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 № 4 з дисципліни

«Технології паралельних та розподілених обчислень»

«Розробка паралельних програм з використанням пулів потоків, екзекьюторів та ForkJoinFramework»

Виконав(ла) ІП-01 Галько М.В.

(шифр, прізвище, ім'я,

Перевірив(ла) Стеценко І. В.

(прізвище, ім'я, по

Київ 2022

1. **Завдання:**

2. **Побудуйте алгоритм статистичного аналізу тексту та визначте характеристики випадкової величини «довжина слова в символах» з використанням ForkJoinFramework. 20 балів. Дослідіть побудований алгоритм аналізу текстових документів на ефективність експериментально. 10 балів.**

3. **Реалізуйте один з алгоритмів комп'ютерного практикуму 2 або 3 з використанням ForkJoinFramework та визначте прискорення, яке отримане за рахунок використання ForkJoinFramework. 20 балів.**

4. **Розробіть та реалізуйте алгоритм пошуку спільних слів в текстових документах з використанням ForkJoinFramework. 20 балів.**

5. **Розробіть та реалізуйте алгоритм пошуку текстових документів, які відповідають заданим ключовим словам (належать до області «Інформаційні технології»), з використанням ForkJoinFramework. 30 балів.**

2. **Хід роботи. Завдання 1, 3, 4**

Для виконання лабораторної роботи було прийнято рішення розділити усю роботу на 2 проекти. Перший з низ реалізує завдання 1, 3 та 4, а другий – 2.

Отже, щодо першого проекту, для нього стало необхідним створити структуру класів, що виконає аналіз структури. Тому визначимо як буде реалізовано кожну задачу:

- Завдання 1. Алгоритм статистичного аналізу тексту реалізуємо у головному класі TextFileAnalyser. Для визначення характеристик випадкової величини "довжина слова в символах" використаємо клас WordsLengthsTask, який є

підкласом RecursiveTask. Цей алгоритм виконає аналіз текстових документів у паралельних потоках, використовуючи ForkJoinFramework. Результати аналізу, такі як кількість слів, середня довжина слова та максимальна довжина слова, виведемо на консоль.

- Завдання 3. Алгоритм пошуку спільних слів у текстових документах реалізуємо у класі CommonWordsTask, який також є підкласом RecursiveTask. Цей алгоритм виконає пошук унікальних слів у текстових документах та знайде спільні слова серед них. Результати пошуку спільних слів та їх кількість виведемо на консоль.
- Завдання 4. Алгоритм пошуку текстових документів, що відповідають заданим ключовим словам, реалізуємо у класі KeyWordsInFileTask, також підкласі RecursiveTask. Цей алгоритм виконує пошук заданих ключових слів у текстових документах та підраховує кількість документів, в яких присутні всі ключові слова, деякі з них або жодного (виводяться на консоль).

Для тестування у класі Main було продемонстровано використання реалізованих алгоритмів. Зокрема, для кожної кількості потоків зі списку threadsCount створювався екземпляр TextFileAnalyser, і виконувалися методи для отримання характеристик документів та пошуку слів. У якості тексту був обраний твір «The Great Gatsby» і розділений на різні файли. Структури файлів і папок можна побачити на рис. 2.1.

Виконавши програму, отримаємо вивід у консоль виду рис. 2.2. На основі даних з неї сформуємо таблицю із результатами (таблиця 2.1-3) і представляємо залежність часу виконання роботи програми від кількості потоків у графічному вигляді на рис. 2.3.

Використовувалися наступні додаткові дані:

- keyWords; "Gatsby", "Daisy", "Tom", "Jordan", "Nick", "Myrtle", "George", "Wilson";
- Варіанти кількості потоків: 2, 4, 6, 8, 9.

