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

Звіт

Комп‘ютерного практикуму № 2 з дисципліни
«Технології паралельних та розподілених обчислень»

«Розробка паралельних алгоритмів множення матриць та дослідження їх ефективності»

Виконав(ла) ІП-01 Галько М.В. _____
(шифр, прізвище, ім'я, по батькові)

Перевірив(ла) Стеценко І. В. _____
(прізвище, ім'я, по батькові)

Київ 2022

Завдання 1-2. Реалізації:

Реалізуйте стрічковий алгоритм множення матриць. Результат множення записуйте в об’єкт класу Result. **30 балів.**

Реалізуйте алгоритм Фокса множення матриць. **30 балів.**

Хід роботи

Для виконання поставленого завдання стало необхідним створити структуру класів та інтерфейсів. Окрім Main маємо наступні:

1. Matrix – клас матриця; утримує двовимірний масив та має відповідні методи для взаємодії із ячейками. Також отримання копій, вивід, розвертання, додавання, порівняння та генерування матриць.
2. Result – клас для результуючої матриці класу Matrix, має методи взаємодії із нею.
3. IMatricesMultiplier – інтерфейс, що має метод множення матриць, який визначається у імплементуючих класах. Також має статичні методи для зведення двовимірних масивів матриць в одну, розділення однієї матриці на декілька частин.
4. StandardMultiplier – клас, що визначає інтерфейс та виконує стандартне множення матриць
5. TapeMultiplier – клас, що визначає інтерфейс та виконує множення матриць стрічковим алгоритмом. Також утримує record TapeMultiplierTask, що імплементує Callable<Integer> для виконання множення одного стовпця та рядка.
6. FoxMultiplier – клас, що визначає інтерфейс та виконує множення матриць алгоритмом Фокса. Також утримує record FoxMultipluerTask, що імплементує Callable<Integer> для виконання множення матриць (передаємо частини блоки основної матриці).
7. Clock – допоміжний клас для тестування, для контролю часу: відрахування, встановлення часу середнього виконання програми.

Для перевірки коректності роботи алгоритмів виконаємо множення заданих матриць 4 на 4 і звіримо результати виконання на сайті (https://ua.onlinemschool.com/math/assistance/matrix/multiply/) (рис. 1) та у консолі (рис. 2).

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \cdot \begin{pmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{pmatrix} = \begin{pmatrix} 30 & 70 & 110 & 150 \\ 70 & 174 & 278 & 382 \\ 110 & 278 & 446 & 614 \\ 150 & 382 & 614 & 846 \end{pmatrix}$$

Рис. 1 – Множення матриць 4 на 4 онлайн

```
Standard, Tape and Fox result matrices are equal
30 70 110 150
70 174 278 382
110 278 446 614
150 382 614 846

30 70 110 150
70 174 278 382
110 278 446 614
150 382 614 846

30 70 110 150
70 174 278 382
110 278 446 614
150 382 614 846
```

Рис. 2 – Результуючі матриці 3-х алгоритмів

Отже алгоритми працюють коректно і можна переходити до самого тестування на великих матрицях.

Завдання 3-4. Експерименти:

Виконайте експерименти, варіюючи розмірність матриць, які перемножуються, для обох алгоритмів, та реєструючи час виконання алгоритму. Порівняйте результати дослідження ефективності обох алгоритмів. **20 балів.**

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

Для реалізації тестування необхідно зробити 2 цикли: зовнішний (задає розміри матриці від 1000 до 2500 із кроком у 500) та внутрішній (перебирає кількість потоків від 1 до 10). Кожні параметри тестуємо по 10 раз і формуємо середній час виконання операції. Виконавши програму для 2 алгоритмів із параметрами і для стандартного без, формуємо таблиці та графіки для стрічкового алгоритму (рис. 3) та Фокса (рис. 4). В таблицях маємо показники виконання стандартного алгоритму, розмірності матриць, кількість потоків та у нижній частині час виконання відповідного алгоритму. У центрі формуємо дані прискорення використовуючи дані виконання часу стандартного алгоритму та того, що аналізуємо. В кінці кінців формуємо графік у якому поєднуємо результати обох алгоритмів для матриці розміром 2500 (рис. 5).

