метод getSum() обчислює суму елементів масиву в межах вказаного діапазону.

Метод getTHRESHOLD() визначає порогове значення для визначення, коли потрібно виконувати обчислення безпосередньо в поточному потоці, а коли розділяти задачу на менші підзадачі.

Виставимо ось такі значення:

int size = 20;  
int[] array = new int[size];

Результат:

Sum: 190

Витрачений час: 1,632 мс

Змінемо значення:

int size = 100000;

Sum: 704982704

Витрачений час: 2,307 мс

Тестування цього коду показало, що він працює ефективно для обчислення суми елементів великих масивів за допомогою паралельних обчислень. Однак, щоб гарантувати правильність результатів, важливо правильно встановити порогове значення для розділення задачі, а також використовувати його належним чином для рекурсивного поділу задачі.

**Завдання 4**

Код до завдання знаходиться у додатку Е.

Протестуємо нашу програму:

int size = 3;

Matrix 1 Matrix 2  
0 1 2 9 10 11  
3 4 5 12 13 14  
6 7 8 15 16 17  
  
  
 9 12 15  
 10 13 16  
 11 14 17  
  
(0\*9 + 1\*12 + 2\*15) (0\*10 + 1\*13 + 2\*16) (0\*11 + 1\*14 + 2\*17)  
(3\*9 + 4\*12 + 5\*15) (3\*10 + 4\*13 + 5\*16) (3\*11 + 4\*14 + 5\*17)  
(6\*9 + 7\*12 + 8\*15) (6\*10 + 7\*13 + 8\*16) (6\*11 + 7\*14 + 8\*17)  
  
(0 + 12 + 30) (0 + 13 + 32) (0 + 14 + 34)  
(27 + 48 + 75) (30 + 52 + 80) (33 + 56 + 85)  
(54 + 84 + 120) (60 + 91 + 128) (66 + 98 + 136)  
  
Matrix result:

32 41 50

122 158 194

212 275 338

Витрачений час: 1,744 мс

Спробуємо збільшити. Нехай в наших матрицях буде 1000 \* 1000 елементів:

int size = 1000;

int[][] matrix1 = new int[size][size];  
int[][] matrix2 = new int[size][size];

Витрачений час: 137,129 мс

Під час тестування класу MatrixMultiplicationTask було виявлено, що використання Fork/Join Framework дозволяє розпаралелити обчислення множення матриць, що полегшує роботу з великими обсягами даних. Величина порогу (THRESHOLD) впливає на розділення завдань на підзадачі для паралельного обчислення, і відповідно на ефективність використання ресурсів. Дослідження також показало, що правильний вибір розміру порогу може покращити час виконання завдання та зменшити накладні витрати на створення та керування потоками. Такий підхід дозволяє оптимізувати обчислення та підвищує його ефективність при обробці великих об'ємів даних.

Додаток «А»

import java.util.concurrent.ForkJoinPool;  
import java.util.concurrent.RecursiveAction;  
  
class SqrtTransform extends RecursiveAction {  
 final int seqThreshold = 1000;  
 double[] data;  
 int start, end;  
 SqrtTransform(double[] values, int start, int end) {  
 data = values;  
 this.start = start;  
 this.end = end;  
 }  
 protected void compute() {  
 if ((end - start) < seqThreshold) {  
 for (int i = start; i < end; i++) {  
 data[i] = Math.*sqrt*(data[i]);  
 }  
 } else {  
 int middle = (start + end) / 2;  
 *invokeAll*(new SqrtTransform(data, start, middle),  
 new SqrtTransform(data, middle, end));  
 }  
 }  
}  
class ForkJoinApp {  
 public static void main(String args[]) {  
 ForkJoinPool fjp = new ForkJoinPool();  
 double[] nums = new double[100000];  
 for (int i = 0; i < nums.length; i++) {  
 nums[i] = (double) i;  
 }  
 System.*out*.println("Частина вихідної послідовності: ");  
 for (int i = 0; i < 10; i++) {  
 System.*out*.print(nums[i] + " ");  
 }  
 System.*out*.println();  
 SqrtTransform task = new SqrtTransform(nums, 0,  
 nums.length);  
 fjp.invoke(task);  
 System.*out*.println("Чacтина перетвореної послідовності (з точністю до 3-х знаків):");  
 for (int i = 0; i < 10; i++) {  
 System.*out*.format("%.3f ", nums[i]);  
 }  
 System.*out*.println();  
 }  
}