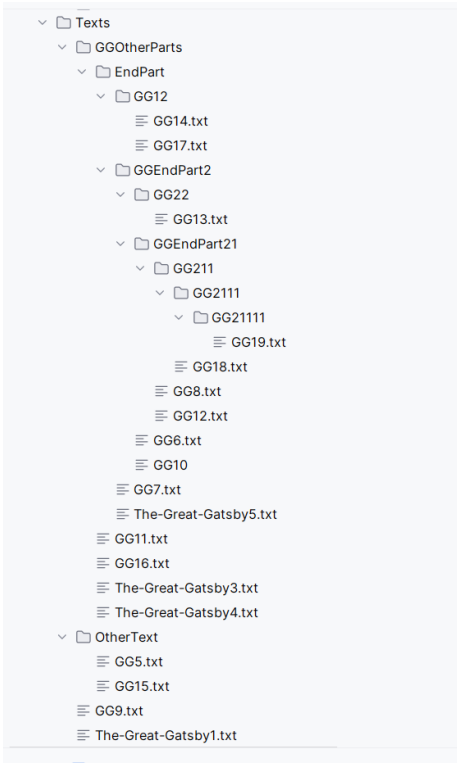


Рис. 2.1 – Структура папки із файлами тексту

```
-------------------------------------------------------
Threads count: 2
Count: 53009
Mean length: 3
Max length: 19
WordsLength Time: 226 ms

Common words: [, a, i, the, with, have, all, and, was, her, you, at, be, she, for, if, in, it, that, of, then, to, this, from]
Count: 24
CommonWords Time: 118 ms

Files and key word count:
NO : 0
MID: 19
ALL: 1
KeyWords Time: 44 ms

-------------------------------------------------------
Threads count: 4
Count: 53009
Mean length: 3
Max length: 19
WordsLength Time: 72 ms

Common words: [, a, i, the, with, have, all, and, was, her, you, at, be, she, for, if, in, it, that, of, then, to, this, from]
Count: 24
CommonWords Time: 53 ms

Files and key word count:
NO : 0
MID: 19
ALL: 1
KeyWords Time: 32 ms
```

Рис. 2.2 – Вивід програмного забезпечення

Таблиця 2.1 – Результати з аналізу текстів

| Words | Mean length | Max length | Common words | Key words NO | Key words NOT ALL | Key words ALL |
|---|---|---|---|---|---|---|
| 53009 | 3 | 19 | 24 | 0 | 19 | 1 |

Спільні слова: "a", "i", "the", "with", "have", "all", "and", "was", "her", "you", "at", "be", "she", "for", "if", "in", "it", "that", "of", "then", "to", "this", "from".

Таблиця 2.2 – Результати часу виконання для різної кількості потоків

| Кількість потоків | 2 | 4 | 6 | 8 | 9 |
|---|---|---|---|---|---|
| Час виконання (ms) | 226 | 72 | 16 | 32 | 16 |

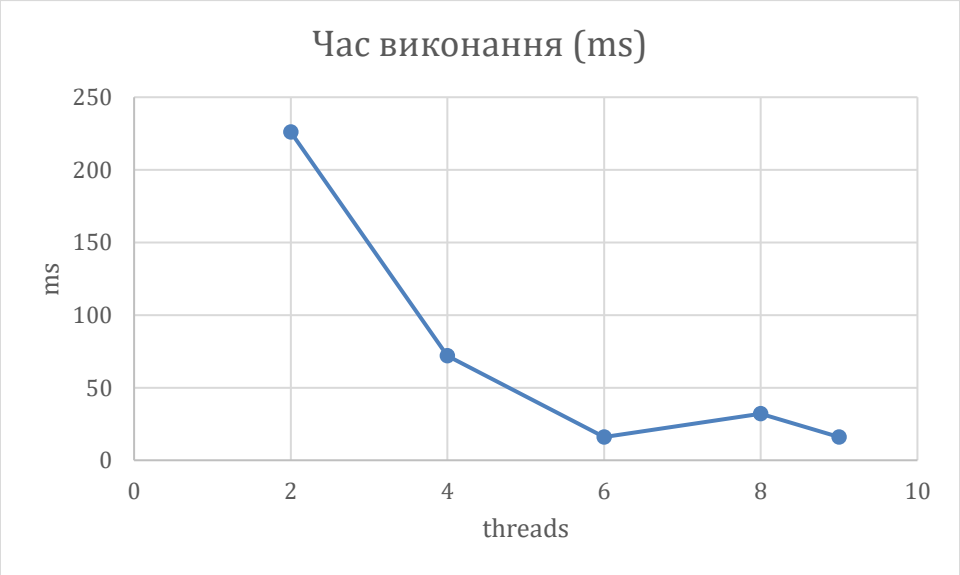


Рис.2.3 – Графік залежності часу виконання роботи від кількості потоків

Бачимо, що використання більшої кількості потоків (6, 8, 9) призводить до скорочення часу виконання аналізу тексту та пошуку слів, порівняно з меншою кількістю потоків (2, 4).

Час виконання завдання зменшується при збільшенні кількості потоків до певної межі, після чого подальше збільшення кількості потоків не призводить до значних змін в часі виконання. Також програма успішно знаходить спільні слова та виконує пошук за ключовими словами у текстових документах.

