

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. І. СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

Лабораторна робота №2

із дисципліни «Технології паралельних та розподілених обчислень»  
на тему «Розробка паралельних алгоритмів множення матриць та  
дослідження їх ефективності»

Підготував:

Студент ФІОТ, гр. ІТ-83,

Троян О.С.

Перевірив:

пос. Дифучина О. Ю.

Київ 2020

**Мета роботи:** Створити паралельні алгоритми множення матриць та дослідити їх ефективність

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

1. Стрічковий алгоритм - кожен процес розраховує рядок результуючої матриці, кожна ітерація – один елемент рядка результуючої матриці  
Ініціалізуємо потоки даними для розрахунку (рядки)

```
private void prepareWorkers(){
    for (int i = 0; i < result.getMatrix().length; i++) {
        workers.add(new TapeWorker(i, i, first, second, result.getMatrix()));
    }
}
```

```
public TapeWorker(int row, int column, int[][] first, int[][] second, int[][] result) {
    this.row = row;
    this.column = column;
    this.first = first;
    this.second = second;
    this.result = result;
}

@Override
public void run(){
    result[row][column] = multiplyRowAndColumn(row, column);
    column = (column + 1) % result[0].length;
}

private int multiplyRowAndColumn(int row, int column){
    int result = 0;
    for (int i = 0; i < first[0].length; i++) {
        result += first[row][i] * second[i][column];
    }
    return result;
}
```

Кожен потік розраховує рядок результуючої матриці.

Кількість процесів == кількість рядків першої матриці.

```
public int getIterationsCount(){
    return first[0].length;
}
```

```
public class IterationsManager {  
    public Result iterate(MultiplicationStrategy strategy){  
        for (int i = 0; i < strategy.getIterationsCount(); i++) {  
            strategy.nextIteration();  
        }  
        return strategy.getResult();  
    }  
}
```

Розрахунок (використовується обмеження кількості потоків)

```
public void nextIteration() {
    ExecutorService executor = Executors.newFixedThreadPool(threadsCount);
    for (TapeWorker worker : workers) {
        executor.execute(worker);
    }
    executor.shutdown();
    try {
        executor.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

Результат (час та останній рядок множення одиничних матриць  
100 на 100)

[illegible]

2. Алгоритм Фокса – вираховуємо кількість процесів залежно від порядку матриці, це повинно бути число корень якого є кратним порядку матриці. А розмір блоків які будуть розраховуватися процесами це  $m=n/q$ , де  $n$  - розмірність матриці, а  $q$  кількість процесів.

## Ініціалізуємо потоки:

```

public FoxStrategy(int[][] first, int[][] second, int threadsCount, int blockRank) {
    this.blockRank = blockRank;
    this.first = first;
    this.threadsCount = threadsCount;
    this.result = new int[first.length][second[0].length];
    horizontalBlocksCount = first[0].length / blockRank;
    verticalBlocksCount = first.length / blockRank;
    workers = new FoxWorker[verticalBlocksCount][horizontalBlocksCount];
    for (int i = 0; i < workers.length; i++) {
        for (int j = 0; j < workers[0].length; j++) {
            workers[i][j] = new FoxWorker(i, j, blockRank, first, second, result);
        }
    }
}

```

Кожен потік перемножує блоки двох основних матриць, і сумує отримані результати у відповідних блоках результуючої матриці.

```

@Override
public void run() {
    int[][] firstBlock = getBlock(first, firstBlockRow, firstBlockColumn);
    int[][] secondBlock = getBlock(second, secondBlockRow, secondBlockColumn);

    int[][] resultBlock = InlineStrategy.multiply(firstBlock, secondBlock).getMatrix();
    applyBlockToResult(resultBlock);

    firstBlockColumn = (firstBlockColumn + 1) % blocksCount;
    secondBlockRow = (secondBlockRow + 1) % blocksCount;
}

```

```

private int[][] getBlock(int[][] matrix, int blockRow, int blockColumn){
    int[][] block = new int[blockRank][blockRank];
    for (int i = 0; i < blockRank; i++) {
        for (int j = 0; j < blockRank; j++) {
            block[i][j] = matrix[blockRow * blockRank + i][blockColumn * blockRank + j];
        }
    }
    return block;
}

private void applyBlockToResult(int[][] block){
    for (int i = 0; i < blockRank; i++) {
        for (int j = 0; j < blockRank; j++) {
            result[firstBlockRow * blockRank + i][secondBlockColumn * blockRank + j] += block[i][j];
        }
    }
}

```

Кількість ітерацій == кількість рядків поділена на ранг блоку

