**Міністерство освіти і науки України**

**Національний технічний університет України**

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

**Факультет інформатики та обчислювальної техніки**

**Кафедра інформатики та програмної інженерії**

**Звіт**

Комп‘ютерного практикуму № 2 з дисципліни

«Технології паралельних та розподілених обчислень»

**«Розробка паралельних алгоритмів множення матриць та дослідження їх ефективності»**

**Виконав(ла)**

(шифр, прізвище, ім'я, по батькові)

*ІП-01 Корнієнко В.С.*

**Перевірив(ла)**

(прізвище, ім'я, по батькові)

*Стеценко І. В.*

Київ 2023

**Завдання:**

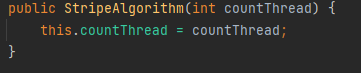
1. Реалізуйте стрічковий алгоритм множення матриць. Результат множення записуйте в об’єкт класу Result. **30 балів.**
2. Реалізуйте алгоритм Фокса множення матриць. **30 балів.**
3. Виконайте експерименти, варіюючи розмірність матриць, які перемножуються, для обох алгоритмів, та реєструючи час виконання алгоритму. Порівняйте результати дослідження ефективності обох алгоритмів. **20 балів.**
4. Виконайте експерименти, варіюючи кількість потоків, що використовується для паралельного множення матриць, та реєструючи час виконання. Порівняйте результати дослідження ефективності обох алгоритмів. **20 балів.**

**Хід роботи:**

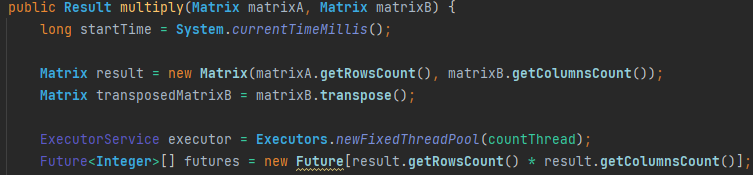
**Стрічковий алгоритм:**

В даній роботі стрічковий алгоритм був реалізований наступним чином:

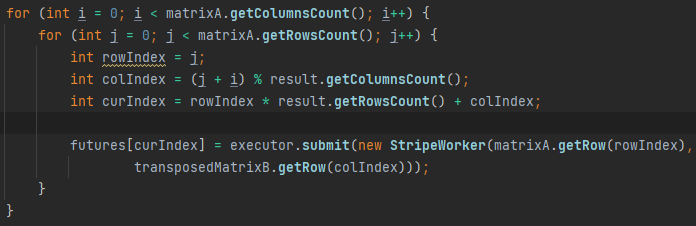
На початку ми ініціалізуємо клас з параметром кількості потоків, далі ця величина буде використовуватися під час створення ThreadPool-у

****

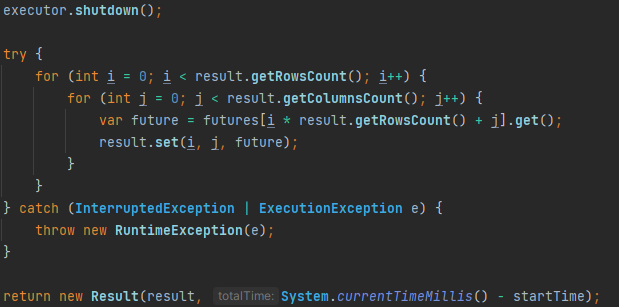
Створюємо метод multiply, який і буде множити матриці. В ньому в першу чергу ініціалізуємо результуючу матрицю, створюємо FixedThreadPool та масив Future<Integer>[] для зберігання результатів роботи пулу потоків

****

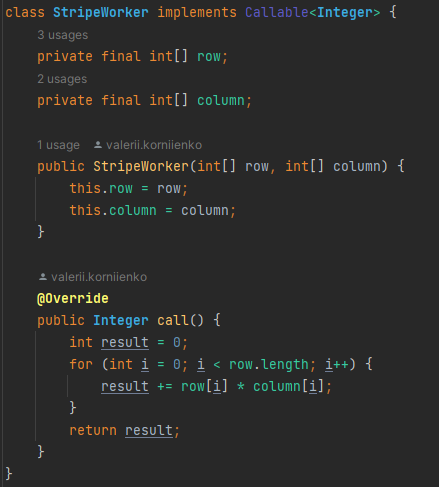
На наступному кроці ми додаємо задачі в пул потоків, а саме кожному потоку даємо 1 рядок з першої матриці та 1 колонку з 2 матриці. Для збільшення швидкодії операції діставання колонки, матриця 2 спочатку була транспонована.