## 3. Хід роботи. Завдання 2

У другій роботі реалізуємо алгоритм Фокса з використанням ForkJoinFramework. Для цього відтворюємо код класів з 2-ої лабораторної: `Main`, `Matrix`, `Result`, `StandardMultiplier`, `IMatricesMultiplier`, `FoxMultiplier`; та додаткові: `FoxMultiplierTask`, `FoxForkJoinMultiplier`.

У класі `Main` виконаємо тестування обох алгоритмів (Fox та FoxForkJoin) з різними розмірами матриць та кількістю потоків. Вхідні дані:

- Розміри матриць: 500, 1000, 1500, 2000, 2500, 3000;
- Кількість потоків: 2, 4, 8, 9;

Клас `Matrix` використовується для представлення матриць та виконання операцій з ними, таких як множення. А клас `Result` зберігає результати множення матриць.

Інтерфейс `IMatricesMultiplier` описує метод `multiply`, який потрібно реалізувати для різних алгоритмів множення матриць.

Клас `StandardMultiplier` реалізує стандартний алгоритм множення матриць.

Клас `FoxMultiplier` реалізує множення матриць алгоритм Фокса.

Клас `FoxMultiplierTask` представляє завдання, яке буде виконуватися у паралельних потоках для алгоритму Фокса.

Клас `FoxForkJoinMultiplier` використовує `ForkJoinPool` для множення матриць з використанням `ForkJoinFramework`.

## 4. Результати завдання 2

Програма виконує вивід у консоль як на рис. 4.1.

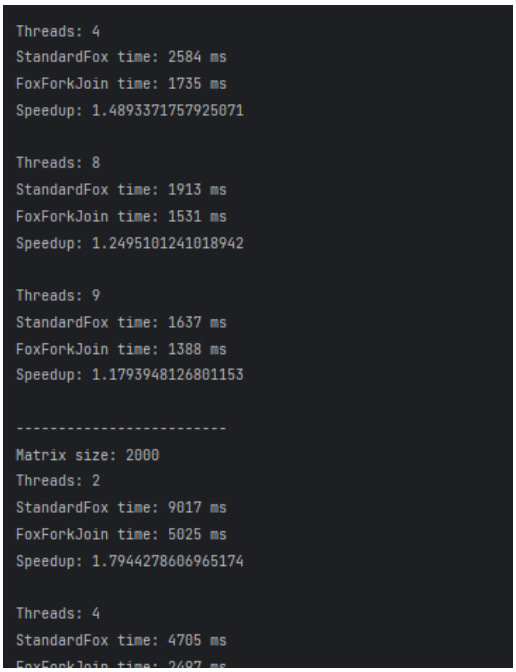


Рис. 2.1 – Вивід ПЗ завдання 2

Зібравши усі дані разом, маємо змогу представити їх результати у вигляді таблиці (рис. 2.2). Далі продемонструємо результати 2-х алгоритмів на графіках окремо на рис. 2.3 для Фокса та на рис. 2.4 для FoxForkJoin.

| Matrix size | Threads | StandardFox time (ms) | FoxForkJoin time (ms) | Speedup |
|---|---|---|---|---|
| 500 | 2 | 158 | 164 | 0.963 |
| 500 | 4 | 85 | 69 | 1.232 |
| 500 | 8 | 59 | 57 | 1.035 |
| 500 | 9 | 60 | 53 | 1.132 |
| 1000 | 2 | 853 | 692 | 1.233 |
| 1000 | 4 | 530 | 629 | 0.843 |
| 1000 | 8 | 614 | 584 | 1.051 |
| 1000 | 9 | 450 | 506 | 0.889 |
| 1500 | 2 | 3926 | 3213 | 1.222 |
| 1500 | 4 | 2584 | 1735 | 1.489 |
| 1500 | 8 | 1913 | 1531 | 1.25 |
| 1500 | 9 | 1637 | 1388 | 1.179 |
| 2000 | 2 | 9017 | 5025 | 1.794 |
| 2000 | 4 | 4705 | 2497 | 1.884 |
| 2000 | 8 | 3231 | 2263 | 1.428 |
| 2000 | 9 | 2814 | 2224 | 1.265 |
| 2500 | 2 | 19699 | 10409 | 1.892 |
| 2500 | 4 | 12550 | 4959 | 2.531 |
| 2500 | 8 | 9316 | 4679 | 1.991 |
| 2500 | 9 | 8382 | 5019 | 1.67 |
| 3000 | 2 | 68016 | 13742 | 4.949 |
| 3000 | 4 | 41413 | 8312 | 4.982 |
| 3000 | 8 | 26286 | 7036 | 3.736 |
| 3000 | 9 | 24479 | 7158 | 3.42 |