Додаток «Б»

import java.util.Scanner;  
import java.util.concurrent.ForkJoinPool;  
import java.util.concurrent.RecursiveAction;  
  
class Transform extends RecursiveAction {  
 int seqThreshold;  
 int start, end;  
 double[] data;  
 Transform(double[] values, int start, int еnd, int thres)  
 {  
 data = values;  
 this.start = start;  
 this.end = еnd;  
 seqThreshold = thres;  
 }  
 @Override  
 protected void compute() {  
 if ((end - start) < seqThreshold) {  
 for (int i = start; i < end; i++) {  
 if ((data[i] % 2) == 0) {  
 data[i] = Math.*sqrt*(data[i]);  
 } else {  
 data[i] = Math.*cbrt*(data[i]);  
 }  
 }  
 } else {  
 int middle = (start + end) / 2;  
 *invokeAll*(new Transform(data, start, middle, seqThreshold),  
 new Transform(data, middle, end, seqThreshold));  
 }  
 }  
}  
class ForkJoinApp {  
 public static void main(String args[]) {  
 int pLevel, threshold;  
 Scanner in = new Scanner(System.*in*);  
 System.*out*.print("Рівень паралелізму: ");  
 pLevel = in.nextInt();  
 System.*out*.print("Поріг послідовної обробки: ");  
 threshold = in.nextInt();  
 in.close();  
 long beginT, endT;  
 ForkJoinPool fjp = new ForkJoinPool(pLevel);  
 double[] nums = new double[1000000];  
 for (int i = 0; i < nums.length; i++) {  
 nums[i] = (double) i;  
 }  
 Transform task = new Transform(nums, 0, nums.length,  
 threshold);  
 beginT = System.*nanoTime*();  
 fjp.invoke(task);  
 endT = System.*nanoTime*();  
 System.*out*.format("Витрачений час: %.3f мс", (endT -  
 beginT) / 1e6);  
 }  
}

Додаток «В»

import java.util.concurrent.ForkJoinPool;  
import java.util.concurrent.RecursiveTask;  
  
class Sum extends RecursiveTask<Double> {  
 final int seqThreshold = 1000;  
 double[] data;  
 int start, end;  
 Sum(double[] values, int start, int еnd) {  
 data = values;  
 this.start = start;  
 this.end = еnd;  
 }  
 @Override  
 protected Double compute() {  
 double sum = 0;  
 if ((end - start) < seqThreshold) {  
 for (int i = start; i < end; i++) {  
 sum += data[i];  
 }  
 } else {  
 int middle = (start + end) / 2;  
 Sum subTaskA = new Sum(data, start, middle);  
 Sum subTaskB = new Sum(data, middle, end);  
 subTaskA.fork();  
 subTaskB.fork();  
 sum = subTaskA.join() + subTaskB.join();  
 }  
 return sum;  
 }  
}  
class ForkJoinApp {  
 public static void main(String args[]) {  
 ForkJoinPool fjp = new ForkJoinPool();  
 double[] nums = new double[100000];  
 for (int i = 0; i < nums.length; i++) {  
 nums[i] = (double) (((i % 2) == 0) ? -i : i);  
 }  
 Sum task = new Sum(nums, 0, nums.length);  
 double summation = fjp.invoke(task);  
 System.*out*.println("Підсумовування: " + summation);  
 }  
}

Додаток «Г»

import java.util.concurrent.RecursiveTask;  
  
/\*  
Створіть додаток, що буде виконувати множення матриці  
на коефіцієнт та виконувати подальше підсумовування елементів  
матриці за допомогою засобів каркасу Fork/Join Framework.  
\*/  
class MatrixTask extends RecursiveTask<Integer> {  
 int THRESHOLD = 10;  
 int[][] matrix;  
 int colStart;  
 int colEnd;  
 int rowStart;  
 int rowEnd;  
 int scale;  
  