Standard time	1361	14866	49190	106335
Threads / Size	**1000**	**1500**	**2000**	**2500**
1	1.984	6.422	9.718	10.609
2	3.187	10.440	15.121	16.986
3	3.086	11.989	17.662	19.036
4	2.738	10.867	17.254	19.969
5	2.622	10.641	17.254	19.969
6	2.749	10.596	17.394	20.274
7	2.563	10.274	17.205	20.488
8	2.733	10.649	17.339	20.899
9	2.549	10.447	17.033	20.220
10	2.761	10.345	17.098	20.386
time	686	2315	5062	10023
	427	1424	3253	6260
	441	1240	2785	5586
	497	1368	2851	5325
	519	1397	2851	5325
	495	1403	2828	5245
	531	1447	2859	5190
	498	1396	2837	5088
	534	1423	2888	5259
	493	1437	2877	5216

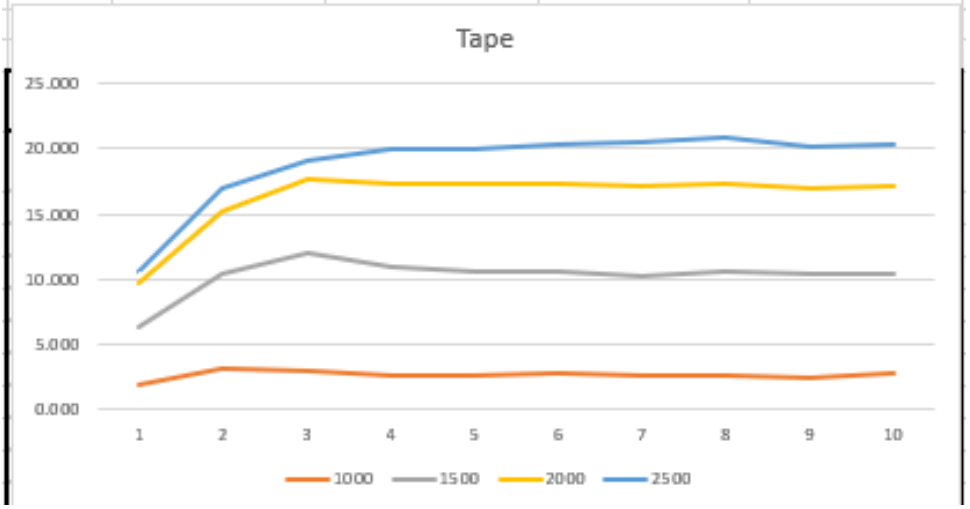

Рис. 3 – таблиця результатів та графік стрічкового алгоритму

Бачимо з рис. 3, що велика кількість потоків більш допомагає із збільшенням розміру матриць. Загалом алгоритм працює й на одному потоці швидше ніж стандартний мінімум у 2 рази в залежності від об’єму інформації. Також при збільшенні розміру матриці, наприклад, у 2 рази, час виконання відповідно збільшується у 4. Також можна помітити, що при збільшенні розміру матриці, алгоритм стає все менш прискореним. На графіку можемо спостерігати “перелом” у прискоренні у кращу сторону при 2-3 потоках. Далі ж збільшення кількості потоків не призводить до значних змін.

Standard time	1361	14866	49190	106335
Threads / Size	**1000**	**1500**	**2000**	**2500**
1	1.086	0.943	1.138	1.084
2	2.050	7.120	6.814	4.274
3	2.291	7.066	8.714	4.624
4	3.472	10.843	9.719	8.278
5	4.075	12.028	14.592	14.499
6	4.112	12.245	14.861	13.357
7	4.188	12.205	15.847	15.158
8	3.900	10.947	13.166	13.909
9	4.582	13.214	15.196	15.880
10	4.614	14.836	16.817	14.379
Fox time	1253	15760	43218	98057
	664	2088	7219	24879
	594	2104	5645	22997
	392	1371	5061	12846
	334	1236	3371	7334
	331	1214	3310	7961
	325	1218	3104	7015
	349	1358	3736	7645
	297	1125	3237	6696
	295	1002	2925	7395