Далі ми завершуємо роботу пулу потоків та дістаємо результати з масиву futures, попередньо дочекавшись виконання потоків завдяки методу get()



Для кожної дії submit ми створюємо новий StripeWorker, клас що безпосередньо перемножує рядок матриці на її стовпчик.

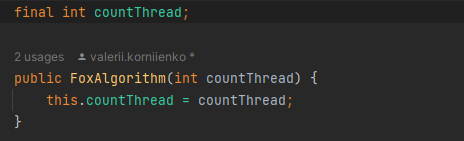


Можемо також побачити, що StripeWorker implements Callable<Integer>, це і дає можливість додавати результати в масив типу Future<Integer>[]

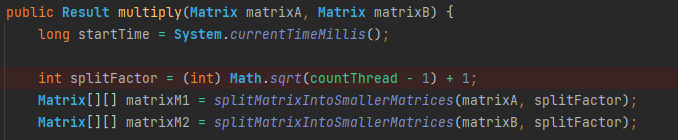
**Алгоритм Фокса**

Тепер переглянемо принцип роботи та реалізації алгоритму Фокса.

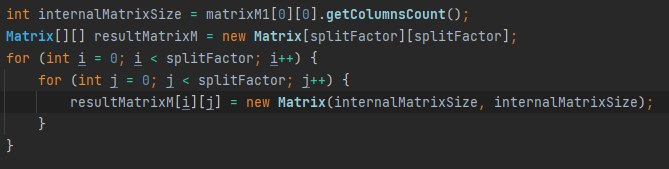
На початку ми ініціалізуємо клас з використанням числа кількості потоків аналогічно стрічковому алгоритму.



Створюємо метод multiply, який і буде множити матриці. В ньому в першу чергу ділимо початкові матриці на менші матриці (двовимірний масив матриць), при чому кількість частин на які ділиться матриця визначається змінною splitFactor

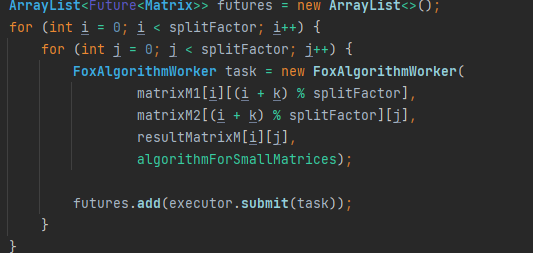


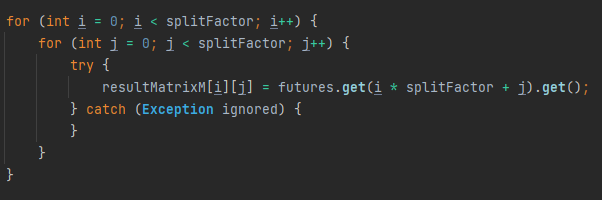
Далі ми ініціалізуємо результуючі матриці та створюємо пул потоків з заданою кількістю потоків



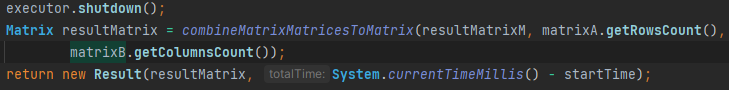


На наступному кроці ми в циклі, спочатку додаємо в пул потоків задачі множення менших матриць, а потім дістаємо результати роботи потоків, оновлюємо значення в матриці resultMatrixM

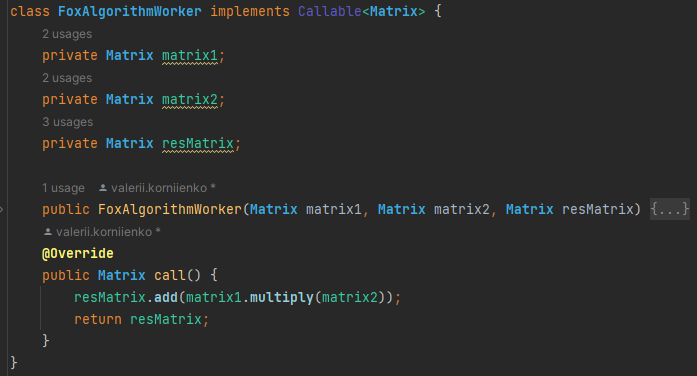
****

****

В кінці роботи програми ми завершуємо роботу пулу потоків та повертаємо результуючу матрицю. Остання фактично є матрицею, скомбінованою з менших частин, які ми рахували в ході роботи програми.

