МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО» ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Кафедра інформатики та програмної інженерії

Звіт

З компʼютерного практикуму № 2 з дисципліни

«Технології паралельних обчислень»

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

**Виконав(ла)** *ІП-15 Мєшков Андрій*

(шифр, прізвище, ім'я, по батькові)

**Перевірив** *Дифучина О. Ю.*

(шифр, прізвище, ім'я, по батькові)

Київ 2024

# 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

Лістинг коду:

App.java

public class App {

public static void main(String[] args) {

Matrix matrix1 = *new* Matrix(1000);

Matrix matrix2 = *new* Matrix(1000);

test(matrix1, matrix2);

}

private static void test(Matrix matrix1, Matrix matrix2) {

int threadsCount = 2;

Multiplier multiplier = *new* Multiplier(matrix1, matrix2);

RowMultiplier rowMultiplier = *new* RowMultiplier(matrix1, matrix2, threadsCount);

FoxMultiplier foxMultiplier = *new* FoxMultiplier(

matrix1, matrix2,

MathUtils.roundToNearest(matrix1.size() / threadsCount),

threadsCount

);

long startTime = System.currentTimeMillis();

Matrix result = multiplier.optimiseMult();

long standartAlgoEnd = System.currentTimeMillis();

System.out.println(standartAlgoEnd - startTime);

Matrix rowResult = rowMultiplier.mult();

long rowAlgoEnd = System.currentTimeMillis();

System.out.println(rowAlgoEnd - standartAlgoEnd);

Matrix foxResult = foxMultiplier.mult();

long foxAlgoEnd = System.currentTimeMillis();

System.out.println(foxAlgoEnd - rowAlgoEnd);

*if* (result.isEqual(rowResult) && result.isEqual(foxResult)) {

System.out.println("Norm");

} *else* {

System.out.println("Not Norm");

}

}

}

Matrix.java

public class Matrix {

private int[][] matrix;

private int matrixSize;

static Matrix zeroMatrix(int size) {

int[][] matrix = *new* int[size][size];

*for* (int i = 0; i < size; i++) {

*for* (int j = 0; j < size; j++) {

matrix[i][j] = 0;

}

}

*return* *new* Matrix(matrix);

}

public Matrix(int[][] matrix) {

*this*.matrix = matrix;

*this*.matrixSize = matrix.length;

}

public Matrix(int size) {

*this*.matrixSize = size;

int[][] matrix = *new* int[size][size];

*for* (int i = 0; i < size; i++) {

*for* (int j = 0; j < size; j++) {

matrix[i][j] = 1;

}

}

*this*.matrix = matrix;

}

public int[] getRow(int index) {

*return* matrix[index];

}

public int[][] getMatrix() {

*return* matrix;

}

public int size() {

*return* matrixSize;

}

public void printMatrix() {

*for* (int i = 0; i < matrix.length; i++) {

*for* (int j = 0; j < matrix[i].length; j++) {

System.out.print(matrix[i][j] + " ");

}

System.out.println();

}

}

public Matrix transpose() {

int size = size();

int[][] transposedMatrix = *new* int[size][size];

*for* (int i = 0; i < size; i++) {

*for* (int j = 0; j < size; j++) {

transposedMatrix[j][i] = matrix[i][j];

}

}

*return* *new* Matrix(transposedMatrix);

}

void add(Matrix addedMatrix) {

int size = size();

*for* (int i = 0; i < size; i++) {

int[] addedRow = addedMatrix.getRow(i);

*for* (int j = 0; j < size; j++) {

matrix[i][j] += addedRow[j];

}

}

}

boolean isEqual(Matrix matrix) {

int size = size();

int[][] matrixValues = matrix.getMatrix();

*for* (int i = 0; i < size; i++) {

*for* (int j = 0; j < size; j++) {

*if* (matrixValues[i][j] != *this*.matrix[i][j]) {

*return* false;

}

}

}

*return* true;

}

}

ComplexMatrix.java

import java.util.ArrayList;

public class ComplexMatrix {

Matrix[][] matrix;

ComplexMatrix(Matrix matrix, int blockSize) {

*if* (matrix.size() % blockSize != 0) {

System.out.println(blockSize);

*this*.matrix = *new* Matrix[0][0];

} *else* {

int blocksCount = matrix.size() / blockSize;

*this*.matrix = *new* Matrix[blocksCount][blocksCount];