 MatrixTask(int[][] matrix, int colStart, int colEnd, int rowStart, int rowEnd, int scale) {  
 this.matrix = matrix;  
 this.colStart = colStart;  
 this.colEnd = colEnd;  
 this.rowStart = rowStart;  
 this.rowEnd = rowEnd;  
 this.scale = scale;  
 }  
  
 @Override  
 protected Integer compute() {  
 if ((rowEnd - rowStart) \* (colEnd - colStart) <= THRESHOLD)  
 return sumMatrix(matrix, rowStart, rowEnd, colStart, colEnd);  
  
 int midRow = (rowStart + rowEnd) / 2;  
 int midCol = (colStart + colEnd) / 2;  
  
 MatrixTask topLeft = new MatrixTask(matrix, rowStart, midRow, colStart, midCol, scale);  
 MatrixTask topRight = new MatrixTask(matrix, rowStart, midRow, midCol, colEnd, scale);  
 MatrixTask bottomLeft = new MatrixTask(matrix, midRow, rowEnd, colStart, midCol, scale);  
 MatrixTask bottomRight = new MatrixTask(matrix, midRow, rowEnd, midCol, colEnd, scale);  
  
 topLeft.fork();  
 topRight.fork();  
 bottomLeft.fork();  
  
 int sumBR = bottomRight.compute();  
 int sumBL = bottomLeft.join();  
 int sumTR = topRight.join();  
 int sumTL = topLeft.join();  
  
 return sumBR + sumBL + sumTL + sumTR;  
 }  
  
 private Integer sumMatrix(int[][] matrix, int rowStart, int rowEnd, int colStart, int colEnd) {  
 Integer sum = 0;  
 for (int i = rowStart; i < rowEnd; i++) {  
 for (int j = colStart; j < colEnd; j++) {  
 sum += matrix[i][j] \* scale;  
 }  
 }  
 return sum;  
 }  
}

import java.util.concurrent.ForkJoinPool;  
/\*  
Створіть додаток, що буде виконувати множення матриці  
на коефіцієнт тавиконувати подальше підсумовування елементів  
матриці за допомогою засобів каркасу Fork/Join Framework.  
\*/  
  
public class Main {  
 public static void main(String[] args) {  
 int size = 1000;  
 int scale = 2;  
 int[][] matrix = new int[size][size];  
 *writeToMatrix*(matrix);  
  
 //soutMatrix(matrix);  
  
 ForkJoinPool pool = new ForkJoinPool();  
 MatrixTask matrixTaskObject = new MatrixTask(matrix, 0, size, 0, size, scale);  
  
 int result = pool.invoke(matrixTaskObject);  
 System.*out*.println("Result: " + result);  
 //System.out.println(.getS());  
 }  
 private static int[][] writeToMatrix(int[][] matrix){  
 for (int i = 0; i < matrix.length; i++) {  
 for (int j = 0; j < matrix[i].length; j++) {  
 matrix[i][j] = i + j;  
 }  
 }  
 return matrix;  
 }  
 private static void soutMatrix(int[][] matrix){  
 for (int[] i : matrix) {  
 for (int j : i)  
 System.*out*.printf("%3d ", j);  
 System.*out*.println();  
 }  
 }  
}

Додаток «Д»

/\*Написати програму, що буде виконувати підсумовування елементів числового масиву за алгоритмом модифікованого каскадного підсумовування з  
використанням Fork/Join Framework. Розмір вхідного масиву має бути не  
меншим, ніж 10 000 елементів.  
\*/  
  
import java.util.concurrent.RecursiveTask;  
import java.util.concurrent.ForkJoinPool;  
  
/\*Написати програму, що буде виконувати підсумовування елементів числового масиву  
за алгоритмом модифікованого каскадного підсумовування з  
використанням Fork/Join Framework. Розмір вхідного масиву має бути не  
меншим, ніж 10 000 елементів.\*/  
  
public class Main {  
 public static void main(String[] args) {  
 int size = 20;  
 int[] array = new int[size];  
 *getMassive*(array);  
  
 ForkJoinPool pool = new ForkJoinPool();  
 int result = pool.invoke(new SumTask(array, 0, array.length));  
  