****

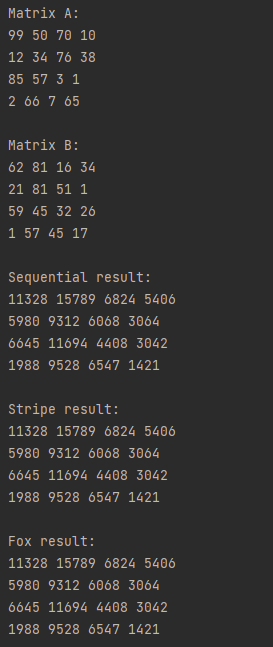
Клас FoxAlgorithmWorker виглядає наступним чином:

****

**Тестування алгоритмів**

Перевірка правильності роботи алгоритмів:

Для перевірки правильності роботи алгоритмів запустимо на виконання стрічковий, послідовний алгоритми та алгоритм Фокса та порівняємо результати їх роботи



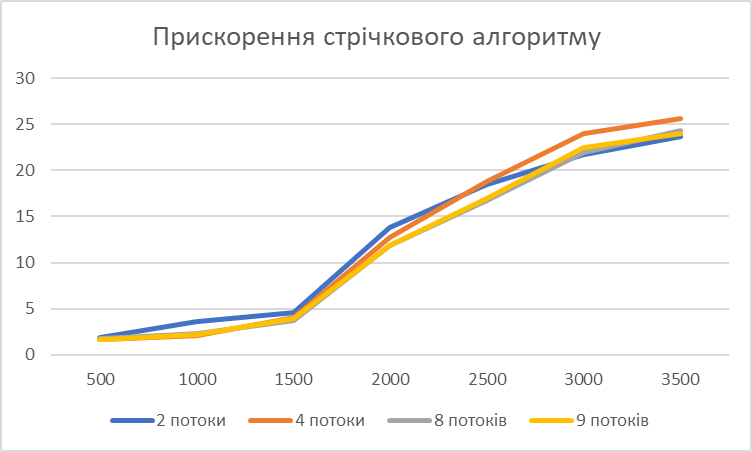
Можемо побачити, що всі 3 алгоритми повернули правильний результат множення матриці «А» на матрицю «В»

**Дослідження ефективності алгоритмів**

Для дослідження ефективності алгоритмів було проведено ряд тестів з різними розмірностями матриці та різною кількістю потоків. В результаті було отримано наступні результати:

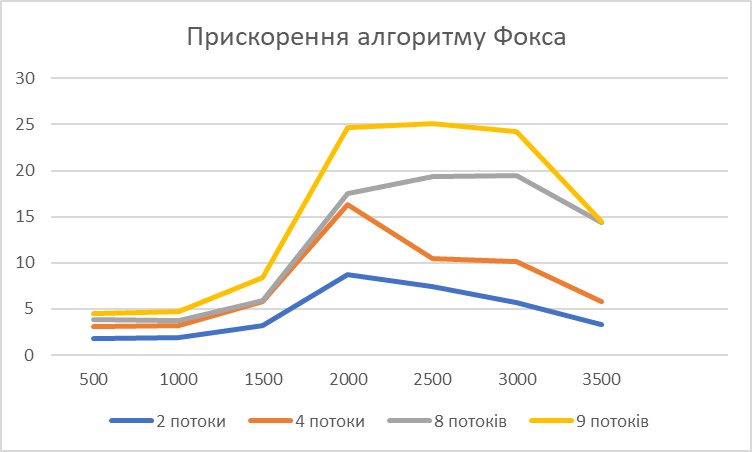
Для стрічкового алгоритму:





Бачимо що прискорення стрічкового алгоритму збільшується з розмірністю матриці, проте кількість потоків фактично не впливає на результат. Єдиною умовою є кількість потоків > 2