*for* (int blockRowIndex = 0; blockRowIndex < blocksCount; blockRowIndex++) {

ArrayList<int[][]> blocks = *new* ArrayList<int[][]>();

*for* (int blockColumn = 0; blockColumn < blocksCount; blockColumn++) {

blocks.add(*new* int[blockSize][blockSize]);

}

*for* (int realRowIndex = blockRowIndex \* blockSize; realRowIndex < (blockRowIndex + 1) \* blockSize; realRowIndex++) {

int[] realRow = matrix.getRow(realRowIndex);

*for* (int realColumn = 0; realColumn < matrix.size();realColumn++) {

int blockColumn = realColumn / blockSize;

int[][] block = blocks.get(blockColumn);

block[realRowIndex - blockSize \* blockRowIndex][realColumn - blockSize \* blockColumn] = realRow[realColumn];

blocks.set(blockColumn, block);

}

}

*for* (int blockColumn = 0; blockColumn < blocks.size(); blockColumn++) {

int[][] block = blocks.get(blockColumn);

*this*.matrix[blockRowIndex][blockColumn] = *new* Matrix(block);

}

}

}

}

ComplexMatrix(Matrix[][] matrix) {

*this*.matrix = matrix;

}

Matrix convertToMatrix() {

int blocksCount = matrix.length;

int blockSize = matrix[0][0].size();

int size = blockSize \* blocksCount;

int[][] result = *new* int[size][size];

*for* (int i = 0; i < size; i++) {

*for* (int j = 0; j < size; j++) {

result[i][j] = matrix[i / blockSize][j / blockSize].getRow(i % blockSize)[j % blockSize];

}

}

*return* *new* Matrix(result);

}

public int size() {

*return* matrix.length;

}

void printMatrix() {

*for* (int i = 0; i < matrix.length; i++) {

*for* (int j = 0; j < matrix[i].length; j++) {

matrix[i][j].printMatrix();

System.out.println();

}

System.out.println();

System.out.println();

}

}

Matrix[] getRow(int index) {

*return* matrix[index];

}

}

Multiplier.java

public class Multiplier {

Matrix matrix1;

Matrix matrix2;

public Multiplier(Matrix matrix1, Matrix matrix2) {

*this*.matrix1 = matrix1;

*this*.matrix2 = matrix2;

}

public Matrix mult() {

int[][] result = *new* int[matrix1.size()][matrix2.size()];

int[][] matrix1 = *this*.matrix1.getMatrix();

int[][] matrix2 = *this*.matrix2.getMatrix();

*for*(int i = 0; i < *this*.matrix1.size(); i++) {

*for*(int j = 0; j < *this*.matrix2.size(); j++) {

result[i][j] = 0;

*for*(int k = 0; k < *this*.matrix1.size(); k++) {

result[i][j] += matrix1[i][k] \* matrix2[k][j];

}

}

}

*return* *new* Matrix(result);

}

public Matrix optimiseMult() {

Matrix transpMatrix2 = matrix2.transpose();

int[][] result = *new* int[matrix1.size()][matrix2.size()];

*for*(int i = 0; i < matrix1.size(); i++) {

*for*(int j = 0; j < matrix2.size(); j++) {

int[] row = matrix1.getRow(i);

int[] column = transpMatrix2.getRow(j);

result[i][j] = 0;

*for*(int k = 0; k < matrix1.size(); k++) {

result[i][j] += row[k] \* column[k];

}

}

}

*return* *new* Matrix(result);

}

}

RowMultiplier.java

import java.util.ArrayList;

import java.util.concurrent.*\**;

public class RowMultiplier {

Matrix matrix1;

Matrix matrix2;

Matrix transpMatrix2;

int threadsCount;

int columnForTasks;

public RowMultiplier(Matrix matrix1, Matrix matrix2, int threadsCount) {

*this*.matrix1 = matrix1;

*this*.matrix2 = matrix2;

transpMatrix2 = matrix2.transpose(); *//*

*this*.threadsCount = threadsCount;

*this*.columnForTasks = MathUtils.roundToNearest(matrix1.size() / threadsCount);

}

public int[][] getNewRows(int threadIndex, int iteration) {

int[][] columns = *new* int[columnForTasks][matrix1.size()];

*for* (int column = 0; column < columnForTasks; column++) {

int rowIndex = (threadIndex + iteration + column) % matrix2.size();

columns[column] = transpMatrix2.getRow(rowIndex);