 System.*out*.println("Sum: " + result);  
 }  
 private static int[] getMassive(int[] arr){  
 for (int i = 0; i < arr.length; i++)  
 arr[i] = i;  
  
 return arr;  
 }  
  
}

import java.util.concurrent.RecursiveTask;  
  
public class SumTask extends RecursiveTask<Integer> {  
 int[] array;  
 int start;  
 int end;  
  
 public SumTask(int[] array, int start, int end) {  
 this.array = array;  
 this.start = start;  
 this.end = end;  
 }  
  
 @Override  
 protected Integer compute() {  
 int THRESHOLD = getTHRESHOLD();  
 if (start - end <= THRESHOLD)  
 return getSum();  
  
 int center = (start + end) / 2;  
  
 SumTask leftArray = new SumTask(array, start, center);  
 SumTask rightArray = new SumTask(array, center, end);  
  
 leftArray.fork();  
 int rightSum = rightArray.join();  
 int leftSum = leftArray.compute();  
  
 return (Integer) rightSum + leftSum;  
 }  
  
 private int getSum() {  
 int sum = 0;  
 for (int i = start; i < end; i++)  
 sum += array[i];  
 return sum;  
 }  
  
 private int getTHRESHOLD() {  
 return (int) Math.*ceil*(Math.*log10*(array.length) / Math.*log10*(2));  
 }  
}

Додаток «Е»

import java.util.concurrent.RecursiveTask;  
import java.util.concurrent.ForkJoinPool;  
  
public class Main {  
 static int *number* = 0;  
 public static void main(String[] args) {  
 int size = 1000;  
 int[][] matrix1 = new int[size][size];  
 int[][] matrix2 = new int[size][size];  
 *writeToMatrix*(matrix1);  
 *writeToMatrix*(matrix2);  
  
 System.*out*.println("Matrix 1");  
 //soutMatrix(matrix1);  
  
 System.*out*.println("Matrix 2");  
 //soutMatrix(matrix2);  
  
 ForkJoinPool pool = new ForkJoinPool();  
 MatrixMultiplicationTask task = new MatrixMultiplicationTask(matrix1, matrix2, 0, size, 0, size);  
  
 int[][] result = pool.invoke(task);  
  
 System.*out*.println("Matrix result:");  
 //soutMatrix(result);  
 }  
 private static int[][] writeToMatrix(int[][] matrix){  
 for (int i = 0; i < matrix.length; i++) {  
 for (int j = 0; j < matrix[i].length; j++) {  
 matrix[i][j] = *number*++;  
 //matrix[i][j] = i + j;  
 }  
 }  
 return matrix;  
 }  
 private static void soutMatrix(int[][] matrix){  
 for (int i = 0; i < matrix.length; i++) {  
 for (int j = 0; j < matrix[i].length; j++)  
 System.*out*.printf("%2d ", matrix[i][j]);  
 System.*out*.println();  
 }  
 }  
}  
/\*  
Matrix 1 Matrix 2  
0 1 2 0 1 2  
1 2 3 1 2 3  
2 3 4 2 3 4  
  
(0\*0 + 1\*1 + 2\*2) (0\*1 + 1\*2 + 2\*3) (0\*2 + 1\*3 + 2\*4)  
(1\*0 + 2\*1 + 3\*2) (1\*1 + 2\*2 + 3\*3) (1\*2 + 2\*3 + 3\*4)  
(2\*0 + 3\*1 + 4\*2) (2\*1 + 3\*2 + 4\*3) (2\*2 + 3\*3 + 4\*4)  
  
5 8 11  
8 14 20  
11 20 29  
  
  
// разные  
Matrix 1 Matrix 2  
0 1 2 9 10 11  
3 4 5 12 13 14  
6 7 8 15 16 17  
  
  
 9 12 15  
 10 13 16  
 11 14 17  
  
(0\*9 + 1\*12 + 2\*15) (0\*10 + 1\*13 + 2\*16) (0\*11 + 1\*14 + 2\*17)  
(3\*9 + 4\*12 + 5\*15) (3\*10 + 4\*13 + 5\*16) (3\*11 + 4\*14 + 5\*17)  
(6\*9 + 7\*12 + 8\*15) (6\*10 + 7\*13 + 8\*16) (6\*11 + 7\*14 + 8\*17)  
  