Рис. 2.2 – Результат роботи алгоритмів

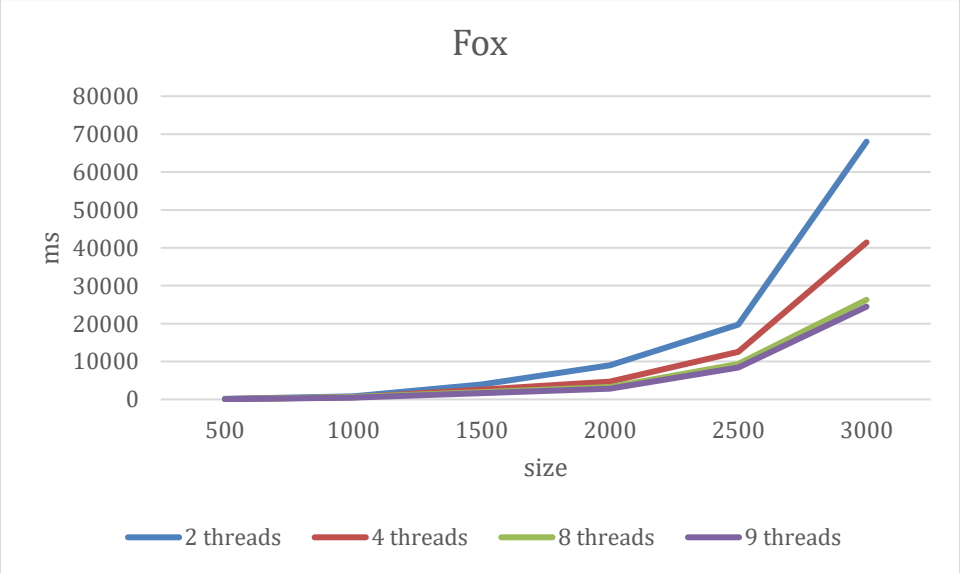


Рис.2.3 – Залежність часу роботи від розміру та кількості потоків для Fox

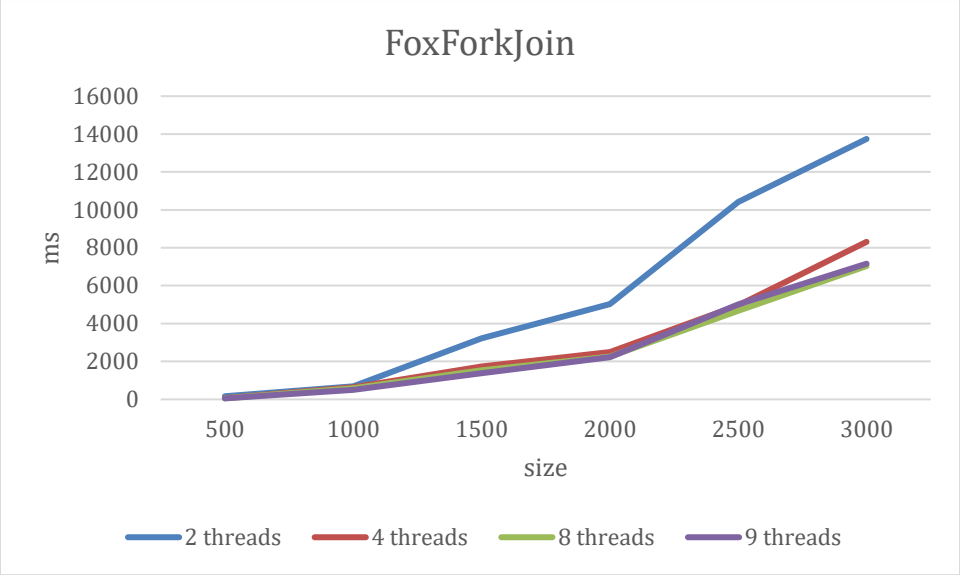


Рис.2.4 – Залежність часу роботи від розміру та кількості потоків для FoxForkJoin

Прорахувавши прискорення, побудуємо графік прискорення FoxForkJoin (рис. 2.5).