}

*return* columns;

}

public Matrix mult() {

*if* (matrix1.size() != matrix2.size()) {

*return* null;

}

int[][] result = *new* int[matrix1.size()][matrix2.size()];

ExecutorService executor = Executors.newFixedThreadPool(threadsCount);

ArrayList<RowMultiplyTask> tasks = *new* ArrayList<RowMultiplyTask>();

*// ArrayList<Future<int[]>> futures = new ArrayList<Future<int[]>>();*

ArrayList<Future<int[]>> futures = *new* ArrayList<Future<int[]>>(); *//*

*for*(int iteration = 0; iteration < matrix2.size(); iteration+=columnForTasks) {

*for*(int rowIndex = 0; rowIndex < matrix1.size(); rowIndex++) {

RowMultiplyTask thread = *new* RowMultiplyTask(

rowIndex,

iteration,

matrix1.getRow(rowIndex),

getNewRows(rowIndex, iteration)

);

tasks.add(thread);

}

*try*{

futures.addAll(executor.invokeAll(tasks));

tasks.clear();

*for* (int taskIndex = 0; taskIndex < futures.size(); taskIndex++) {

int row = taskIndex % matrix1.size();

int[] resultRow = futures.get(taskIndex).get();

*for* (int column = 0; column < resultRow.length; column++) {

result[row][(row + iteration + column) % matrix1.size()] = resultRow[column];

}

}

futures.clear();

} *catch* (InterruptedException e) {

*throw* *new* RuntimeException(e);

} *catch* (ExecutionException e) {

*throw* *new* RuntimeException(e);

}

}

executor.shutdown();

*try* {

executor.awaitTermination(Long.MAX\_VALUE, TimeUnit.NANOSECONDS);

} *catch* (InterruptedException e) {

e.printStackTrace();

}

*return* *new* Matrix(result);

}

}

RowMultiplyTask.java

import java.util.concurrent.Callable;

public class RowMultiplyTask implements Callable<int[]> { *// Callable<int[]> {*

int index;

int[] row;

int[][] columns;

int iteration;

public RowMultiplyTask(int index, int iteration, int[] row, int[][] columns) {

*this*.index = index;

*this*.row = row;

*this*.iteration = iteration;

*this*.columns = columns;

}

@Override

public int[] call(){

int[] result = *new* int[columns.length]; *//*

*for* (int column = 0; column < columns.length; column++) {

int[] row2 = columns[column];

*for*(int columnIndex = 0; columnIndex < row2.length; columnIndex++) {

result[column] += row[columnIndex] \* row2[columnIndex]; *//*

}

}

*return* result;

}

}

FoxMultiplier.java

import java.util.ArrayList;

import java.util.concurrent.*\**;

public class FoxMultiplier {

ComplexMatrix matrix1;

ComplexMatrix matrix2;

int blockSize;

int threadsCount;

public FoxMultiplier(Matrix matrix1, Matrix matrix2, int blockSize, int threadsCount) {

*this*.matrix1 = *new* ComplexMatrix(matrix1, blockSize);

*this*.matrix2 = *new* ComplexMatrix(matrix2, blockSize);

*this*.blockSize = blockSize;

*this*.threadsCount = threadsCount;

}

public Matrix mult() {

Matrix[][] result = *new* Matrix[matrix1.size()][matrix2.size()];

*for*(int i = 0; i < *this*.matrix1.size(); i++) {

*for* (int j = 0; j < *this*.matrix2.size(); j++) {

result[i][j] = Matrix.zeroMatrix(blockSize);

}

}

ExecutorService executor = Executors.newFixedThreadPool(threadsCount);

ArrayList<FoxMultiplyTask> tasks = *new* ArrayList<FoxMultiplyTask>();

ArrayList<Future<Matrix>> futures = *new* ArrayList<Future<Matrix>>();

*for*(int iteration = 0; iteration < matrix2.size(); iteration++) {

*for*(int rowIndex = 0; rowIndex < matrix1.size(); rowIndex++) {

Matrix[] row1 = matrix1.getRow(rowIndex);

Matrix[] row2 = matrix2.getRow((rowIndex + iteration) % matrix2.size());

int columnRow1Index = (iteration + rowIndex) % matrix2.size();

