

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

Лабораторна робота №2

**Технології паралельних обчислень**

|  |  |  |
| --- | --- | --- |
| Виконав  студент групи ІТ-03: |  | Перевірив: |
| Чабан А.Є. | Дифучина О.Ю |
|  | | Дата: |
| Оцінка: |

Київ 2023

# Завдання: Text Description automatically generated

# Хід виконання:

[\*Посилання на GitHub репозиторій\*](https://github.com/antonchaban/MultiThreading/tree/master/src/main/java/lab2)

Завдання №1:

Для роботи з матрицями та зберігання результату множення матриць створимо клас MatrixEntity, за допомогою якого будемо працювати з матрицями, використовуючи створені методи:  
  
**package** lab2.Tools;

**import** **java.util.Arrays**;

/\*\*

\* Клас MatrixEntity використовується для роботи та зберігання матриць

\*/

**public** **class** **MatrixEntity** {

**private** **final** **int**[][] matrix;

/\*\*

\* Конструктор класу MatrixEntity, до конструктора передається матриця, яку необхідно зберегти

\*

\* @param matrix матриця у вигляді двомірного масиву, яку необхідно зберегти

\*/

**public** **MatrixEntity**(**int**[][] matrix) {

**this**.matrix = matrix;

}

/\*\*

\* Конструктор класу MatrixEntity, до конструктора передаються розміри матриці, яку необхідно зберегти

\*

\* @param rowsSize кількість рядків матриці

\* @param columnsSize кількість стовпців матриці

\*/

**public** **MatrixEntity**(**int** rowsSize, **int** columnsSize) {

matrix = **new** **int**[rowsSize][columnsSize];

}

**public** **int**[][] **getMatrix**() {

**return** matrix;

}

**public** **int** **getRowsSize**() {

**return** matrix.length;

}

**public** **int** **getColumnsSize**() {

**return** matrix[**0**].length;

}

/\*\*

\* Метод, який повертає значення елемента матриці за індексами

\*

\* @param i індекс рядка

\* @param j індекс стовпця

\* @return значення елемента матриці за індексами

\*/

**public** **int** **get**(**int** i, **int** j) {

**return** matrix[i][j];

}

/\*\*

\* Метод, який задає значення елемента матриці за індексами

\*

\* @param i індекс рядка

\* @param j індекс стовпця

\* @param value значення елемента матриці за індексами

\*/

**public** **void** **set**(**int** i, **int** j, **int** value) {

matrix[i][j] = value;

}

/\*\*

\* Метод, який виводить матрицю в консоль

\*/

**public** **void** **print2D**() {

Arrays.stream(matrix).map(**Arrays:**:toString).forEach(System.out::println);

}

}

Далі створимо клас ParallelCalculator в якому реалізуємо метод multiplyMatrix, який прийматиме 2 матриці та кількість потоків в яких буде працювати алгоритм, повертає сутність в якій записано результат

**package** lab2.Algorithms.Parallel;

**import** **lab2.Tools.MatrixEntity**;

**import** **java.util.ArrayList**;

/\*\*

\* Клас ParallelCalculator використовується для багатопоточного множення матриць

\* в середині класу було реалізовано стрічковий алгоритм множення матриць

\*/

**public** **class** **ParallelCalculator** {

/\*\*

\* Метод для множення матриць за допомогою стрічкового алгоритму

\* @param matrixEntity1 перша матриця

\* @param matrixEntity2 друга матриця

\* @param threadsCount кількість потоків

\* @return результат множення матриць у вигляді MatrixEntity

\*/

**public** MatrixEntity **multiplyMatrix**(MatrixEntity matrixEntity1, MatrixEntity matrixEntity2, **int** threadsCount) {

// Перевірка того чи можна множити матриці між собою (кількість стовпців першої матриці

// має бути рівною кількості рядків другої матриці)

**if** (matrixEntity1.getColumnsSize() != matrixEntity2.getRowsSize()) {

**throw** **new** **IllegalArgumentException**("matrices cannot be multiplied because the " +

"number of columns of matrix A is not equal to the number of rows of matrix B.");