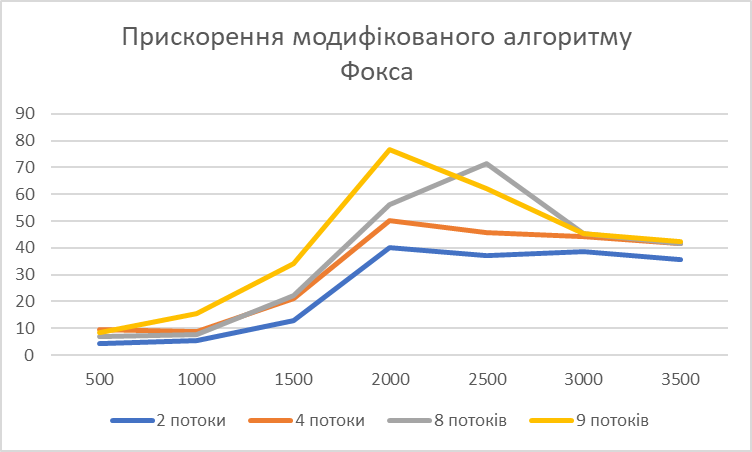




З графіку алгоритму Фокса бачимо, що він має суттєву перевагу при розмірності матриці 1500-3000 над стрічковим алгоритмом. Також можемо побачити, що прискорення алгоритму Фокса залежить від кількості потоків.

В ході виконання лабораторної роботи також була досліджена можливість модифікацій алгоритму Фокса. Для цього можна замінити стандартний послідовний алгоритм на більш швидкий послідовний. В процесі тестування було виявлено наступні результати:





Бачимо прискорення алгоритму в 80 раз при розмірності матриць 2000 та спад при 3000, аналогічний стандартному алгоритму Фокса

**Висновок**

Загальний висновок з проведеного дослідження ефективності алгоритмів для розпаралелювання операції множення матриць полягає в наступному:

Перш за все, результати показують, що стрічковий алгоритм демонструє збільшення прискорення з ростом розмірності матриці, але кількість потоків майже не впливає на його ефективність.

З іншого боку, алгоритм Фокса показує помітну перевагу над стрічковим алгоритмом при розмірності матриці в діапазоні 1500-3000. Крім того, прискорення алгоритму Фокса залежить від кількості потоків, що використовуються.

Дослідження також включало модифікацію алгоритму Фокса, яка полягала у заміні стандартного послідовного алгоритму на більш швидкий послідовний. Результати показали значне прискорення алгоритму на 80 разів для матриць розмірністю 2000, але згасання цього прискорення при розмірності 3000, подібне до стандартного алгоритму Фокса.

Крім того, виявлено, що зміна кількості процесів має незначний вплив на ефективність стрічкового алгоритму через велику кількість задач з обмеженим обчислювальним завданням. З іншого боку, алгоритм Фокса має меншу кількість задач, але вони вимагають більшого обчислювального ресурсу.

Отже, в результаті дослідження було показано, що ефективність різних алгоритмів розпаралелювання матричних операцій залежить від розмірності матриці та кількості потоків. Вибір оптимального алгоритму залежить від конкретних умов та вимог задачі. Наприклад, для великих матриць або при невеликій кількості доступних потоків стрічковий алгоритм може бути ефективнішим, при розмірності матриць 1500-3000 алгоритм Фокса буде кращим варіантом. Також, при розмірностях до 1500, алгоритм Фокса показує себе краще ніж стрічковий.

**Лістинг коду програми:**

**Main.java**

public class **Main** {  
 public static void main(**String**[] args) {  
// testAlgorithmsAccuracy();  
  
 int[] matrixSizes = {500, 1000, 1500, 2000, 2500, 3000, 3500};  
 int[] threadsCounts = {2, 4, 8, 9};  
  
 *testAlgorithmsSpeed*(matrixSizes, threadsCounts);  
 }  
  
 private static void testAlgorithmsSpeed(int[] matrixSizes, int[] threadsCounts) {  
 for (int matrixSize : matrixSizes) {  
 **Matrix** matrixA = **MatrixHelper**.*generateRandomMatrix*(matrixSize);  
 **Matrix** matrixB = **MatrixHelper**.*generateRandomMatrix*(matrixSize);  
 **System**.*out*.**println**(**"-------------------------"**);  
 **System**.*out*.**println**(**"Matrix size: "** + matrixSize);  
  
 long sequentialTime = *checkAlgorithmSpeed*(matrixA, matrixB, new **SequentialAlgorithm**(), 5);  
 **System**.*out*.**println**(**"\nSequential algorithm: "** + sequentialTime + **" ms"**);  
  
 for (int threads : threadsCounts) {  
 **System**.*out*.**println**(**"\nThreads count: "** + threads);  
 long stripeTime = *checkAlgorithmSpeed*(matrixA, matrixB, new **StripeAlgorithm**(threads), 5);  
 long foxTime = *checkAlgorithmSpeed*(matrixA, matrixB, new **FoxAlgorithm**(threads), 5);  
  
