



Міністерство освіти і науки України  
Національний технічний університет України  
«Київський політехнічний інститут імені Ігоря Сікорського»  
Факультет інформатики та обчислювальної техніки  
Кафедра інформатики і програмної інженерії

Лабораторна робота №2  
з дисципліни  
Технології паралельних обчислень

Виконав:

студент групи ІІ-13:  
Бабашев О. Д.

Перевірив:

ст.викл.  
Дифучин А. Ю.

Київ 2024

## **Завдання до комп'ютерного практикуму 2 «Розробка паралельних алгоритмів множення матриць та дослідження їх ефективності»**

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

### **Хід роботи**

#### **Лістинг коду:**

#### **ParallelFoxMultiplier.java**

```
public class ParallelFoxMultiplier implements IMultiplier {  
    private final int threads;  
  
    public ParallelFoxMultiplier(int threads) {  
        this.threads = threads;  
    }  
  
    @Override  
    public Result multiply(int[][] A, int[][] B) {  
        int n = A.length;  
        Result result = new Result(n, n);  
        int blockSize = (int) Math.ceil((double) n / threads);  
        Thread[] threads = new Thread[this.threads];  
  
        for (int threadId = 0; threadId < this.threads; threadId++) {  
            int finalThreadId = threadId;
```

```

threads[threadId] = new Thread() -> {

    for (int block = 0; block < this.threads; block++) {
        int startRow = finalThreadId * blockSize;
        int endRow = Math.min((finalThreadId + 1) * blockSize, n);

        for (int i = startRow; i < endRow; i++) {
            for (int j = 0; j < n; j++) {
                int sum = 0;
                for (int k = block * blockSize; k < Math.min((block + 1) *
blockSize, n); k++) {
                    sum += A[i][k] * B[k][j];
                }
                int currentValue = result.getData(i, j);
                result.setData(i, j, currentValue + sum);
            }
        }
    }

});
threads[threadId].start();
}

for (Thread thread : threads) {
    try {
        thread.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

```

```

        return result;
    }
}

```

### **ParallelMultiplier.java**

```
public class ParallelMultiplier implements IMultiplier {
```

```
    private final int threads;
```

```

    public ParallelMultiplier(int threads) {
        this.threads = threads;
    }

```

```
@Override
```

```
public Result multiply(int[][] A, int[][] B) {
```

```
    int n = A.length;
```

```
    Result result = new Result(n, n);
```

```
    Thread[] threads = new Thread[this.threads];
```

```
    for (int i = 0; i < this.threads; i++) {
```

```
        final int threadIndex = i;
```

```
        threads[i] = new Thread(() -> {
```

```
            int chunkSize = n / this.threads;
```

```
            int startRow = threadIndex * chunkSize;
```

```
            int endRow = (threadIndex == this.threads - 1) ? n : startRow +
chunkSize;
```

```
            for (int row = startRow; row < endRow; row++) {
```

```
                for (int col = 0; col < n; col++) {
```

```
                    int sum = 0;
```

```
                    for (int k = 0; k < n; k++) {
```

```

        sum += A[row][k] * B[k][col];
    }
    result.setData(row, col, sum);
}
}
});

    threads[i].start();
}

for (Thread thread : threads) {
    try {
        thread.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

    return result;
}
}

IMultiplier.java
public interface IMultiplier {
    Result multiply(int[][] A, int[][] B);
}

Main.java
import java.util.Random;

public class Main {
    private static final Random random = new Random();

```

```

public static int[][] initMatrix(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] = random.nextInt(11);
        }
    }
    return matrix;
}

```

```

public static double measureTimeOfPerformance(IMultiplier multiplier, int[][]
A, int[][] B, int iterations, Result[] outResult) {
    long totalTime = 0;

    for (int i = 0; i < iterations; i++) {
        long startTime = System.currentTimeMillis();
        outResult[0] = multiplier.multiply(A, B);
        long endTime = System.currentTimeMillis();
        totalTime += (endTime - startTime);
    }
    return totalTime / (iterations * 1000.0);
}