}

var height = matrixEntity1.getRowsSize();

var width = matrixEntity2.getColumnsSize();

var resultMatrix = **new** MatrixEntity(height, width);

// Встановлюємо кількість рядків які будуть обровблятися в одному потоці

var rowsPerThread = height / threadsCount;

var threads = **new** ArrayList<Thread>();

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

var from = i \* rowsPerThread;

**int** to;

// Встановлюємо межі роботи для кожного потоку

**if** (i == threadsCount - **1**) {

to = height;

} **else** {

to = (i + **1**) \* rowsPerThread;

}

threads.add(**new** Thread(() -> { // Створюємо поток який буде рахувати результат у встановленому діапазоні

**for** (**int** row = from; row < to; row++) { // Ітеруємось по призначених рядках у межах встановленого діапазону

**for** (**int** col = **0**; col < width; col++) {

**for** (**int** k = **0**; k < matrixEntity2.getRowsSize(); k++) { // Множимо елементи матриць

resultMatrix.set(row, col, resultMatrix.get(row, col)

+ matrixEntity1.get(row, k) \* matrixEntity2.get(k, col));

}

}

}

}));

}

**for** (Thread thread : threads) {

thread.start();

}

**try** { // Чекаємо поки всі потоки закінчать свою роботу

**for** (Thread thread : threads) {

thread.join();

}

} **catch** (InterruptedException e) {

e.printStackTrace();

}

**return** resultMatrix;

}

}

Для множення матриці ми розділяємо її між доступними потоками (наприклад на скріншоті матриця 8 х 8 і доступно 4 потоки)

Text

Description automatically generated  
Відповідно кожен потік працюватиме з 2 рядками, а в самому вкладеному циклі відбувається підрахунок значення елементу матриці за звичайним алгоритмом множення матриць

Завдання №2:

Суть алгоритму Фокса полягає в тому що ми розбиваємо нашу матрицю на підзадачі, відповідно до доступної кількості потоків. Ці підзадачі це матриці меншого розміру, так звані блоки, множення яких виконується паралельно

Для реалізації алгоритму Фокса будемо використовувати вже створену раніше MatrixEntity. В цьому випадку алгоритм виходить більш об’ємним, тому для зручності винесемо роботу яка виконується в потоці до окремого класу який буде називатися FoxCalculatorThread. В класі FoxCalculator виконується «розбиття» матриці на підзадачі відповідно до кількості доступних тредів, а розмір «підматриці» записано до змінної step. Далі до масиву тредів додаємо наші підзадачі з множення матриць. В класі FoxCalculatorThread, в методі run() ми виконуємо множення наших маленьких блоків першої матриці, на блоки другої.

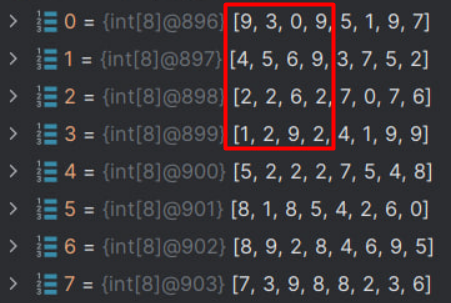
Text

Description automatically generatedText

Description automatically generated

На рисунку бачимо 2 матриці, з яких виконується копіювання менших блоків (в методі copyBlock() ), в цьому випадку в нас 4 потоки тому матриця розміром 8 х 8 розділена на 4 блоки розміром 4 х 4. Для підрахунку значення одного блоку копіюємо спочатку першу частину матриці 1 і матриці 2

A picture containing text

Description automatically generated

і перемножаємо їх між собою за звичайним алгоритмом множення матриці   
SequentialCalculator().multiplyMatrix(blockFirst, blockSecond); В результаті множення ми отримуємо resBlock, значення якого і записуємо до resultMatrix у відповідний блок (як бачимо в цій матриці всі інші блоки вже підраховані і видно наш пустий блок який ще в процесі обчислення):

Text

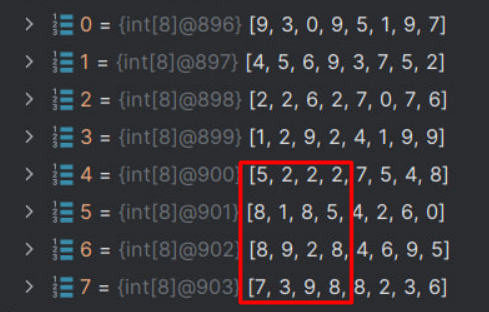
Description automatically generated

На другій ітерації (у випадку з матрицею таких розмірів) ми множимо між собою наступний блок 1 матриці

Text

Description automatically generated

На інший блок другої матриці



І далі отримані результати додаємо до результатів отриманих в цьому блоці результуючої матриці на першій ітерації.