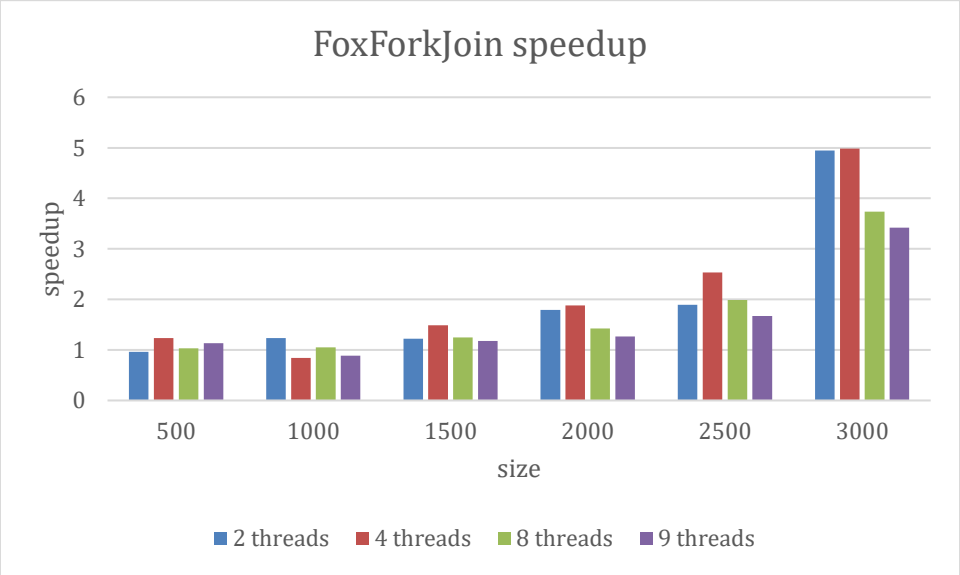


Рис.2.5 – Графік прискорення

Проаналізувавши результати таблиць та графіків можемо зробити наступні висновки:

1. Збільшення кількості потоків зазвичай призводить до збільшення прискорення. У більшості випадків більша кількість потоків дозволяє досягти більшого прискорення. Однак, для деяких комбінацій розміру матриці та кількості потоків, прискорення може бути незначним або навіть меншим за однопотоковий варіант.

2. Прискорення залежить від розміру матриці. Із збільшенням розміру, стає більш очевидним прискорення. Наприклад для 3000 значення стають у 2 рази більше за попередній тест на 2500, та досягає 5.

3. Прискорення залежить від кількості потоків. На рис. 2.5 видно, що найкращі результати досягаються при 4 потоках, і трохи гірші на 2-х. Але при маленьких матрицях (розмір до 1500) спостерігається все гірше прискорення при збільшуванні кількості потоків.

4. Алгоритм FoxForkJoin показує загалом кращі результати порівняно зі стандартним алгоритмом Fox. У більшості випадків FoxForkJoin забезпечує більше прискорення, що свідчить про його ефективнішу роботу при використанні багатопотокового середовища.