```
@Override
public int getIterationsCount() {
    return first[0].length / blockRank;
}
```

```
@Override
public void nextIteration() {
    ExecutorService executor = Executors.newFixedThreadPool(threadsCount);
    for (var array : workers) {
        for (FoxWorker worker : array){
            executor.execute(worker);
        }
    }
    executor.shutdown();
    try {
        executor.awaitTermination(Long.MAX_VALUE, TimeUnit.NANOSECONDS);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

## Результати

[illegible]

3. Для перевірки швидкодії з різною кількістю потоків буде використовуватись одиничні матриці 500 на 500, кількість потоків: 2, 4, 8, 16, 32, 64, 128, 256

```
Workers test: 2 workers
Fox method (500): 778
Tape method: 1669

Workers test: 4 workers
Fox method (500): 682
Tape method: 1507

Workers test: 8 workers
Fox method (500): 533
Tape method: 2835

Workers test: 16 workers
Fox method (500): 683
Tape method: 3980

Workers test: 32 workers
Fox method (500): 1225
Tape method: 9699

Workers test: 64 workers
Fox method (500): 2258
Tape method: 14126

Workers test: 128 workers
Fox method (500): 3596
Tape method: 28635

Workers test: 256 workers
Fox method (500): 5835
```

Як бачимо при використанні великої кількості потоків час виконання зменшується, так як для переключення між потоками та склеювання результатів їх виконання потрібно більше часу. Час для стрічкового алгоритму майже лінійно зростає з збільшенням кількості потоків, що більша за необхідну, у той час як час для алгоритма фокса різко зростає після проходження грані, де кількість потоків перевищує необхідну.

Алгоритм Фокса при великій кількості потоків показує значне зменшення ефективності. Я вважаю це через зменшення кількості блоків, та збільшення їх розміру.

#### 4. Заміри часу для різних розмірностей матриці

##### 4.1. 8 потоків

```
Size test: 100 matrix rank  
Fox method: 106  
Tape method: 446  
  
Size test: 200 matrix rank  
Fox method: 120  
Tape method: 917  
  
Size test: 20 matrix rank  
Fox method: 7  
Tape method: 90  
  
Size test: 50 matrix rank  
Fox method: 16  
Tape method: 204
```

```
Size test: 500 matrix rank  
Fox method: 802  
Tape method: 3093  
  
Size test: 1000 matrix rank  
Fox method: 4003  
Tape method: 12623  
  
Size test: 1500 matrix rank  
Fox method: 11392  
Tape method: 36492
```

##### 4.2. 16 потоків

```
Size test: 100 matrix rank  
Fox method: 155  
Tape method: 970  
  
Size test: 200 matrix rank  
Fox method: 254  
Tape method: 2000  
  
Size test: 20 matrix rank  
Fox method: 4  
Tape method: 155  
  
Size test: 50 matrix rank  
Fox method: 37  
Tape method: 389
```



```
Size test: 500 matrix rank  
Fox method: 965  
Tape method: 4796  
  
Size test: 1000 matrix rank  
Fox method: 5419  
Tape method: 15530  
  
Size test: 1500 matrix rank  
Fox method: 11625  
Tape method: 39311
```

#### 4.3. 2 потоки

```
Size test: 100 matrix rank  
Fox method: 50  
Tape method: 125  
  
Size test: 200 matrix rank  
Fox method: 73  
Tape method: 335  
  
Size test: 20 matrix rank  
Fox method: 4  
Tape method: 41  
  
Size test: 50 matrix rank  
Fox method: 9  
Tape method: 158
```

```
Size test: 500 matrix rank  
Fox method: 864  
Tape method: 1588  
  
Size test: 1000 matrix rank  
Fox method: 4036  
Tape method: 18247  
  
Size test: 1500 matrix rank  
Fox method: 16783  
Tape method: 63037  
  
Size test: 2000 matrix rank  
Fox method: 32177  
Tape method: 156285
```

#### 4.4. 4 потоки

```
Size test: 500 matrix rank  
Fox method: 719  
Tape method: 1275  
  
Size test: 1000 matrix rank  
Fox method: 3079  
Tape method: 12210  
  
Size test: 1500 matrix rank  
Fox method: 10620  
Tape method: 38780
```

З результатів видно, що чем більшої розмірності матриця тим ефективніший алгоритм фокса, для матриць малої розмірності ефективніший стрічковий алгоритм за умови використання оптимального розміру пулу потоків.

**Висновок:** Під час виконання лабораторної роботи ми створили паралельні алгоритми множення матриць та дослідили їх ефективність