```

```

public static void main(String[] args){
    int[] sizes = {1000,1500,2000,2500};
    int[] threadsArray = {4,9,16};
    int iterations = 1;

    for (int size : sizes) {

```

```

int[][] A = initMatrix(size);
int[][] B = initMatrix(size);

System.out.println("Size: " + size);

IMultiplier sequentialMultiplier = new SequentialMultiplier();
Result[] sequentialResult = new Result[1];
double sequentialTime = measureTimeOfPerformance(sequentialMultiplier,
A, B, iterations, sequentialResult);

System.out.printf("Sequential: " + sequentialTime + "\n");

//sequentialResult[0].printMatrix();
//System.out.println();

for (int threads : threadsArray) {
    IMultiplier parallelMultiplier = new ParallelMultiplier(threads);
    Result[] parallelResult = new Result[1];
    double parallelTime = measureTimeOfPerformance(parallelMultiplier,
A, B, iterations, parallelResult);

    IMultiplier parallelFoxMultiplier = new ParallelFoxMultiplier(threads);
    Result[] parallelFoxResult = new Result[1];
    double foxTime = measureTimeOfPerformance(parallelFoxMultiplier, A,
B, iterations, parallelFoxResult);

    System.out.printf("Threads: " + threads + " ");
    System.out.printf("Parallel: " + parallelTime + " ");
    System.out.printf("Parallel Fox: " + foxTime + "\n");
}

```

```

        if (!MatrixHelper.NotEqual(sequentialResult[0].getMatrix(),
parallelResult[0].getMatrix())) {
            System.err.println("Sequential and Parallel results do not match for " +
threads + " threads!");
        }

        if (MatrixHelper.NotEqual(sequentialResult[0].getMatrix(),
parallelFoxResult[0].getMatrix())) {
            System.err.println("Sequential and Parallel Fox results do not match
for " + threads + " threads!");
        }
    }
    System.out.println();
}
}
}
}

```

### **MatrixHelper.java**

```

public class MatrixHelper {
    public static boolean NotEqual(int[][] matrix1, int[][] matrix2) {
        if (matrix1.length != matrix2.length || matrix1[0].length != matrix2[0].length)
        {
            return true;
        }
        for (int i = 0; i < matrix1.length; i++) {
            for (int j = 0; j < matrix1[i].length; j++) {
                if (matrix1[i][j] != matrix2[i][j]) {
                    return true;
                }
            }
        }
    }
}

```



```
    }  
    return false;  
}  
}
```

### **Result.java**

```
public class Result {  
  
    private final int[][] data;  
  
    public Result(int rows, int cols) {  
        data = new int[rows][cols];  
    }  
  
    public synchronized void setData(int row, int col, int value) {  
        data[row][col] = value;  
    }  
  
    public synchronized int getData(int row, int col) {  
        return data[row][col];  
    }  
  
    public int[][] getMatrix() {  
        return data;  
    }  
  
    public void printMatrix() {  
        for (int[] row : data) {  
            for (int val : row) {
```

```

        System.out.print(val + "\t");
    }

    System.out.println();
}
}
}

```

### **SequentialMultiplier.java**

```
public class SequentialMultiplier implements IMultiplier {
```

```
    @Override
```

```
    public Result multiply(int[][] A, int[][] B) {
```

```
        int n = A.length;
```

```
        Result result = new Result(n, n);
```

```
        for (int i = 0; i < n; i++) {
```

```
            for (int j = 0; j < n; j++) {
```

```
                int sum = 0;
```

```
                for (int k = 0; k < n; k++) {
```

```
                    sum += A[i][k] * B[k][j];
```

```
                }
```

```
                result.setData(i, j, sum);
```

```
            }
```

```
        }
```

```
        return result;
```

```
    }
```

```
}
```