## 5. Код завдань 1, 3, 4

```
import java.io.File;
import java.util.HashSet;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        File theGreatGatsbyPath = new File("Texts");
        List<String> keyWords = List.of("Gatsby", "Daisy", "Tom", "Jordan",
"Nick", "Myrtle", "George", "Wilson");
        int[] threadsCount = {2, 4, 6, 8, 9};

        System.out.println(theGreatGatsbyPath.getAbsolutePath());
        for (var count : threadsCount) {
            System.out.println("----------------------------------------
---------------");
            System.out.println("Threads count: " + count);
            TextFileAnalyser theGreatGatsbyAnalyser = new
TextFileAnalyser(count, theGreatGatsbyPath);
            testGetWordsLengths(theGreatGatsbyAnalyser);
            testGetCommonWords(theGreatGatsbyAnalyser);
            testKeyWordsInFiles(theGreatGatsbyAnalyser, keyWords);
        }
    }

    private static void testGetWordsLengths(TextFileAnalyser
textFileAnalyser) {
        long startTime = System.currentTimeMillis();
        textFileAnalyser.wordsLengthForkJoin();
        long totalTime = System.currentTimeMillis() - startTime;

        System.out.println("Count: " + textFileAnalyser.getWordsCount());
        System.out.println("Mean length: " +
textFileAnalyser.getMeanLength());
        System.out.println("Max length: " +
textFileAnalyser.getMaxLength());
        System.out.println("WordsLength Time: " + totalTime + " ms");
        System.out.println();
    }

    private static void testGetCommonWords(TextFileAnalyser
textFileAnalyser) {
        long startTime = System.currentTimeMillis();
        HashSet<String> commonWords =
textFileAnalyser.getCommonWordsForkJoin();
        long time = System.currentTimeMillis() - startTime;

        System.out.println("Common words: " + commonWords);
        System.out.println("Count: " + commonWords.size());
        System.out.println("CommonWords Time: " + time + " ms");
        System.out.println();
    }

    private static void testKeyWordsInFiles(TextFileAnalyser
textFileAnalyser, List<String> keyWords) {
        long startTime = System.currentTimeMillis();
        textFileAnalyser.getKeyWordsInFiles(keyWords);
        long time = System.currentTimeMillis() - startTime;
```

```
        System.out.println("Files and key word count: ");
        System.out.println("NO : " +
textFileAnalyser.getCountNoKeyWords());
        System.out.println("MID: " +
textFileAnalyser.getCountNotAllKeyWords());
        System.out.println("ALL: " +
textFileAnalyser.getCountAllKeyWords());
        System.out.println("KeyWords Time: " + time + " ms");
        System.out.println();
    }
}
```

```
import java.io.File;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.concurrent.ForkJoinPool;

public class TextFileAnalyser {
    private final ForkJoinPool forkJoinPool;
    private final File filePath;
    private List<Integer> wordsLength = new ArrayList<>();
    private HashMap<String, List<String>> keyWordsInFiles = new
HashMap<>();
    private List<String> keyWords = new ArrayList<>();
    public TextFileAnalyser(int threadsCount, File filePath) {
        forkJoinPool = new ForkJoinPool(threadsCount);
        this.filePath = filePath;
    }

    public void wordsLengthForkJoin() {
        wordsLength = forkJoinPool.invoke(new WordsLengthsTask(filePath));
    }

    public HashSet<String> getCommonWordsForkJoin() {
        return forkJoinPool.invoke(new CommonWordsTask(filePath));
    }

    public HashMap<String, List<String>> getKeyWordsInFiles(List<String>
keyWords) {
        this.keyWords = keyWords;
        List<String> keyWordsLowerCase = new ArrayList<>();
        for (String word : keyWords) {
            keyWordsLowerCase.add(word.toLowerCase());
        }
        return keyWordsInFiles = forkJoinPool.invoke(new
KeyWordsInFileTask(filePath, keyWordsLowerCase));
    }

    public Integer getMeanLength() {
        if (wordsLength.size() == 0) {
            return 0;
        }
        return wordsLength.stream().reduce(0, Integer::sum) /
wordsLength.size();
    }

    public Integer getMaxLength() {
```

```
        if (wordsLength.size() == 0) {
            return 0;
        }
        return wordsLength.stream().reduce(0, Integer::max);
    }

    public Integer getWordsCount() {
        return wordsLength.size();
    }

    public int getCountNoKeyWords() {
        int count = 0;
        for (var file : keyWordsInFiles.entrySet()) {
            if (file.getValue().size() == 0) {
                count++;
            }
        }
        return count;
    }

    public int getCountAllKeyWords() {
        int count = 0;
        for (var file : keyWordsInFiles.entrySet()) {
            if (file.getValue().size() == keyWords.size()) {
                count++;
            }
        }
        return count;
    }

    public int getCountNotAllKeyWords() {
        int count = 0;
        for (var file : keyWordsInFiles.entrySet()) {
            if (file.getValue().size() < keyWords.size() &&
file.getValue().size() != 0) {
                count++;
            }
        }
        return count;
    }

}
```

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Objects;
import java.util.concurrent.RecursiveTask;

class WordsLengthsTask extends RecursiveTask<List<Integer>> {
    private final File file;

    WordsLengthsTask(File file) {
        this.file = file;
    }
```

```
    @Override
    protected List<Integer> compute() {
        if (!file.isDirectory()) {
            return getAllLengthsInFile(file);
        }