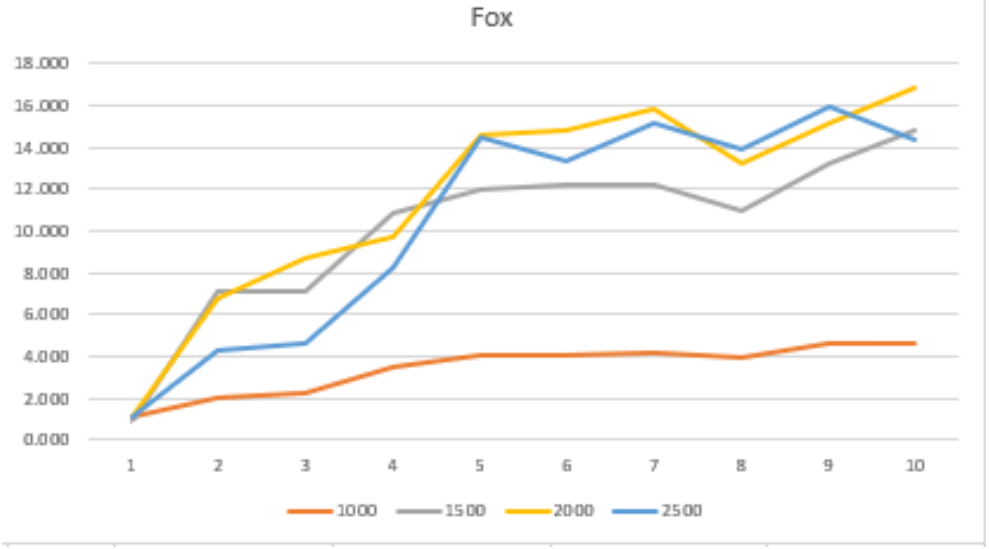

Рис. 4 – таблиця результатів та графік алгоритму Фокса

Алгоритм фокса вже не показує такі передбачувані результати як стрічковий. З рис. 4 видно, що алгоритм слід використовувати на більш великих об’ємах інформації. Також треба відзначити, що кількість блоків розбиття матриці формується наступною формулою: Math.sqrt(capacity - 1) + 1. Де capacity – кількість потоків. Отже, збільшення секцій у матриці буде відбуватися при значеннях кількості потоків: 5, 10. Там і можна спостерігати найбільш продуктивні зони. Загалом, алгоритм при збільшенні потоків тільки йде к прискоренню на відміну від стрічкового, що найбільш продуктивний на 2-3 потоках.

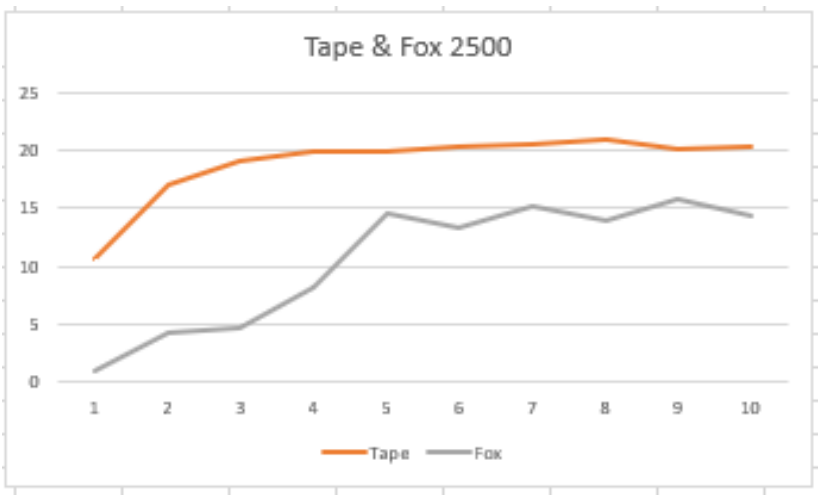

Рис. 5 – Графік прискорення в залежності від кількості потоків для стрічкового алгоритму та Фокса при матриці розмірності 2500

Висновок

З рис. 5 можна зробити висновок, що стрічковий алгоритм має певне прискорення при малих кількостях потоків, але це прискорення зменшується при збільшенні кількості потоків і розмірності матриць. З іншого боку, алгоритм Фокса показує свою ефективність при збільшенні кількості потоків та на великих матрицях. Стрічковий алгоритм стає менш ефективним при збільшенні розмірності матриць.