 **System**.*out*.**println**(**"\tStripe algorithm with "** + threads + **" threads: "** + stripeTime + **" ms"**);  
 **System**.*out*.**println**(**"\tFox algorithm with "** + threads + **" threads: "** + foxTime + **" ms"**);  
 }  
 }  
 }  
  
 static void testAlgorithmsAccuracy() {  
 **Matrix** matrixA = **MatrixHelper**.*generateRandomMatrix*(4);  
 **Matrix** matrixB = **MatrixHelper**.*generateRandomMatrix*(4);  
  
 **Matrix** resultSequential = new **SequentialAlgorithm**().**multiply**(matrixA, matrixB).**getResultMatrix**();  
 **Matrix** resultStripe = new **StripeAlgorithm**(2).**multiply**(matrixA, matrixB).**getResultMatrix**();  
 **Matrix** resultFox = new **FoxAlgorithm**(2).**multiply**(matrixA, matrixB).**getResultMatrix**();  
  
 **System**.*out*.**println**(**"Matrix A:"**);  
 matrixA.**print**();  
  
 **System**.*out*.**println**(**"\nMatrix B:"**);  
 matrixB.**print**();  
  
 **System**.*out*.**println**(**"\nSequential result:"**);  
 resultSequential.**print**();  
  
 **System**.*out*.**println**(**"\nStripe result:"**);  
 resultStripe.**print**();  
  
 **System**.*out*.**println**(**"\nFox result:"**);  
 resultFox.**print**();  
 }  
  
 static long checkAlgorithmSpeed(**Matrix** matrixA, **Matrix** matrixB, IMatrixMultiplicationAlgorithm multiplicationAlgorithm, int iterations) {  
 long sum = 0;  
 for (int i = 0; i < iterations; i++) {  
 sum += multiplicationAlgorithm.**multiply**(matrixA, matrixB).**getTotalTime**();  
 }  
 return sum / iterations;  
 }  
  
}

**Result.java**

public class **Result** {  
 private final long totalTime;  
 private final **Matrix** resultMatrix;  
  
 public Result(**Matrix** resultMatrix, long totalTime) {  
 this.totalTime = totalTime;  
 this.resultMatrix = resultMatrix;  
 }  
  
 public long getTotalTime() {  
 return totalTime;  
 }  
  
 public **Matrix** getResultMatrix() {  
 return resultMatrix;  
 }  
}

**SequentialAlgorithm.java**

public class **SequentialAlgorithm** implements IMatrixMultiplicationAlgorithm {  
 **@Override** public **Result** multiply(**Matrix** matrixA, **Matrix** matrixB) {  
 long startTime = **System**.*currentTimeMillis*();  
  
 int[][] result = new int[matrixA.**getRowsCount**()][matrixB.**getColumnsCount**()];  
 for (int i = 0; i < matrixA.**getRowsCount**(); i++) {  
 for (int j = 0; j < matrixB.**getColumnsCount**(); j++) {  
 int value = 0;  
 for (int k = 0; k < matrixA.**getColumnsCount**(); k++) {  
 value += matrixA.**get**(i,k) \* matrixB.**get**(k,j);  
 }  
 result[i][j] = value;  
 }  
 }  
  
 long duration = **System**.*currentTimeMillis*() - startTime;  
 return new **Result**(new **Matrix**(result), duration);  
 }  
}

**StripeAlgorithm.java**

import **java.util.concurrent.**\*;  
  
public class **StripeAlgorithm** implements IMatrixMultiplicationAlgorithm {  
 private final int countThread;  
  
 public StripeAlgorithm(int countThread) {  
 this.countThread = countThread;  
 }  
  
 public **Result** multiply(**Matrix** matrixA, **Matrix** matrixB) {  
 long startTime = **System**.*currentTimeMillis*();  
  
 **Matrix** result = new **Matrix**(matrixA.**getRowsCount**(), matrixB.**getColumnsCount**());  
 **Matrix** transposedMatrixB = matrixB.**transpose**();  
  