## Результати виконання коду:

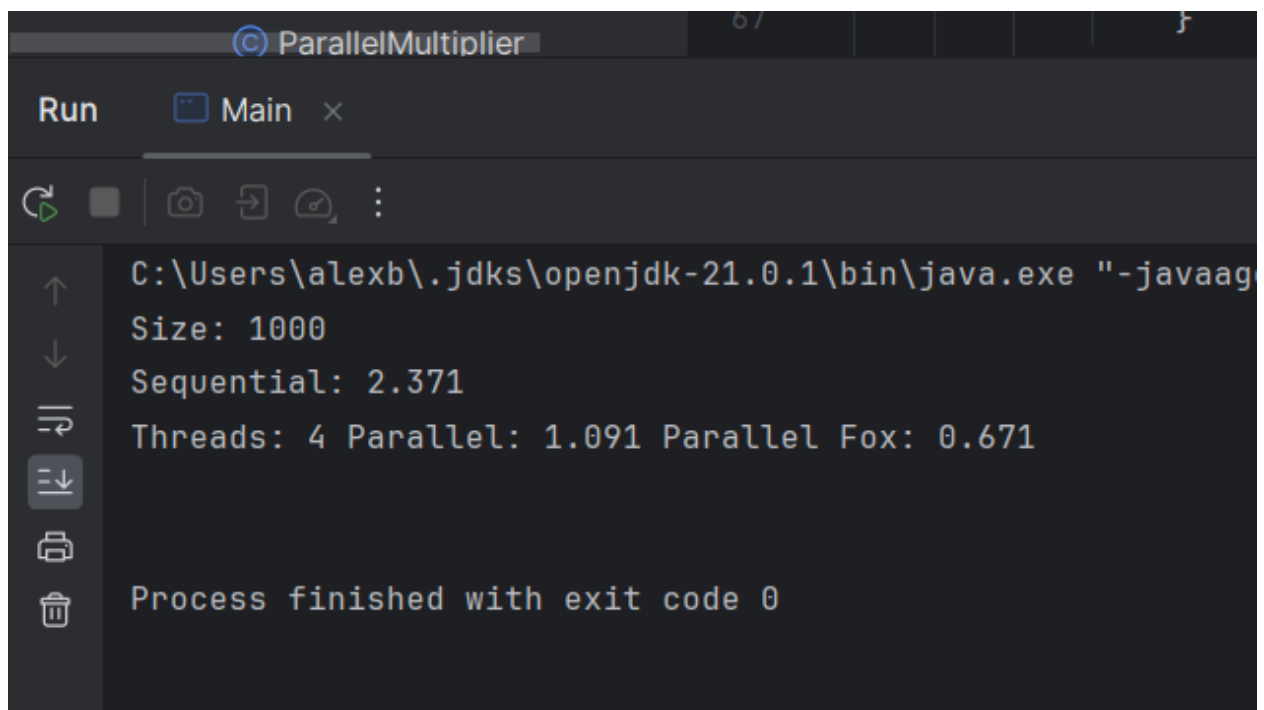
Для алгоритму Фокса кількість потоків повинна бути квадратом цілого числа, щоб матриці можна було поділити на блоки рівного розміру. Відповідно і матриці мають бути квадратні.

Стрічковий алгоритм. Цей метод розподіляє роботу між потоками на основі рядків матриці. Кожен потік обробляє певний набір рядків. Цей підхід простий, але може призвести до нерівномірного завантаження потоків, особливо якщо кількість рядків не ділиться рівномірно на кількість потоків.

Алгоритм Фокса. Використовує алгоритм Фокс для розподілу роботи. Кожен потік обробляє блоки матриці, де кожен блок включає рядки і стовпці. Цей підхід може забезпечити більш рівномірне розподілення роботи між потоками, але є складнішим в реалізації.