        ArrayList<Integer> wordsLength = new ArrayList<>();
        List<RecursiveTask<List<Integer>>> tasks = new LinkedList<>();

        for (File entry : Objects.requireNonNull(file.listFiles())) {
            WordsLengthsTask task = new WordsLengthsTask(entry);
            tasks.add(task);
            task.fork();
        }

        for (RecursiveTask<List<Integer>> task : tasks) {
            wordsLength.addAll(task.join());
        }

        return wordsLength;
    }

    private static List<Integer> getAllLengthsInFile(File file) {
        List<Integer> wordLengths = new ArrayList<>();

        try (BufferedReader reader = new BufferedReader(new
FileReader(file))) {
            String line = reader.readLine();
            while (line != null) {
                for (String word : getAllWordsInLine(line)) {
                    wordLengths.add(word.length());
                }
                line = reader.readLine();
            }
        } catch (Exception ignored) {
        }

        return wordLengths;
    }

    private static String[] getAllWordsInLine(String line) {
        return line.trim().split("(\\s|\\p{Punct})+");
    }
}
```

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Objects;
import java.util.concurrent.RecursiveTask;

class CommonWordsTask extends RecursiveTask<HashSet<String>> {
    private final File file;

    CommonWordsTask(File file) {
        this.file = file;
```

```
    }

    @Override
    protected HashSet<String> compute() {
        if (!file.isDirectory()) {
            return getUniqueWordsInFile(file);
        }
        HashSet<String> commonWords;
        List<RecursiveTask<HashSet<String>>> tasks = new ArrayList<>();

        for (File entry : Objects.requireNonNull(file.listFiles())) {
            CommonWordsTask task = new CommonWordsTask(entry);
            tasks.add(task);
            task.fork();
        }

        commonWords = tasks.get(0).join();
        for (RecursiveTask<HashSet<String>> task : tasks) {
            commonWords.retainAll(task.join());
        }

        return commonWords;
    }

    private static HashSet<String> getUniqueWordsInFile(File file) {
        HashSet<String> uniqueWords = new HashSet<>();
        try (BufferedReader reader = new BufferedReader(new
FileReader(file))) {
            String line = reader.readLine();
            while (line != null) {
                String[] words = line.trim().split("(\\s|\\p{Punct})+");
                for (String word : words) {
                    uniqueWords.add(word.toLowerCase());
                }
                line = reader.readLine();
            }
        } catch (Exception ignored) {
        }
        return uniqueWords;
    }
}
```

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.util.*;
import java.util.concurrent.RecursiveTask;

class KeyWordsInFileTask extends RecursiveTask<HashMap<String,
List<String>>> {
    private final File file;
    private final List<String> keyWords;

    KeyWordsInFileTask(File file, List<String> keyWords) {
        this.file = file;
        this.keyWords = keyWords;
    }

    @Override
```

```
    protected HashMap<String, List<String>> compute() {
        if (!file.isDirectory()) {
            return checkKeyWords(file, keyWords);
        }
        HashMap<String, List<String>> keyWordsFiles = new HashMap<>();
        List<RecursiveTask<HashMap<String, List<String>>>> tasks = new
ArrayList<>();

        for (File entry : Objects.requireNonNull(file.listFiles())) {
            KeyWordsInFileTask task = new KeyWordsInFileTask(entry,
keyWords);
            tasks.add(task);
            task.fork();
        }

        for (RecursiveTask<HashMap<String, List<String>>> task : tasks) {
            keyWordsFiles.putAll(task.join());
        }

        return keyWordsFiles;
    }

    private static HashMap<String, List<String>> checkKeyWords(File file,
List<String> keyWords) {
        HashSet<String> uniqueWords = new HashSet<>();
        String fileName = file.getPath() + file.getName();

        try (BufferedReader reader = new BufferedReader(new
FileReader(file))) {
            String line = reader.readLine();
            while (line != null) {
                String[] words = line.trim().split("(\\s|\\p{Punct})+");
                for (String word : words) {
                    uniqueWords.add(word.toLowerCase());
                }
                line = reader.readLine();
            }
        } catch (Exception ignored) {
        }

        List<String> matchedWords = new ArrayList<>(keyWords);
        matchedWords.retainAll(uniqueWords);

        HashMap<String, List<String>> fileMatches = new HashMap<>();
        fileMatches.put(file.getName(), matchedWords);

        return fileMatches;
    }
}
```

## 6. Код завдання 2:

```java
import java.util.concurrent.ExecutionException;

public class Main {
    public static void main(String[] args) throws ExecutionException, InterruptedException {
        int[] matrixSizes = {500, 1000, 1500, 2000, 2500, 3000};
        int[] threadsCounts = {2, 4, 8, 9};

        testAlgorithmsSpeed(matrixSizes, threadsCounts);
    }

    private static void testAlgorithmsSpeed(int[] matrixSizes, int[] threadsCounts) throws
ExecutionException, InterruptedException {
        for (int matrixSize : matrixSizes) {
            Matrix matrixA = new Matrix(matrixSize, matrixSize, true);
            Matrix matrixB = new Matrix(matrixSize, matrixSize, true);
            System.out.println("-------------------------");
            System.out.println("Matrix size: " + matrixSize);
            for (int thread : threadsCounts) {
                FoxForkJoinMultiplier foxAlgorithmForkJoin = new FoxForkJoinMultiplier(thread);
                long foxSpeed = checkAlgorithmSpeed(new FoxMultiplier(thread), matrixA, matrixB, 5);
                long ffjSpeed = checkAlgorithmSpeed(foxAlgorithmForkJoin, matrixA, matrixB, 5);
                System.out.println("Threads: " + thread);
                System.out.println("StandardFox time: " + foxSpeed + " ms");
                System.out.println("FoxForkJoin time: " + ffjSpeed + " ms");
                System.out.println("Speedup: " + (double) foxSpeed / ffjSpeed);
                System.out.println();
            }
        }
    }

    static long checkAlgorithmSpeed(IMatricesMultiplier multiplier, Matrix matrixA, Matrix matrixB,
int iterations) throws ExecutionException, InterruptedException {
        long sum = 0;
        for (int i = 0; i < iterations; i++) {
            long startTime = System.currentTimeMillis();
            multiplier.multiply(matrixA, matrixB);
            long endTime = System.currentTimeMillis();
            sum += endTime - startTime;
        }
        return sum / iterations;
    }
}
```

```java
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 Matrix multiply(Matrix matrix2) {
        int[][] result = new int[data.length][matrix2.getColumnsNumber()];
        for (int i = 0; i < data.length; i++) {
            for (int j = 0; j < matrix2.getColumnsNumber(); j++) {
                for (int k = 0; k < data[0].length; k++) {
                    result[i][j] += data[i][k] * matrix2.getValue(k, j);
                }
            }
        }
        return new Matrix(result);
    }
```

```
    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;
    }
}
```

```
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();
    }
}
```

```
public class StandardMultiplier implements IMatricesMultiplier {
    @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;
    }
}
```

```
import java.util.concurrent.ExecutionException;

public interface IMatricesMultiplier {
    Result multiply(Matrix matrixA, Matrix matrixB) throws ExecutionException, InterruptedException;

    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();
    }
}
```

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

public final class FoxMultiplier implements IMatricesMultiplier {
    private final 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));
    }

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

```
import java.util.concurrent.ForkJoinPool;

public class FoxForkJoinMultiplier implements IMatricesMultiplier {
    private final ForkJoinPool forkJoinPool;

    public FoxForkJoinMultiplier(int countThreads) {
        forkJoinPool = new ForkJoinPool(countThreads);
    }

    @Override
    public Result multiply(Matrix matrixA, Matrix matrixB) {
        return new Result(forkJoinPool.invoke(new FoxMultiplierTask(matrixA, matrixB)));
    }
}
```

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.RecursiveTask;

public class FoxMultiplierTask extends RecursiveTask<Matrix> {
    private final Matrix matrixA;
    private final Matrix matrixB;

    public FoxMultiplierTask(Matrix matrixA, Matrix matrixB) {
        this.matrixA = matrixA;
        this.matrixB = matrixB;
    }

    @Override
    public Matrix compute() {
        int matrixSizeLimit = 100;
        if (matrixA.getColumnsNumber() <= matrixSizeLimit) {
            return matrixA.multiply(matrixB);
        }
        int splitSize = 2;
        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);
            }
        }


        for (int s = 0; s < splitSize; s++) {
            List<FoxMultiplierTask> tasks = new ArrayList<>();
            List<Matrix> calculatedSubBlocks = new ArrayList<>();

            for (int i = 0; i < splitSize; i++) {
                for (int j = 0; j < splitSize; j++) {
                    var task = new FoxMultiplierTask(
                            matricesA[i][s],
                            matricesB[s][j]);

                    tasks.add(task);
```

```
                    task.fork();
                }
            }

            for (var task : tasks) {
                var subMatrix = task.join();
                calculatedSubBlocks.add(subMatrix);
            }

            for (int i = 0; i < splitSize; i++) {
                for (int j = 0; j < splitSize; j++) {
                    resultMatrices[i][j].addMatrix(calculatedSubBlocks.get(i * splitSize + j));
                }
            }
        }

        return IMatricesMultiplier.combineMatrices(resultMatrices);
    }
}
```