Клас FoxCalculator:

**package** lab2.Algorithms.Fox;

**import** **lab2.Tools.MatrixEntity**;

**import** **lombok.Getter**;

**import** **lombok.RequiredArgsConstructor**;

**import** **lombok.Setter**;

/\*\*

\* Клас FoxCalculator використовується для многопоточного множення матриць за алгоритмом Фокса

\* суть алгоритму полянає в поділі матриці на частини які виконуються в окремих потоках

\*/

**@Getter**

**@Setter**

**@RequiredArgsConstructor**

**public** **class** **FoxCalculator** {

**private** MatrixEntity matrixEntity1;

**private** MatrixEntity matrixEntity2;

**private** **int** threadsCount;

**private** MatrixEntity resultMatrix;

/\*\*

\* Конструктор класу FoxCalculator, в який передаємо матриці, які будемо множити та кількість потоків

\* в яких буде виконуватися алгоритм

\* @param matrixEntity1 перша матриця

\* @param matrixEntity2 друга матриця

\* @param threadsCount кількість потоків

\*/

**public** **FoxCalculator**(MatrixEntity matrixEntity1, MatrixEntity matrixEntity2, **int** threadsCount) {

**this**.matrixEntity1 = matrixEntity1;

**this**.matrixEntity2 = matrixEntity2;

**this**.resultMatrix = **new** MatrixEntity(matrixEntity1.getRowsSize(), matrixEntity2.getColumnsSize());

**if** (threadsCount > matrixEntity1.getRowsSize() \* matrixEntity2.getColumnsSize() / **4**) {

**this**.threadsCount = matrixEntity1.getRowsSize() \* matrixEntity2.getColumnsSize() / **4**;

} **else** **this**.threadsCount = Math.max(threadsCount, **1**);

}

**public** MatrixEntity **multiplyMatrix**() {

var step = (**int**) Math.ceil(**1.0** \* matrixEntity1.getRowsSize() / (**int**) Math.sqrt(threadsCount));

FoxCalculatorThread[] threads = **new** FoxCalculatorThread[threadsCount];

var idx = **0**;

**for** (**int** i = **0**; i < matrixEntity1.getRowsSize(); i += step) {

**for** (**int** j = **0**; j < matrixEntity2.getColumnsSize(); j += step) {

threads[idx] = **new** FoxCalculatorThread(matrixEntity1, matrixEntity2, i, j, step, resultMatrix);

idx++;

}

}

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

threads[i].start();

}

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

**try** {

threads[i].join();

} **catch** (InterruptedException e) {

e.printStackTrace();

}

}

**return** resultMatrix;

}

}

Клас FoxCalculatorThread:

**package** lab2.Algorithms.Fox;

**import** **lab2.Algorithms.Sequential.SequentialCalculator**;

**import** **lab2.Tools.MatrixEntity**;

/\*\*

\* Клас для розпаралеленого множення матриць за алгоритмом Фокса

\*/

**public** **class** **FoxCalculatorThread** **extends** Thread {

**private** **final** MatrixEntity matrixEntity1;

**private** **final** MatrixEntity matrixEntity2;

**private** **final** **int** curRowShift;

**private** **final** **int** curColShift;

**private** **final** **int** blockSize;

**private** **final** MatrixEntity resultMatrix;

/\*\*

\* @param matrixEntity1 - матриця 1

\* @param matrixEntity2 - матриця 2

\* @param curRowShift - початковий рядок

\* @param curColShift - початковий стовпець

\* @param blockSize - розмір блоку матриці для потоку

\* @param resultMatrix - матриця для результату

\*/

**public** **FoxCalculatorThread**(MatrixEntity matrixEntity1, MatrixEntity matrixEntity2, **int** curRowShift,

**int** curColShift, **int** blockSize, MatrixEntity resultMatrix) {

**this**.resultMatrix = resultMatrix;

**this**.matrixEntity1 = matrixEntity1;

**this**.matrixEntity2 = matrixEntity2;

**this**.curRowShift = curRowShift;

**this**.curColShift = curColShift;

**this**.blockSize = blockSize;

}

**@Override**

**public** **void** **run**() {

var m1RowSize = blockSize;

var m2ColSize = blockSize;