У цьому звіті було розглянуто реалізацію стрічкового алгоритму та алгоритму Фокса для множення матриць. Були проведені експерименти з варіюванням розмірності матриць та кількості потоків для обох алгоритмів. Результати дослідження показали, що алгоритм Фокса є більш ефективним на великих об'ємах даних та при збільшенні кількості потоків, тоді як стрічковий алгоритм має прискорення при малих кількостях потоків, але втрачає свою ефективність при збільшенні розмірності матриць.

В цілому, обидва алгоритми мають свої переваги та обмеження, і вибір між ними залежить від конкретного контексту застосування, розміру матриць та доступних ресурсів.

Код

```
package Containers;

import java.util.Random;

public class Matrix {
   private final int[][] data;
   private final int rows;
   private final int columns;

   public Matrix(int[][] data) {
       this.data = data;
       this.rows = data.length;
       this.columns = data[0].length;
   }

   public Matrix(int rows, int columns, boolean generateRandom) {
       this.data = generateRandom ? generateMatrix(rows, columns) : new
int[rows][columns];
       this.rows = rows;
       this.columns = columns;
   }

   public int getValue(int i, int j) {
       return data[i][j];
   }

   public void setValue(int i, int j, int value) {
       data[i][j] = value;
   }

   public int getRowsNumber() {
       return rows;
   }

   public int getColumnsNumber() {
       return columns;
   }

   public int[] getRow(int row) {
       return data[row];
   }

   public int[][] getArrayCopy() {
       int[][] copy = new int[rows][columns];
       for (int row = 0; row < rows; row++) {
           System.arraycopy(data[row], 0, copy[row], 0, columns);
       }
       return copy;
   }

   public Matrix getMatrixCopy() {
       return new Matrix(getArrayCopy());
   }

   public void print() {
       for (int[] row : data) {
           for (int value : row) {
               System.out.print(value + " ");
           }
           System.out.println();
       }
       System.out.println();
   }

   public Matrix getTransposedMatrix() {
       int[][] transposed = new int[columns][rows];
       for (int row = 0; row < columns; row++) {
           for (int col = 0; col < rows; col++) {
               transposed[row][col] = data[col][row];
           }
       }
       return new Matrix(transposed);
   }

   public void addMatrix(Matrix matrix) {
```

```
                for (int row = 0; row < rows; row++) {
                    for (int col = 0; col < columns; col++) {
                        data[row][col] += matrix.getValue(row, col);
                    }
                }
            }

            public static boolean compareMatrices(Matrix m1, Matrix m2) {
                if (m1.getRowsNumber() != m2.getRowsNumber() || m1.getColumnsNumber() !=
m2.getColumnsNumber()) {
                    return false;
                }
                for (int row = 0; row < m1.getRowsNumber(); row++) {
                    for (int col = 0; col < m1.getColumnsNumber(); col++) {
                        if (m1.getValue(row, col) != m2.getValue(row, col)) {
                            return false;
                        }
                    }
                }
                return true;
            }

            private int[][] generateMatrix(int rows, int columns) {
                int[][] matrix = new int[rows][columns];
                Random random = new Random();

                for (int i = 0; i < rows; i++) {
                    for (int j = 0; j < columns; j++) {
                        matrix[i][j] = random.nextInt(10);
                    }
                }
                return matrix;
            }
        }
```

```
        package Containers;

        public class Result {
            private final Matrix matrix;

            public Result(int rows, int columns) {
                matrix = new Matrix(rows, columns, false);
            }

            public Result(Matrix matrix) {
                this.matrix = matrix;
            }

            public void setValue(int row, int col, int value) {
                matrix.setValue(row, col, value);
            }

            public Matrix getMatrix() {
                return matrix.getMatrixCopy();
            }

            public void print() {
                matrix.print();
            }
        }
```

```
        package Multipliers;

        import Containers.Matrix;
        import Containers.Result;