 ExecutorService executor = **Executors**.*newFixedThreadPool*(countThread);  
 Future<**Integer**>[] futures = new **Future**[result.**getRowsCount**() \* result.**getColumnsCount**()];  
  
 for (int i = 0; i < matrixA.**getColumnsCount**(); i++) {  
 for (int j = 0; j < matrixA.**getRowsCount**(); j++) {  
 int rowIndex = j;  
 int colIndex = (j + i) % result.**getColumnsCount**();  
 int curIndex = rowIndex \* result.**getRowsCount**() + colIndex;  
  
 futures[curIndex] = executor.**submit**(new **StripeWorker**(matrixA.**getRow**(rowIndex),  
 transposedMatrixB.**getRow**(colIndex)));  
 }  
 }  
  
 executor.**shutdown**();  
  
 try {  
 for (int i = 0; i < result.**getRowsCount**(); i++) {  
 for (int j = 0; j < result.**getColumnsCount**(); j++) {  
 var future = futures[i \* result.**getRowsCount**() + j].**get**();  
 result.**set**(i, j, future);  
 }  
 }  
 } catch (**InterruptedException** | **ExecutionException** e) {  
 throw new **RuntimeException**(e);  
 }  
  
 return new **Result**(result, **System**.*currentTimeMillis*() - startTime);  
 }  
}  
  
class **StripeWorker** implements Callable<**Integer**> {  
 private final int[] row;  
 private final int[] column;  
  
 public StripeWorker(int[] row, int[] column) {  
 this.row = row;  
 this.column = column;  
 }  
  
 **@Override** public **Integer** call() {  
 int result = 0;  
 for (int i = 0; i < row.length; i++) {  
 result += row[i] \* column[i];  
 }  
 return result;  
 }  
}

**FoxAlgorithm.java**

import **java.util.ArrayList**;  
import **java.util.concurrent.**Callable;  
import **java.util.concurrent.**ExecutorService;  
import **java.util.concurrent.Executors**;  
import **java.util.concurrent.**Future;  
  
public class **FoxAlgorithm** implements IMatrixMultiplicationAlgorithm {  
 final int countThread;  
  
 public FoxAlgorithm(int countThread) {  
 this.countThread = countThread;  
 }  
  
 public **Result** multiply(**Matrix** matrixA, **Matrix** matrixB) {  
 long startTime = **System**.*currentTimeMillis*();  
  
 int splitFactor = (int) **Math**.*sqrt*(countThread - 1) + 1;  
 **Matrix**[][] matrixM1 = *splitMatrixIntoSmallerMatrices*(matrixA, splitFactor);  
 **Matrix**[][] matrixM2 = *splitMatrixIntoSmallerMatrices*(matrixB, splitFactor);  
  
 int internalMatrixSize = matrixM1[0][0].**getColumnsCount**();  
 **Matrix**[][] resultMatrixM = new **Matrix**[splitFactor][splitFactor];  
 for (int i = 0; i < splitFactor; i++) {  
 for (int j = 0; j < splitFactor; j++) {  
 resultMatrixM[i][j] = new **Matrix**(internalMatrixSize, internalMatrixSize);  
 }  
 }  
  
 ExecutorService executor = **Executors**.*newFixedThreadPool*(countThread);  
 for (int k = 0; k < splitFactor; k++) {  
 **ArrayList**<Future<**Matrix**>> futures = new **ArrayList**<>();  
 for (int i = 0; i < splitFactor; i++) {  
 for (int j = 0; j < splitFactor; j++) {  
 **FoxAlgorithmWorker** task = new **FoxAlgorithmWorker**(  
 matrixM1[i][(i + k) % splitFactor],  
 matrixM2[(i + k) % splitFactor][j],  
 resultMatrixM[i][j]);  
  
 futures.**add**(executor.**submit**(task));  
 }  
 }  
  
 for (int i = 0; i < splitFactor; i++) {  
 for (int j = 0; j < splitFactor; j++) {  
 try {  
 resultMatrixM[i][j] = futures.**get**(i \* splitFactor + j).**get**();  
 } catch (**Exception** ignored) {  
 }  
 }  
 }  
 }  
 executor.**shutdown**();  
 **Matrix** resultMatrix = *combineMatrixMatricesToMatrix*(resultMatrixM, matrixA.**getRowsCount**(),  
 matrixB.**getColumnsCount**());  
 return new **Result**(resultMatrix, **System**.*currentTimeMillis*() - startTime);  
 }  
  
 public static **Matrix**[][] splitMatrixIntoSmallerMatrices(**Matrix** matrix, int splitFactor) {  
 **Matrix**[][] matrixMatrices = new **Matrix**[splitFactor][splitFactor];  
 int sizeInternal = (int) ((matrix.**getColumnsCount**() - 1) / splitFactor) + 1;  
  
 for (int i = 0; i < splitFactor; i++) {  
 for (int j = 0; j < splitFactor; j++) {  
 matrixMatrices[i][j] = new **Matrix**(sizeInternal, sizeInternal);  
  