**if** (curRowShift + blockSize > matrixEntity1.getRowsSize())

m1RowSize = matrixEntity1.getRowsSize() - curRowShift;

**if** (curColShift + blockSize > matrixEntity2.getColumnsSize())

m2ColSize = matrixEntity2.getColumnsSize() - curColShift;

**for** (**int** k = **0**; k < matrixEntity1.getRowsSize(); k += blockSize) {

var m1ColSize = blockSize;

var m2RowSize = blockSize;

**if** (k + blockSize > matrixEntity2.getRowsSize()) {

m2RowSize = matrixEntity2.getRowsSize() - k;

}

**if** (k + blockSize > matrixEntity1.getColumnsSize()) {

m1ColSize = matrixEntity1.getColumnsSize() - k;

}

var blockFirst = copyBlock(matrixEntity1, curRowShift, curRowShift + m1RowSize,

k, k + m1ColSize);

var blockSecond = copyBlock(matrixEntity2, k, k + m2RowSize,

curColShift, curColShift + m2ColSize);

var resBlock = **new** SequentialCalculator().multiplyMatrix(blockFirst, blockSecond);

**for** (**int** i = **0**; i < resBlock.getRowsSize(); i++) {

**for** (**int** j = **0**; j < resBlock.getColumnsSize(); j++) { // елементи результуючого блоку додаємо до

resultMatrix.set(i + curRowShift, j + curColShift, resBlock.get(i, j) // результату

+ resultMatrix.get(i + curRowShift, j + curColShift)); // відповідного рядка та стовпця результуючої матриці

}

}

}

}

/\*\*

\* @param src - матриця з якої копіюємо

\* @param rowStart - початковий рядок

\* @param rowFinish - кінцевий рядок

\* @param colStart - початковий стовпець

\* @param colFinish - кінцевий стовпець

\* @return копія блоку матриці

\*/

**private** MatrixEntity **copyBlock**(MatrixEntity src, **int** rowStart, **int** rowFinish,

**int** colStart, **int** colFinish) {

var copyMatrix = **new** MatrixEntity(rowFinish - rowStart, colFinish - colStart);

**for** (**int** i = **0**; i < rowFinish - rowStart; i++) {

**for** (**int** j = **0**; j < colFinish - colStart; j++) {

copyMatrix.set(i, j, src.get(i + rowStart, j + colStart));

}

}

**return** copyMatrix;

}

}

Код допоміжного класу SequentialCalculator:

**package** lab2.Algorithms.Sequential;

**import** **lab2.Tools.MatrixEntity**;

**public** **class** **SequentialCalculator** {

**public** MatrixEntity **multiplyMatrix**(MatrixEntity matrixEntity1, MatrixEntity matrixEntity2) {

**if** (matrixEntity1.getColumnsSize() != matrixEntity2.getRowsSize()) {

**throw** **new** **IllegalArgumentException**("matrices cannot be multiplied because the " +

"number of columns of matrix A is not equal to the number of rows of matrix B.");

}

var resultMatrix = **new** MatrixEntity(matrixEntity1.getRowsSize(), matrixEntity2.getColumnsSize());

**for** (**int** i = **0**; i < matrixEntity1.getRowsSize(); i++) {

**for** (**int** j = **0**; j < matrixEntity2.getColumnsSize(); j++) {

**for** (**int** k = **0**; k < matrixEntity1.getColumnsSize(); k++) {

resultMatrix.set(i, j, resultMatrix.get(i, j) + matrixEntity1.get(i, k) \* matrixEntity2.get(k, j));

}

}

}

**return** resultMatrix;

}

}

Завдання №3:

Виконаємо заміри часу для послідовного, стрічкового та алгоритму Фокса на матрицях різної розмірності.