*for*(int columnIndex = 0; columnIndex < row2.length; columnIndex++) {

FoxMultiplyTask task = *new* FoxMultiplyTask(

row1[columnRow1Index],

row2[columnIndex],

rowIndex,

columnIndex

);

tasks.add(task);

}

}

*try*{

futures.addAll(executor.invokeAll(tasks));

*for* (int taskIndex = 0; taskIndex < futures.size(); taskIndex++) {

FoxMultiplyTask task = tasks.get(taskIndex);

Matrix resultElement = futures.get(taskIndex).get();

result[task.globalRow][task.globalColumn].add(resultElement);

}

tasks.clear();

futures.clear();

} *catch* (InterruptedException e) {

*throw* *new* RuntimeException(e);

} *catch* (ExecutionException e) {

*throw* *new* RuntimeException(e);

}

}

executor.shutdown();

*try* {

executor.awaitTermination(Long.MAX\_VALUE, TimeUnit.NANOSECONDS);

} *catch* (InterruptedException e) {

e.printStackTrace();

}

*return* *new* ComplexMatrix(result).convertToMatrix();

}

}

FoxMultiplyTask.java

import java.util.concurrent.Callable;

public class FoxMultiplyTask implements Callable<Matrix> {

Matrix matrix1;

Matrix matrix2;

int globalRow;

int globalColumn;

FoxMultiplyTask(Matrix matrix1, Matrix matrix2, int globalRow, int globalColumn) {

*this*.matrix1 = matrix1;

*this*.matrix2 = matrix2;

*this*.globalColumn = globalColumn;

*this*.globalRow = globalRow;

}

@Override

public Matrix call() {

Multiplier multiplier = *new* Multiplier(matrix1, matrix2);

*return* multiplier.optimiseMult();

}

}

MathUtils.java

public class MathUtils {

public static int roundToNearest(int number) {

int[] options = {125, 250, 500};

int closest = options[0];

*for* (int option *:* options) {

*if* (Math.abs(number - option) < Math.abs(number - closest)) {

closest = option;

}

}

*return* closest;

}

}

**Результат:**

|  |
| --- |
|  |
| Рисунок 1 – Результат запуску програми |

В таблиці зображено порівняння алгоритмів.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Розмір | Звичайний, мс | 2 | | | |
| Стрічковий, мс | Прискорення | Фокс, мс | Прискорення |
| 1000 | 956 | 457 | 2,09190372 | 303 | 3,155115512 |
| 1500 | 3375 | 1573 | 2,145581691 | 1009 | 3,344895937 |
| 2000 | 8566 | 3782 | 2,264939186 | 2291 | 3,738978612 |
| 2500 | 16838 | 6603 | 2,550053006 | 4369 | 3,85397116 |
| 3000 | 28085 | 10423 | 2,694521731 | 7266 | 3,865262868 |
| Розмір | Звичайний, мс | 4 | | | |
| Стрічковий, мс | Прискорення | Фокс, мс | Прискорення |
| 1000 | 956 | 405 | 2,360493827 | 298 | 3,208053691 |
| 1500 | 3375 | 1234 | 2,735008104 | 876 | 3,852739726 |
| 2000 | 8566 | 3020 | 2,836423841 | 2385 | 3,591614256 |
| 2500 | 16838 | 5608 | 3,002496434 | 4095 | 4,111843712 |
| 3000 | 28085 | 7961 | 3,527823138 | 6884 | 4,079750145 |
| Розмір | Звичайний, мс | 8 | | | |
| Стрічковий, мс | Прискорення | Фокс, мс | Прискорення |
| 1000 | 956 | 495 | 1,931313131 | 359 | 2,662952646 |
| 1500 | 3375 | 1562 | 2,160691421 | 951 | 3,548895899 |
| 2000 | 8566 | 3521 | 2,432831582 | 2091 | 4,096604495 |
| 2500 | 16838 | 6192 | 2,719315245 | 4195 | 4,013825983 |
| 3000 | 28085 | 10203 | 2,752621778 | 8723 | 3,219649203 |

# ВИСНОВКИ

В результаті роботи над комп’ютерним практикумом було розроблено програму, що реалізує паралельне множення матриць стрічковим алгоритмом та алгоритмом Фокса. В результаті дослідження виявлено, що розпаралелення пришвидшує множення матриць. Алгоритм Фокса показує кращі результати, особливо на великих розмірах матриць.