Після виконання множення матриць стрічковим алгоритмом та алгоритмом Фокса перевіряється коректність отриманих результатів шляхом порівняння результуючої матриці та матриці отриманої після виконання класичного послідовного множення матриць. В разі помилкової дії алгоритму і не співставлення результатів виконання множення програма виводить помилку.

Перевіримо коректність роботи розроблених алгоритмів запустивши програму.



```
ParallelMultiplier
Run Main x
C:\Users\alexb\.jdk\openjdk-21.0.1\bin\java.exe "-javaag
Size: 1000
Sequential: 2.371
Threads: 4 Parallel: 1.091 Parallel Fox: 0.671
Process finished with exit code 0
```

Рисунок 1.1 – Результат виконання програми

Отримано коректний результат множення матриць розробленими стрічковим алгоритмом та алгоритмом Фокса.

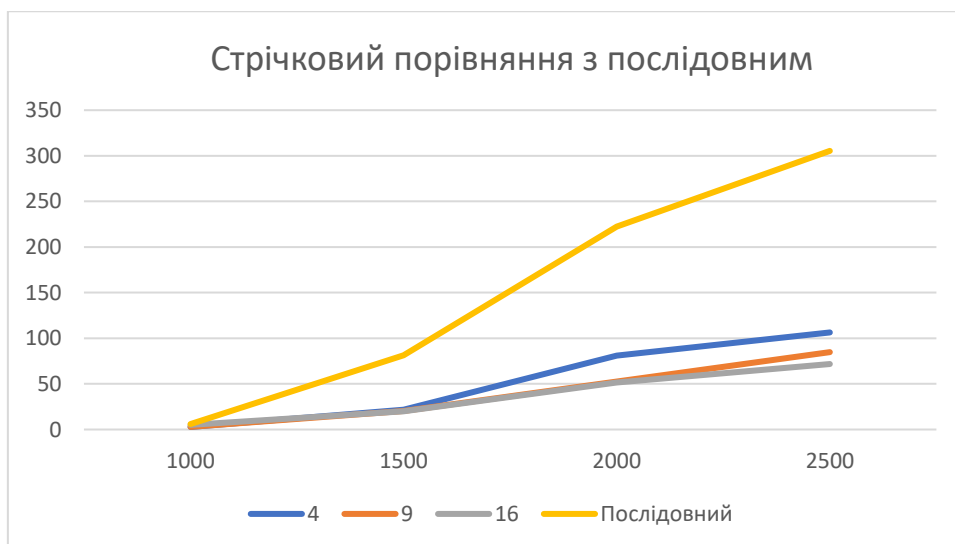
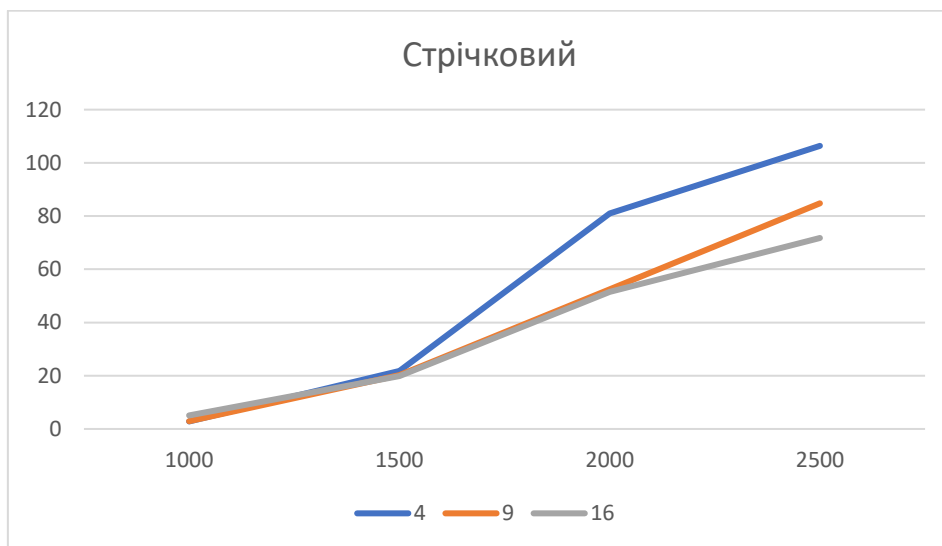
Далі виконаємо експерименти.