Можна помітити що на невеликих матрицях (+- до 500) різниця невелика, а на матрицях великих розмірів алгоритм Фокса та стрічковий алгоритм мають значну перевагу за рахунок розділення роботи між кількома потоками, що прискорює час виконання підрахунків у великих об’ємах.

Text

Description automatically generatedText

Description automatically generated

Text

Description automatically generatedText

Description automatically generated

Text

Description automatically generatedText

Description automatically generated

Для експериментів використовувався наступний код з перевіркою на відповідність кінцевого результату:

**package** lab2.test;

**import** **lab2.Tools.MatrixEntity**;

**import** **lab2.Tools.RandomMatrixGenerator**;

**public** **class** **Task3Test** {

**public** **static** **void** **main**(String[] args) {

RandomMatrixGenerator randomMatrixGenerator = **new** RandomMatrixGenerator();

var MATRIX\_SIZE = **1500**;

var THREADS\_COUNT = **4**;

var startTime = System.currentTimeMillis();

var endTime = System.currentTimeMillis();

var matrixEntity = **new** MatrixEntity(

randomMatrixGenerator

.generateRandomMatrix(MATRIX\_SIZE, MATRIX\_SIZE)

.getMatrix());

var matrixEntity2 = **new** MatrixEntity(

randomMatrixGenerator

.generateRandomMatrix(MATRIX\_SIZE, MATRIX\_SIZE)

.getMatrix());

var sequentialCalculator = **new** lab2.Algorithms.Sequential.SequentialCalculator();

var parallelCalculator = **new** lab2.Algorithms.Parallel.ParallelCalculator();

var foxCalculator = **new** lab2.Algorithms.Fox.FoxCalculator(matrixEntity, matrixEntity2, THREADS\_COUNT);

// Sequential test

startTime = System.currentTimeMillis();

var seqRes = **new** MatrixEntity(sequentialCalculator.multiplyMatrix(matrixEntity, matrixEntity2).getMatrix());

endTime = System.currentTimeMillis();

System.out.println("Sequential: " + (endTime - startTime) + " ms " + "for " + MATRIX\_SIZE + " matrix size" );

// Parallel test

startTime = System.currentTimeMillis();

var parRes = **new** MatrixEntity(parallelCalculator.multiplyMatrix(matrixEntity, matrixEntity2, THREADS\_COUNT).getMatrix());

endTime = System.currentTimeMillis();

System.out.println("Parallel: " + (endTime - startTime) + " ms " + "for " + MATRIX\_SIZE + " matrix size" );

// Fox test

startTime = System.currentTimeMillis();

var foxRes = **new** MatrixEntity(foxCalculator.multiplyMatrix().getMatrix());

endTime = System.currentTimeMillis();

System.out.println("Fox: " + (endTime - startTime) + " ms " + "for " + MATRIX\_SIZE + " matrix size" );

// Check results

**for** (**int** i = **0**; i < MATRIX\_SIZE; i++) {

**for** (**int** j = **0**; j < MATRIX\_SIZE; j++) {

**if** (seqRes.get(i, j) != parRes.get(i, j) || seqRes.get(i, j) != foxRes.get(i, j)) {

System.out.println("Error");

**return**;

}

}

}

}

}

Завдання №4:

В цьому експерименті нам необхідно порівняти час виконання алгоритмів в залежності від кількості потоків в яких вони будуть виконуватися.

Як видно з графіку обидва алгоритми показують найкращі результати при використанні 16 потоків, а в подальшому зміни вже стають незначними.

Графік прискорення алгоритмів в залежності від кількості потоків:

Text

Description automatically generatedText

Description automatically generated

Text

Description automatically generatedText

Description automatically generated

Text

Description automatically generatedText

Description automatically generated

Text

Description automatically generatedText

Description automatically generated

Text

Description automatically generatedText

Description automatically generated

Text

Description automatically generated

**Висновок:** Під час виконання даної лабораторної роботи я освоїв нові методи множення матриць використовуючи паралельні алгоритми для їх множення.