 for (int k = 0; k < sizeInternal; k++) {  
 for (int l = 0; l < sizeInternal; l++) {  
 if (i \* sizeInternal + k >= matrix.**getRowsCount**()  
 || j \* sizeInternal + l >= matrix.**getColumnsCount**()) {  
 matrixMatrices[i][j].**set**(k, l, 0);  
 } else {  
 int element = matrix.**get**(i \* sizeInternal + k, j \* sizeInternal + l);  
 matrixMatrices[i][j].**set**(k, l, element);  
 }  
 }  
 }  
 }  
 }  
 return matrixMatrices;  
 }  
  
 public static **Matrix** combineMatrixMatricesToMatrix(**Matrix**[][] matrixM, int rowsCount, int columnsCount) {  
 **Matrix** resultMatrix = new **Matrix**(rowsCount, columnsCount);  
  
 for (int i = 0; i < matrixM.length; i++) {  
 for (int j = 0; j < matrixM[i].length; j++) {  
  
 for (int k = 0; k < matrixM[i][j].**getRowsCount**(); k++) {  
 for (int l = 0; l < matrixM[i][j].**getColumnsCount**(); l++) {  
  
 if (i \* matrixM[i][j].**getRowsCount**() + k < rowsCount && j \* matrixM[i][j].**getColumnsCount**() + l < columnsCount) {  
  
 resultMatrix.**set**(i \* matrixM[i][j].**getRowsCount**() + k, j \* matrixM[i][j].**getColumnsCount**() + l,  
 matrixM[i][j].**get**(k, l));  
  
 }  
 }  
 }  
 }  
 }  
  
 return resultMatrix;  
 }  
}  
  
class **FoxAlgorithmWorker** implements Callable<**Matrix**> {  
 private **Matrix** matrix1;  
 private **Matrix** matrix2;  
 private **Matrix** resMatrix;  
  
 public FoxAlgorithmWorker(**Matrix** matrix1, **Matrix** matrix2, **Matrix** resMatrix) {  
 this.matrix1 = matrix1;  
 this.matrix2 = matrix2;  
 this.resMatrix = resMatrix;  
 }  
 **@Override** public **Matrix** call() {  
 resMatrix.**add**(matrix1.**multiply**(matrix2));  
 return resMatrix;  
 }  
}

**Matrix.java**

import **java.nio.ByteBuffer**;  
import **java.nio.ByteOrder**;  
  
public class **Matrix** {  
 private int[][] matrixData;  
 public static final int *INT32\_BYTE\_SIZE* = 4;  
  
 public Matrix(int[][] matrix) {  
 this.matrixData = matrix;  
 }  
  
 public Matrix(int height, int width){  
 this.matrixData = new int[height][width];  
 }  
 public int[] getRow(int rowIndex){  
 return matrixData[rowIndex];  
 }  
 public int[][] getMatrix() {  
 return matrixData;  
 }  
  
 public int getRowsCount() {  
 return matrixData.length;  
 }  
  
 public int getColumnsCount() {  
 return matrixData[0].length;  
 }  
  
 public int get(int row, int column) {  
 return matrixData[row][column];  
 }  
  
 public void set(int row, int column, int value) {  
 matrixData[row][column] = value;  
 }  
  
 public void add(**Matrix** matrixB) {  
 for (int i = 0; i < matrixB.**getRowsCount**(); i++) {  
 for (int j = 0; j < matrixB.**getColumnsCount**(); j++) {  
 matrixData[i][j] += matrixB.**get**(i, j);  
 }  
 }  
 }  
  
 public boolean equals(**Matrix** matrix1){  
 if (matrixData.length != matrix1.**getRowsCount**() || matrixData[0].length != matrix1.**getColumnsCount**()) {  
 return false;  
 }  
 for (int i = 0; i < matrixData.length; i++) {  
 for (int j = 0; j < matrix1.**getColumnsCount**(); j++) {  
 if (matrixData[i][j] != matrix1.**get**(i, j)) {  
 return false;  
 }  
 }  
 }  
 return true;  
 }  
  
 public **Matrix** sliceMatrix(int startRowIndex, int endRowIndex, int columnsCount)  
 {  
 **Matrix** subMatrix = new **Matrix**(endRowIndex - startRowIndex + 1, columnsCount);  
 for (int i = startRowIndex; i <= endRowIndex; i++) {  
 for (int j = 0; j < columnsCount; j++) {  
 subMatrix.**set**(i - startRowIndex, j, matrixData[i][j]);  
 }  
 }  
 return subMatrix;  
 }  
  