Для проведення більш об'єктивного заміру по часу виконання алгоритмів та отримання точного порівняння їх ефективності, проведемо обрахування середнього часу виконання 10 ітерацій виконання.

Отримано наступні результати експериментів.

Виконання стрічкового алгоритму в секундах.

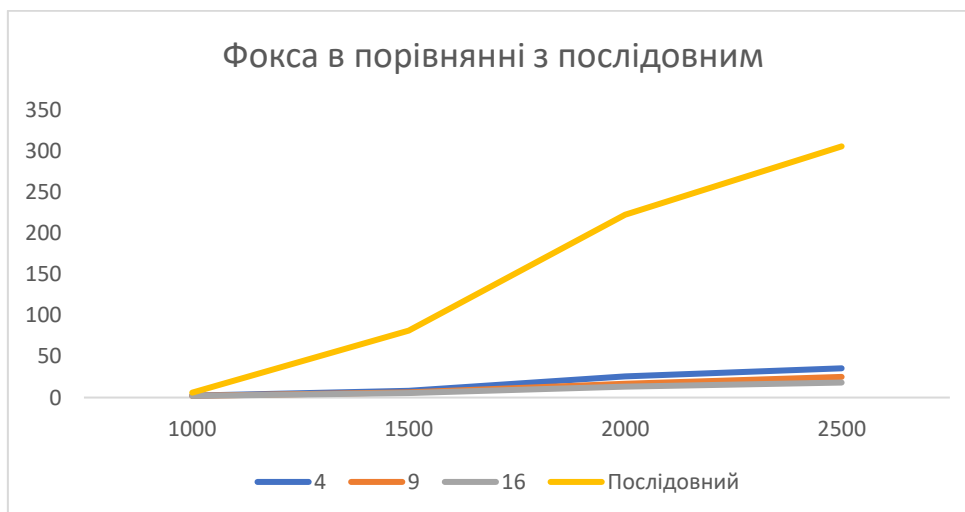
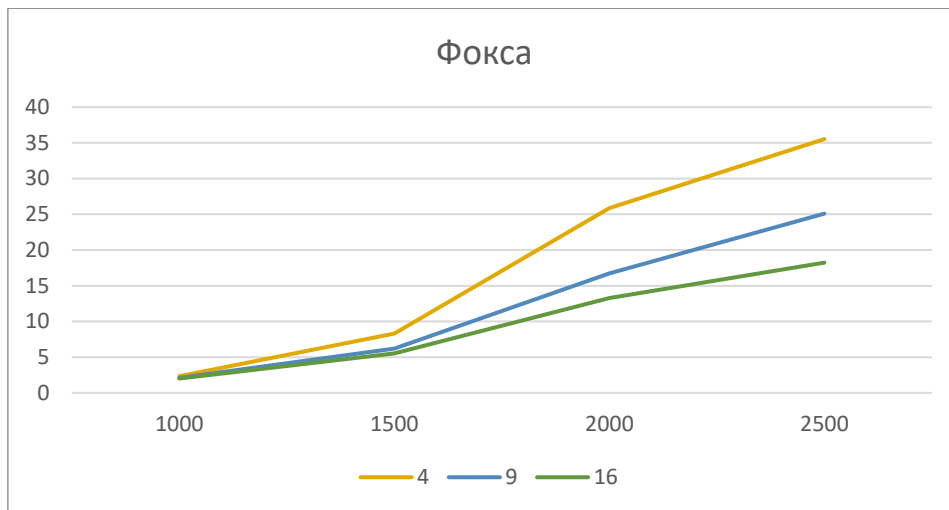
np size	Послідовний	4	9	16
1000	5,967	2,818	2,908	5,08
1500	81,201	21,817	20,274	19,856
2000	222,327	81,012	52,53	51,478
2500	305,415	106,386	84,792	71,77



Зазначимо, що стрічковий алгоритм працює значно швидше за послідовний, також по графіка можем побачити, що зі збільшенням кількості потоків, зменшується швидкість множення.