        import java.util.concurrent.ExecutionException;

        public interface IMatricesMultiplier {
            Result multiply(Matrix matrixA, Matrix matrixB) throws ExecutionException,
InterruptedException;
            int getPoolCapacity();
            void setPoolCapacity(int capacity);
            boolean isParallelAlgorithm();
```

```
        String getName();

        static Matrix combineMatrices(Matrix[][] resultMatrices) {
            int splitSize = resultMatrices.length;
            int fullSizeI = resultMatrices[0][0].getRowsNumber();
            int fullSizeJ = resultMatrices[0][0].getColumnsNumber();
            Matrix resultMatrix = new Matrix(splitSize * fullSizeI, splitSize *
fullSizeJ, false);

            for (int matrixI = 0; matrixI < splitSize; matrixI++) {
                for (int matrixJ = 0; matrixJ < splitSize; matrixJ++) {
                    for (int i = 0; i < fullSizeI; i++) {
                        for (int j = 0; j < fullSizeJ; j++) {
                            resultMatrix.setValue(matrixI * fullSizeI + i, matrixJ *
fullSizeJ + j, resultMatrices[matrixI][matrixJ].getValue(i, j));
                        }
                    }
                }
            }
            return resultMatrix;
        }

        static Matrix[][] getSplitMatrices(Matrix matrix, int splitNumber) {
            int splitSize = (matrix.getColumnsNumber() - 1) / splitNumber + 1;
            Matrix[][] splitMatrices = new Matrix[splitNumber][splitNumber];

            for (int matrixI = 0; matrixI < splitNumber; matrixI++) {
                for (int matrixJ = 0; matrixJ < splitNumber; matrixJ++) {
                    splitMatrices[matrixI][matrixJ] = new Matrix(splitSize, splitSize,
false);
                    for (int i = 0; i < splitSize; i++) {
                        for (int j = 0; j < splitSize; j++) {
                            if (matrixI * splitSize + i >= matrix.getRowsNumber() ||
matrixJ * splitSize + j >= matrix.getColumnsNumber()) {
                                splitMatrices[matrixI][matrixJ].setValue(i, j, 0);
                                continue;
                            }
                            splitMatrices[matrixI][matrixJ].setValue(i, j,
matrix.getValue(matrixI * splitSize + i, matrixJ * splitSize + j));
                        }
                    }
                }
            }
            return splitMatrices;
        }

        static void printMatrices(Matrix[][] resultMatrices) {
            Matrix combined = combineMatrices(resultMatrices);
            for (int i = 0; i < combined.getRowsNumber(); i++) {
                for (int j = 0; j < combined.getColumnsNumber(); j++) {
                    System.out.print(combined.getValue(i, j) + " ");
                }
                System.out.println();
            }
            System.out.println();
        }
    }
```

```
    package Multipliers;

    import Containers.*;

    public class StandardMultiplier implements IMatricesMultiplier {
        @Override
        public int getPoolCapacity() {
            return 1;
        }
        @Override
        public void setPoolCapacity(int capacity) {
        }
        @Override
        public boolean isParallelAlgorithm() {return false;}

        @Override
        public String getName() {
            return "Standard";
```

```
        }

        @Override
        public Result multiply(Matrix matrixA, Matrix matrixB) {
            int rowsA = matrixA.getRowsNumber();
            int colsB = matrixB.getColumnsNumber();
            Result result = new Result(rowsA, colsB);
            for (int i = 0; i < rowsA; i++) {
                for (int j = 0; j < colsB; j++) {
                    int sum = 0;
                    for (int k = 0; k < matrixB.getRowsNumber(); k++) {
                        sum += matrixA.getValue(i, k) * matrixB.getValue(k, j);
                    }
                    result.setValue(i, j, sum);
                }
            }
            return result;
        }
    }
```

```
    package Multipliers;

    import Containers.*;

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

    public final class TapeMultiplier implements IMatricesMultiplier {
        private int poolCapacity;

        public TapeMultiplier(int poolCapacity) {
            this.poolCapacity = poolCapacity;
        }

        @Override
        public Result multiply(Matrix matrixA, Matrix matrixB) throws
ExecutionException, InterruptedException {
            int size = matrixA.getRowsNumber();
