Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

**Комп’ютерний практикум № 2**

з дисципліни «Технології паралельних обчислень»

**Тема:** «Розробка пaралельних алгоритмів множення матриць та дослідження їх ефективності»

|  |  |
| --- | --- |
| **Виконав:**  студент групи ІТ-04  Чапча Святослав  Дата здачі \_\_\_\_\_\_\_\_  Захищено з балом \_\_\_\_\_\_\_ | **Перевірила:**  ас. кафедри ІПІ  Дифучина О.Ю. |

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.**

**Виконання:**

**Лістинг коду:**

**Fox.java**

package Algorithms;  
  
import Models.Matrix;  
import Models.Result;  
import Threads.FoxThread;  
  
import java.util.ArrayList;  
import java.util.concurrent.Callable;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
  
import static Models.Matrix.*generateBlocks*;  
  
public class Fox implements IAlgorithm{  
 private Matrix matrix1;  
 private Matrix matrix2;  
 private int numOfThreads;  
 private int blockSize;  
  
 public Fox(Matrix matrix1, Matrix matrix2, int numOfThreads, int blockSize){  
 this.matrix1 = matrix1;  
 this.matrix2 = matrix2;  
 this.numOfThreads = numOfThreads;  
 this.blockSize = blockSize;  
 }  
 @Override  
 public Result multiply() {  
 if(matrix1.cols != matrix2.rows){  
 throw new IllegalArgumentException("Matrices cannot be multiplied - invalid dimensions");  
 }  
  
 long startTime = System.*currentTimeMillis*();  
 Matrix result = new Matrix(matrix1.rows, matrix2.cols);  
 Matrix[][] blocksMatrix1 = *generateBlocks*(matrix1, blockSize);  
 Matrix[][] blocksMatrix2 = *generateBlocks*(matrix2, blockSize);  
 Matrix[][] blocksResult = *generateBlocks*(result, blockSize);  
  
 ExecutorService executor = Executors.*newFixedThreadPool*(numOfThreads);  
 ArrayList<Callable<Object>> tasks = new ArrayList<>();  
 int numOfBlocks = matrix1.rows / blockSize;  
 for (int i = 0; i < numOfBlocks; i++) {  
 FoxThread thread = new FoxThread(blocksResult, blocksMatrix1, blocksMatrix2, i, blockSize);  
 tasks.add(Executors.*callable*(thread));  
 }  
 try {  
 executor.invokeAll(tasks);  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
  
 executor.shutdown();  
  
 for (int i = 0; i < numOfBlocks; i++) {  
 for (int j = 0; j < numOfBlocks; j++) {  
 double[][] blockMatrix = blocksResult[i][j].items;  
 for (int k = 0; k < blockSize; k++) {  
 for(int l = 0; l < blockSize; l++){  
 result.items[k + (i \* blockSize)][l + (j \* blockSize)] = blockMatrix[k][l];  
 }  
 }  
 }  
 }  
 long finishTime = System.*currentTimeMillis*();  
 return new Result(result, finishTime-startTime);  
 }  
}

**FoxThread.java**

package Threads;  
  
import Models.Matrix;  
import static Models.Matrix.*add*;  
import static Models.Matrix.*multiply*;  
  
public class FoxThread extends Thread{  
 private Matrix[][] blocksMatrix1;  
 private Matrix[][] blocksMatrix2;  
 private Matrix[][] result;  
 private int row;  
 private int blockSize;  
 public FoxThread(Matrix[][] result, Matrix[][] blocksMatrix1, Matrix[][] blocksMatrix2, int row, int blockSize) {  
 this.result = result;  
 this.blocksMatrix1 = blocksMatrix1;  
 this.blocksMatrix2 = blocksMatrix2;  
 this.row = row;  
 this.blockSize = blockSize;  
 }  
  
 @Override  
 public void run() {  
 for (int i = 0; i < blocksMatrix2[0].length; i++) {  
 Matrix resultBlock = new Matrix(new double[blockSize][blockSize]);  
 for (int j = 0; j < blocksMatrix1[0].length; j++) {  
 Matrix block1 = new Matrix(blocksMatrix1[row][j].items);  
 Matrix block2 = new Matrix(blocksMatrix2[j][i].items);  
  
 Matrix temp = *multiply*(block1, block2);  
 resultBlock = *add*(resultBlock, temp);  
 }  
 result[row][i].items = resultBlock.items;  
 }  
 }  
}

**Striped2.java**package Algorithms;  
  
import Models.Matrix;  
import Models.Result;  
import Threads.StripedCallable;  
  
import java.util.ArrayList;  
import java.util.concurrent.ExecutionException;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.concurrent.Future;  
  
public class Striped2 implements IAlgorithm{  
 private Matrix matrix1;  
 private Matrix matrix2;  
 private int numOfThreads;  
  
 public Striped2(Matrix matrix1, Matrix matrix2, int numOfThreads){  
 this.matrix1 = matrix1;  
 this.matrix2 = matrix2;  
 this.numOfThreads = numOfThreads;  
 }  
 @Override  
 public Result multiply(){  
 if(matrix1.cols != matrix2.rows){  
 throw new IllegalArgumentException("Matrices cannot be multiplied - invalid dimensions");  
 }  
  