## Виконання алгоритму Фокса в секундах.

nr size	Послідовний	4	9	16
1000	5,967	2,334	2,101	2,004
1500	81,201	8,314	6,173	5,53
2000	222,327	25,887	16,708	13,275
2500	305,415	35,528	25,101	18,233

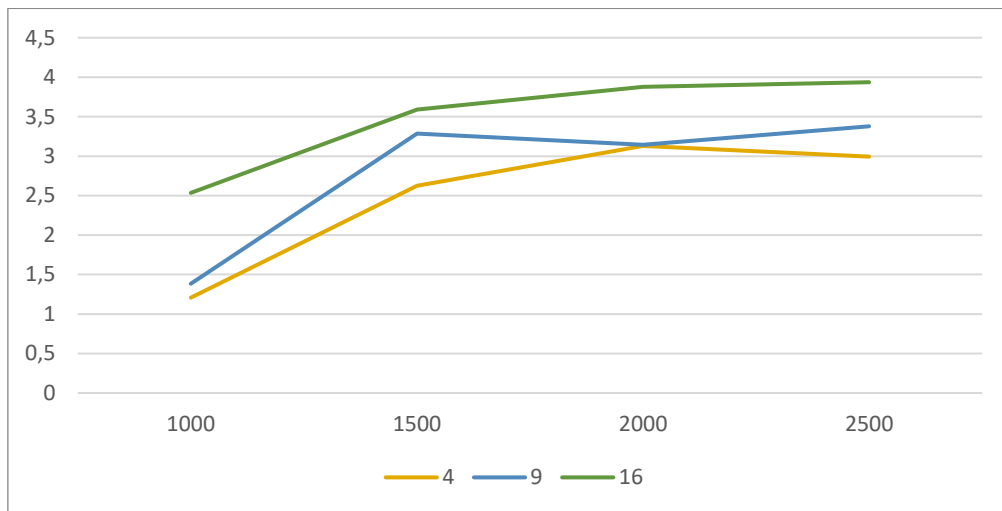


Бачимо, що алгоритм Фокса виконується швидше за стрічковий алгоритм через ефективніший розподіл обчислень між потоками і краще використання кеш-пам'яті і відповідно він є ще ефективнішим в порівнянні з послідовним. Також зазначимо, що зі збільшенням кількості потоків значно зменшується швидкість виконання множення, це можна пояснити тим, що алгоритм Фокса призначений для розподілених систем, де кількість процесів має бути велика.

Наведемо дані про відношення відповідних швидкостей виконання множення алгоритмом Фокса та стрічковим алгоритмом.

Таблиця пришвидшень алгоритму Фокса

np size	4	9	16
1000	1,207369323	1,384103	2,53493
1500	2,624127977	3,284303	3,590597
2000	3,129447213	3,144003	3,877815
2500	2,994426931	3,378033	3,936269



Аналізуючи наведені дані, робимо висновок, що прискорення зростає зі збільшенням кількості потоків та зі збільшенням розмірності матриць.

Таким чином, збільшення кількості потоків до оптимального рівня дозволяє алгоритму Фокса ефективніше використовувати обчислювальні ресурси і досягти значного прискорення виконання множення матриць.

### Висновок

У ході виконання лабораторної роботи було розроблено алгоритми паралельного множення матриць, а саме Фокса та стрічковий. Було проведено експерименти та досліджено ефективність розроблених алгоритмів, результати експериментів очікувані.