            Matrix transposedB = matrixB.getTransposedMatrix();
            Matrix resultMatrix = new Matrix(size, size, false);
            ExecutorService pool = Executors.newFixedThreadPool(poolCapacity);
            ArrayList<TapeMultiplierTask> tasks = new ArrayList<>();
            ArrayList<Future<Integer>> results = new ArrayList<>();

            for (int d = 0; d < matrixA.getColumnsNumber(); d++) {
                for (int i = 0; i < size; i++) {
                    tasks.add(new TapeMultiplierTask(matrixA.getRow(i),
transposedB.getRow((d + i) % size)));
                    results.add(pool.submit(tasks.get(tasks.size() - 1)));
                }
            }
            pool.shutdown();
            for (int d = 0; d < size; d++) {
                for (int i = 0; i < size; i++) {
                    resultMatrix.setValue(i, (d + i) % size, results.get(d * size +
i).get());
                }
            }
            return new Result(resultMatrix);
        }

        @Override
        public String getName() {
            return "Tape";
        }

        @Override
        public int getPoolCapacity() {
            return poolCapacity;
        }

        @Override
        public void setPoolCapacity(int capacity) {
            this.poolCapacity = capacity;
        }
```

```
        @Override
        public boolean isParallelAlgorithm() {
            return true;
        }

        private record TapeMultiplierTask(int[] row, int[] col) implements
Callable<Integer> {
            @Override
            public Integer call() {
                int sum = 0;
                for (int i = 0; i < row.length; i++) {
                    sum += row[i] * col[i];
                }
                return sum;
            }
        }

    }
```

```
    package Multipliers;

    import Containers.*;

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

    public final class FoxMultiplier implements IMatricesMultiplier {
        private int poolCapacity;

        public FoxMultiplier(int poolCapacity) {
            this.poolCapacity = poolCapacity;
        }

        @Override
        public Result multiply(Matrix matrixA, Matrix matrixB) throws
ExecutionException, InterruptedException {
            int splitSize = (int) Math.sqrt(poolCapacity - 1) + 1;
            Matrix[][] matricesA = IMatricesMultiplier.getSplitMatrices(matrixA,
splitSize);
            Matrix[][] matricesB = IMatricesMultiplier.getSplitMatrices(matrixB,
splitSize);
            Matrix[][] resultMatrices = new Matrix[splitSize][splitSize];
            for (int blockI = 0; blockI < splitSize; blockI++) {
                for (int blockJ = 0; blockJ < splitSize; blockJ++) {
                    resultMatrices[blockI][blockJ] = new
Matrix(matricesA[blockI][blockJ].getRowsNumber(),
matricesB[blockI][blockJ].getColumnsNumber(), false);
                }
            }

            ExecutorService pool = Executors.newFixedThreadPool(poolCapacity);
            ArrayList<Future<Matrix>> futureResults = new ArrayList<>();

            for (int s = 0; s < splitSize; s++) {
                for (int i = 0; i < splitSize; i++) {
                    for (int j = 0; j < splitSize; j++) {
                        futureResults.add(pool.submit(new
FoxMultiplierTask(matricesA[i][s], matricesB[s][j])));
                    }
                }
                for (int i = 0; i < splitSize; i++) {
                    for (int j = 0; j < splitSize; j++) {
                        resultMatrices[i][j].addMatrix(futureResults.get(i * splitSize +
j).get());
                    }
                }
                futureResults.clear();
            }
            pool.shutdown();
            return new Result(IMatricesMultiplier.combineMatrices(resultMatrices));
        }

        @Override
        public String getName() {
```

```
            return "Fox";
        }

        @Override
        public int getPoolCapacity() {
            return poolCapacity;
        }

        @Override
        public void setPoolCapacity(int capacity) {
            this.poolCapacity = capacity;
        }

        @Override
        public boolean isParallelAlgorithm() {
            return true;
        }

        private record FoxMultiplierTask(Matrix matrixA, Matrix matrixB) implements
Callable<Matrix> {
            @Override
            public Matrix call() {
                return new StandardMultiplier().multiply(matrixA, matrixB).getMatrix();
            }
        }