 public void updateMatrixSlice(**Matrix** matrix, int indexStartRow, int indexEndRow, int countColumns)  
 {  
 for (int i = indexStartRow; i <= indexEndRow; i++) {  
 for (int j = 0; j < countColumns; j++) {  
 matrixData[i][j] = matrix.**get**(i - indexStartRow, j);  
 }  
 }  
 }  
  
 public int[] toIntBuffer()  
 {  
 int [] array = new int[**getRowsCount**() \* **getColumnsCount**()];  
 int index = 0;  
 for (int i = 0; i < **getRowsCount**(); i++) {  
 for (int j = 0; j < **getColumnsCount**(); j++) {  
 array[index] = matrixData[i][j];  
 index++;  
 }  
 }  
 return array;  
 }  
  
 public byte[] toByteBuffer() {  
 var buffer = **ByteBuffer**.*allocate*(**getRowsCount**() \* **getColumnsCount**() \* *INT32\_BYTE\_SIZE*);  
 buffer.**order**(**ByteOrder**.*nativeOrder*());  
 var intBuffer = buffer.**asIntBuffer**();  
 for (var ints : matrixData) {  
 intBuffer.**put**(ints);  
 }  
  
 return buffer.**array**();  
 }  
  
 public **Matrix** transpose() {  
 int[][] result = new int[matrixData[0].length][matrixData.length];  
 for (int i = 0; i < matrixData.length; i++) {  
 for (int j = 0; j < matrixData[0].length; j++) {  
 result[j][i] = matrixData[i][j];  
 }  
 }  
 return new **Matrix**(result);  
 }  
  
 public **Matrix** clone() {  
 int[][] result = new int[matrixData.length][matrixData[0].length];  
 for (int i = 0; i < matrixData.length; i++) {  
 **System**.*arraycopy*(matrixData[i], 0, result[i], 0, matrixData[0].length);  
 }  
 return new **Matrix**(result);  
 }  
  
 public void print() {  
 for (int i = 0; i < matrixData.length; i++) {  
 for (int j = 0; j < matrixData[0].length; j++) {  
 **System**.*out*.**print**(matrixData[i][j] + **" "**);  
 }  
 **System**.*out*.**println**();  
 }  
 }  
  
 public **Matrix** multiply(**Matrix** matrix2) {  
 int[][] result = new int[matrixData.length][matrix2.**getColumnsCount**()];  
 for (int i = 0; i < matrixData.length; i++) {  
 for (int j = 0; j < matrix2.**getColumnsCount**(); j++) {  
 for (int k = 0; k < matrixData[0].length; k++) {  
 result[i][j] += matrixData[i][k] \* matrix2.**get**(k, j);  
 }  
 }  
 }  
 return new **Matrix**(result);  
 }  
}

**MatrixHelper.java**

import **java.nio.ByteBuffer**;  
import **java.nio.ByteOrder**;  
  
public class **MatrixHelper** {  
 public static **Matrix** generateRandomMatrix(int width, int height, int minValue, int maxValue) {  
 int[][] result = new int[height][width];  
 for (int i = 0; i < height; i++) {  
 for (int j = 0; j < width; j++) {  
 result[i][j] = (int) (**Math**.*random*() \* (maxValue - minValue)) + minValue;  
 }  
 }  
 return new **Matrix**(result);  
 }  
  
 public static **Matrix** generateRandomMatrix(int size) {  
 return *generateRandomMatrix*(size, size, 0, 100);  
 }  
  
 public static **Matrix** createMatrixFromBuffer(int[] array, int rowsCount, int columnsCount) {  
 int[][] matrixData = new int[rowsCount][columnsCount];  
  
 int arrayIndex = 0;  
 for (int i = 0; i < rowsCount; i++) {  
 for (int j = 0; j < columnsCount; j++) {  
 matrixData[i][j] = array[arrayIndex];  
 arrayIndex++;  
 }  
 }  
 return new **Matrix**(matrixData);  
 }  
  
 public static **Matrix** createMatrixFromBuffer(byte[] bytes, int rows, int cols) {  
 var buffer = **ByteBuffer**.*wrap*(bytes);  
 buffer.**order**(**ByteOrder**.*nativeOrder*());  
 var array = new int[rows][cols];  
 for (var i = 0; i < rows; i++) {  
 for (var j = 0; j < cols; j++) {  
 array[i][j] = buffer.**getInt**();  
 }  
 }  
 return new **Matrix**(array);  
 }  
}

P.S. Класи Matrix, MatrixHelper мають додаткові(на даний момент зайві) методи, необхідні для виконання 6, 7 лабораторних робіт