 Matrix result = new Matrix(matrix1.rows, matrix2.cols);  
 Matrix transMatrix2 = matrix2.transpose();  
  
 long startTime = System.*currentTimeMillis*();  
 ExecutorService executor = Executors.*newFixedThreadPool*(numOfThreads);  
 ArrayList<StripedCallable> tasks = new ArrayList<>();  
 ArrayList<Future<Double>> futures = new ArrayList<>();  
  
 for (int i = 0; i < matrix1.rows; i++) {  
 for (int j = 0; j < matrix2.cols; j++) {  
 StripedCallable task = new StripedCallable(matrix1.getRow(i), transMatrix2.getRow(j));  
 tasks.add(task);  
 }  
 }  
 try {  
 futures.addAll(executor.invokeAll(tasks));  
 tasks.clear();  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 executor.shutdown();  
  
 try{  
 for (int i = 0; i < result.rows; i++) {  
 for (int j = 0; j < result.cols; j++) {  
 Future<Double> future = futures.get(i \* result.cols + j);  
 result.items[i][j] = future.get();  
 }  
 }  
 }  
 catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 } catch (ExecutionException e) {  
 throw new RuntimeException(e);  
 }  
  
 long finishTime = System.*currentTimeMillis*();  
 return new Result(result, finishTime - startTime);  
 }  
}

**StripedCallable.java**

package Threads;  
  
import java.util.concurrent.Callable;  
  
public class StripedCallable implements Callable<Double> {  
 private double[] row;  
 private double[] column;  
 public StripedCallable(double[] row, double[] col) {  
 this.row = row;  
 this.column = col;  
 }  
  
 @Override  
 public Double call() {  
 double result = 0;  
 for (int k = 0; k < row.length; k++) {  
 result += row[k] \* column[k];  
 }  
 return result;  
 }  
}

**Matrix.java**

package Models;  
  
import java.util.Arrays;  
  
public class Matrix {  
 public double[][] items;  
 public int rows;  
 public int cols;  
  
 public Matrix(int rows, int cols) {  
 this.items = new double[rows][cols];  
 this.rows = rows;  
 this.cols = cols;  
 }  
  
 public Matrix(double[][] items){  
 this.items = items;  
 this.rows = items.length;  
 this.cols = items[0].length;  
 }  
 public void print() {  
 for (int i = 0; i < this.rows; i++) {  
 for (int j = 0; j < this.cols; j++) {  
 System.*out*.printf("%10.1f", this.items[i][j]);  
 }  
 System.*out*.println();  
 }  
 }  
  
 public Matrix transpose() {  
 Matrix result = new Matrix(this.cols,this.rows);  
 for (int i = 0; i < this.rows; i++) {  
 for (int j = 0; j < this.cols; j++) {  
 result.items[j][i] = this.items[i][j];  
 }  
 }  
 return result;  
 }  
  
 public static Matrix add(Matrix matrix1, Matrix matrix2){  
 Matrix result = new Matrix(matrix1.rows, matrix1.rows);  
  
 for (int i = 0; i < result.rows; i++) {  
 for (int j = 0; j < result.cols; j++) {  
 result.items[i][j] = matrix1.items[i][j] + matrix2.items[i][j];  
 }  
 }  
 return result;  
 }  
  
 public static Matrix multiply(Matrix matrix1, Matrix matrix2){  
 Matrix result = new Matrix(matrix1.rows, matrix2.cols);  
  
 for (int i = 0; i < result.rows; i++) {  
 for (int j = 0; j < result.cols; j++) {  
 for (int k = 0; k < matrix1.cols; k++) {  
 result.items[i][j] += matrix1.items[i][k] \* matrix2.items[k][j];  
 }  
 }  
 }  
 return result;  
 }  
 public double[] getRow(int row) {  
 return this.items[row];  
 }  
  
 public double[] getColumn(int index){  
 double[] column = new double[this.cols];  
 for (int i = 0; i < this.rows; i++) {  
 column[i] = (items[i][index]);  
 }  
 return column;  
 }  
  
 public void generateRandomMatrix() {  
 for (int i = 0; i < this.rows; i++) {  
 for (int j = 0; j < this.cols; j++) {  
 this.items[i][j] = Math.*random*();  
 }  
 }  
 }  
  
 public static Matrix[][] generateBlocks(Matrix matrix, int blockSize){  
 int numOfBlocks = matrix.items.length / blockSize;  
 Matrix[][] blocks = new Matrix[numOfBlocks][numOfBlocks];  
 for(int i = 0; i < numOfBlocks; i++){  
 for(int j = 0; j < numOfBlocks; j++){  
 blocks[i][j] = new Matrix(blockSize,blockSize);  
 for(int k = 0; k < blockSize; k++){  
 for(int l = 0; l < blockSize; l++){  
 blocks[i][j].items[k][l] = matrix.items[i \* blockSize + k][l+(j\*blockSize)];  
 }  
 }  
 }  
 }  
 return blocks;  
 }  
}