    }
```

```
    import Containers.*;
    import Multipliers.FoxMultiplier;
    import Multipliers.IMatricesMultiplier;
    import Multipliers.StandardMultiplier;
    import Multipliers.TapeMultiplier;

    import java.util.concurrent.ExecutionException;

    public class Main {
        public static void main(String[] args) throws ExecutionException,
InterruptedException {
            int[][] dataA = {
                    {1, 2, 3, 4},
                    {5, 6, 7, 8},
                    {9, 10, 11, 12},
                    {13, 14, 15, 16}
            };

            int[][] dataB = {
                    {1, 5, 9, 13},
                    {2, 6, 10, 14},
                    {3, 7, 11, 15},
                    {4, 8, 12, 16}
            };
            Matrix smallA = new Matrix(dataA);
            Matrix smallB = new Matrix(dataB);
            Matrix bigA = new Matrix(2000, 2000, true);
            Matrix bigB = new Matrix(2000, 2000, true);

            //checkAlorithmsResults(smallA, smallB);
            testAlgorithm(new StandardMultiplier());
            testAlgorithm(new TapeMultiplier(1));
            testAlgorithm(new FoxMultiplier(1));
        }

        private static void testAlgorithm(IMatricesMultiplier multiplier) throws
ExecutionException, InterruptedException {
            System.out.println("     " + multiplier.getName());
            for (int size = 1000; size <= 2500; size += 500) {
                System.out.printf("     MATRIX SIZE: %-5d\n", size);
                Matrix matrixA = new Matrix(size, size, true);
                Matrix matrixB = new Matrix(size, size, true);
                if (multiplier.isParallelAlgorithm()) {
                    for (int capacity = 1; capacity <= 10; capacity++) {
                        System.out.printf("     THREADS: %-5d ", capacity);
                        multiplier.setPoolCapacity(capacity);
                        repeatMultiplication(multiplier, matrixA, matrixB);
                    }
```

```
                } else {
                    repeatMultiplication(multiplier, matrixA, matrixB);
                }
                System.out.println();
            }
        }

        private static void repeatMultiplication(IMatricesMultiplier multiplier, Matrix
matrixA, Matrix matrixB) throws ExecutionException, InterruptedException {
            Clock.averageOn();
            for (int i = 0; i < 10; i++) {
                Clock.start();
                multiplier.multiply(matrixA, matrixB);
                Clock.stop();
            }
            System.out.printf("     AVERAGE TIME: %-5d\n", Clock.getAverage());
            Clock.averageOff();
        }

        private static void checkAlorithmsResults(Matrix smallA, Matrix smallB) throws
ExecutionException, InterruptedException {
            Result standardResult = new StandardMultiplier().multiply(smallA, smallB);
            Result tapeResult = new TapeMultiplier(1).multiply(smallA, smallB);
            Result foxResult = new FoxMultiplier(1).multiply(smallA, smallB);
            if (Matrix.compareMatrices(standardResult.getMatrix(),
tapeResult.getMatrix()) &&
                    Matrix.compareMatrices(standardResult.getMatrix(),
foxResult.getMatrix())) {
                System.out.println("Standard, Tape and Fox result matrices are equal");
            } else {
                System.out.println("Standard, Tape and Fox result matrices are not
equal");
            }
            standardResult.print();
            tapeResult.print();
            foxResult.print();
        }
    }
```

```
    import java.util.ArrayList;

    public class Clock {
        static private long startTime;
        static private final ArrayList<Long> averageTimes = new ArrayList<>();
        static private int times = 0;

        static void start() {
            times++;
            startTime = System.currentTimeMillis();
        }

        static void stop() {
            long stopTime = System.currentTimeMillis();
            if (averageTimes.size() > 0) {
                averageTimes.set(averageTimes.size() - 1,
averageTimes.get(averageTimes.size() - 1) + (stopTime - startTime));
            }
        }

        static void averageOn() {
            averageTimes.add(0L);
        }

        static void averageOff() {
            averageTimes.clear();
            times = 0;
        }

        static Long getAverage() {
            return averageTimes.get(averageTimes.size() - 1) / times;
        }
    }
```