(0 + 12 + 30) (0 + 13 + 32) (0 + 14 + 34)  
(27 + 48 + 75) (30 + 52 + 80) (33 + 56 + 85)  
(54 + 84 + 120) (60 + 91 + 128) (66 + 98 + 136)  
  
32 41 50  
122 158 194  
212 275 338  
\* \*/

import java.util.concurrent.RecursiveTask;  
  
class MatrixMultiplicationTask extends RecursiveTask<int[][]> {  
 private static final int *THRESHOLD* = 1000; // Порогове значення для розміру підматриць  
 private final int[][] matrix1;  
 private final int[][] matrix2;  
 private final int rowStart;  
 private final int rowEnd;  
 private final int colStart;  
 private final int colEnd;  
  
 public MatrixMultiplicationTask(int[][] matrix1, int[][] matrix2, int rowStart, int rowEnd, int colStart, int colEnd) {  
 this.matrix1 = matrix1;  
 this.matrix2 = matrix2;  
 this.rowStart = rowStart;  
 this.rowEnd = rowEnd;  
 this.colStart = colStart;  
 this.colEnd = colEnd;  
 }  
  
 @Override  
 protected int[][] compute() {  
 if ((rowEnd - rowStart) \* (colEnd - colStart) <= *THRESHOLD*) {  
 return multiply(matrix1, matrix2, rowStart, rowEnd, colStart, colEnd);  
 } else {  
 int midRow = (rowStart + rowEnd) / 2;  
 int midCol = (colStart + colEnd) / 2;  
  
 MatrixMultiplicationTask topLeft = new MatrixMultiplicationTask(matrix1, matrix2, rowStart, midRow, colStart, midCol);  
 MatrixMultiplicationTask topRight = new MatrixMultiplicationTask(matrix1, matrix2, rowStart, midRow, midCol, colEnd);  
 MatrixMultiplicationTask bottomLeft = new MatrixMultiplicationTask(matrix1, matrix2, midRow, rowEnd, colStart, midCol);  
 MatrixMultiplicationTask bottomRight = new MatrixMultiplicationTask(matrix1, matrix2, midRow, rowEnd, midCol, colEnd);  
  
 topLeft.fork();  
 topRight.fork();  
 bottomLeft.fork();  
 int[][] bottomRightResult = bottomRight.compute();  
 int[][] topLeftResult = topLeft.join();  
 int[][] topRightResult = topRight.join();  
 int[][] bottomLeftResult = bottomLeft.join();  
  
 return joinResults(topLeftResult, topRightResult, bottomLeftResult, bottomRightResult);  
 }  
 }  
  
 private int[][] multiply(int[][] matrix1, int[][] matrix2, int rowStart, int rowEnd, int colStart, int colEnd) {  
 int[][] result = new int[rowEnd - rowStart][colEnd - colStart];  
 int colsMatrix2 = matrix2[0].length; // Кількість стовпців у транспонованій матриці 2  
 for (int i = rowStart; i < rowEnd; i++) {  
 for (int j = colStart; j < colEnd; j++) {  
 int sum = 0;  
 for (int k = 0; k < colsMatrix2; k++) {  
 sum += matrix1[i][k] \* matrix2[j][k]; // Множимо елементи по відповідним індексам  
 }  
 result[i - rowStart][j - colStart] = sum;  
 }  
 }  
 return result;  
 }  
  
  
 private int[][] joinResults(int[][] topLeft, int[][] topRight, int[][] bottomLeft, int[][] bottomRight) {  
 int[][] result = new int[topLeft.length + bottomLeft.length][topLeft[0].length + topRight[0].length];  
 for (int i = 0; i < topLeft.length; i++) {  
 System.*arraycopy*(topLeft[i], 0, result[i], 0, topLeft[i].length);  
 System.*arraycopy*(topRight[i], 0, result[i], topLeft[i].length, topRight[i].length);  
 }  
 for (int i = 0; i < bottomLeft.length; i++) {  
 System.*arraycopy*(bottomLeft[i], 0, result[i + topLeft.length], 0, bottomLeft[i].length);  
 System.*arraycopy*(bottomRight[i], 0, result[i + topLeft.length], bottomLeft[i].length, bottomRight[i].length);  
 }  
 return result;  
 }  
}