**Main.java**

import Algorithms.Fox;  
import Algorithms.Standard;  
import Algorithms.Striped;  
import Algorithms.Striped2;  
import Models.Matrix;  
import Models.Result;  
  
public class Main {  
 public static void main(String[] args) {  
 int sizeOfMatrix = 10;  
 int numOfThreads = 8;  
 int blockSize = 2;  
  
 Matrix matrix1 = new Matrix(sizeOfMatrix,sizeOfMatrix);  
 matrix1.generateRandomMatrix();  
 System.*out*.println("Matrix1 Before:");  
 //matrix1.print();  
 System.*out*.println();  
  
 Matrix matrix2 = new Matrix(sizeOfMatrix,sizeOfMatrix);  
 matrix2.generateRandomMatrix();  
 System.*out*.println("Matrix2 Before:");  
 //matrix2.print();  
 System.*out*.println();  
  
 Standard standard = new Standard(matrix1, matrix2);  
 Result resultOfStandard = standard.multiply();  
 System.*out*.println("Matrix After Standard Multiply:");  
 //resultOfStandard.matrix.print();  
 System.*out*.println("Time:" + resultOfStandard.time);  
 System.*out*.println();  
  
 Striped striped = new Striped(matrix1, matrix2, numOfThreads);  
 Result resultOfStriped = striped.multiply();  
 System.*out*.println("Matrix After Striped Multiply:");  
 resultOfStriped.matrix.print();  
 System.*out*.println("Time:" + resultOfStriped.time);  
 System.*out*.println();  
  
 Striped2 striped2 = new Striped2(matrix1, matrix2, numOfThreads);  
 Result resultOfStriped2 = striped2.multiply();  
 System.*out*.println("Matrix After Striped Callable Multiply:");  
 resultOfStriped2.matrix.print();  
 System.*out*.println("Time:" + resultOfStriped2.time);  
 System.*out*.println();  
  
 Fox fox = new Fox(matrix1, matrix2, numOfThreads, blockSize);  
 Result resultOfFox = fox.multiply();  
 System.*out*.println("Matrix After Fox Multiply:");  
 resultOfFox.matrix.print();  
 System.*out*.println("Time:" + resultOfFox.time);  
 System.*out*.println();  
  
 }  
}

Повний код лабораторної роботи:  
<https://github.com/whitetark/multithreading/tree/main/lab02>

**Результати дослідження:**

Стрічковий алгоритм:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Size | Standard Algorithm | 2 Threads | | 4 Threads | | 8 Threads | |
| time | diff | time | diff | time | diff |
| 500 | 0.158 | 0.144 | 1.097 | 0.216 | 0.731 | 0.23 | 0.687 |
| 1000 | 1.361 | 0.759 | 1.793 | 1.017 | 1.338 | 0.962 | 1.415 |
| 1500 | 12.647 | 2.632 | 4.805 | 2.3 | 5.499 | 3.503 | 3.61 |
| 2000 | 44.947 | 5.52 | 8.143 | 6.221 | 7.225 | 5.372 | 8.367 |

Алгоритм фокса:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Size | Standard Algorithm | 2 Threads | | 4 Threads | | 8 Threads | |
| time | diff | time | diff | time | diff |
| 500 | 0.158 | 0.186 | 0.849 | 0.14 | 1.129 | 0.12 | 1.317 |
| 1000 | 1.361 | 1.301 | 1.046 | 0.825 | 1.65 | 0.688 | 1.978 |
| 1500 | 12.647 | 3.936 | 3.213 | 2.264 | 5.586 | 1.639 | 7.716 |
| 2000 | 44.947 | 8.638 | 5.203 | 5.150 | 8.728 | 3.434 | 13.089 |

**Висновок:** У ході лабораторної роботи були розроблені реалізації стрічкового та алгоритму Фокса для множення матриць. Після порівняння ефективності маємо такі результати.

Збільшуючи розмір матриці, бачимо, що обидва алгоритми виявляють більшу різницю у порівнянні зі звичайним алгоритмом. Час кожного алгоритма збільшується, але з ним і збільшується різниця. Тобто розмір матриці впливає на обидва алгоритми.

Збільшуючи кількість потоків, ми бачимо, що стрічковий алгоритм трохи погіршується, трохи покращується або відпрацьовує приблизно таку саму кількість часу, тобто стрічковий алгоритм не сильно залежить від кількості потоків. У свою чергу алгоритм Фокса навпаки працює більш продуктивно зі збільшенням кількості потоків.

Якщо ми порівняємо роботу обох алгоритмів, то можемо помітити, що стрічковий алгоритм завжди виявляє себе краще при 2-х потоках, але потім втрачає свою перевагу, оскільки алгоритм Фокса краще працює на 4+ потоках.