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 Корнієнко В.С.* _____
(шифр, прізвище, ім'я, по батькові)

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

Київ 2023

Завдання:

1. 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.**
2. Реалізуйте один з алгоритмів комп'ютерного практикуму 2 або 3 з використанням ForkJoinFramework та визначте прискорення, яке отримане за рахунок використання ForkJoinFramework. **20 балів.**
3. Розробіть та реалізуйте алгоритм пошуку спільних слів в текстових документах з використанням ForkJoinFramework. **20 балів.**
4. Розробіть та реалізуйте алгоритм пошуку текстових документів, які відповідають заданим ключовим словам (належать до області «Інформаційні технології»), з використанням ForkJoinFramework. **30 балів.**

Хід роботи:

1. **Побудуйте алгоритм статистичного аналізу тексту та визначте характеристики випадкової величини «довжина слова в символах» з використанням ForkJoinFramework.**

Оскільки завдання 1, 3, 4 орієнтовані на роботу з файловою системою, а саме на роботу з текстовими файлами та папками, створимо класи TextFile та Folder відповідно.

Клас Folder містить в собі всі вкладені папки та текстові файли. Також для більшої зручності користування до класу була додана статична фабрика, яка створює Folder на основі аргументу File бібліотеки java.io.

Клас має наступну структуру:

```
public class Folder {
    2 usages
    private final List<Folder> subFolders;
    2 usages
    private final List<TextFile> textFiles;

    valerii.korniienko *
    public Folder(List<Folder> subFolders, List<TextFile> textFiles) {...}
    3 usages  valerii.korniienko *
    List<TextFile> getTextFiles() { return this.textFiles; }
    3 usages  valerii.korniienko
    List<Folder> getSubFolders() { return this.subFolders; }
```

Статична фабрика Folder:

```
public static Folder loadFromDirectory(File dir) throws IOException {
    List<TextFile> documents = new LinkedList<>();
    List<Folder> subFolders = new LinkedList<>();

    for (File entry : Objects.requireNonNull(dir.listFiles())) {
        if (entry.isDirectory()) {
            subFolders.add(Folder.loadFromDirectory(entry));
        } else {
            documents.add(TextFile.createFromFile(entry));
        }
    }
    return new Folder(subFolders, documents);
}
```

В свою чергу клас TextFile містить в собі всі лінії текстового файлу та його ім’я. Також, аналогічно класу Folder, даний клас має статичну фабрику

Клас TextFile має наступну структуру:

```
public class TextFile {
    2 usages
    private final List<String> lines;
    2 usages
    private final String name;

    valerii.korniienko
    public TextFile(List<String> lines, String name) {...}

    3 usages  valerii.korniienko
    public List<String> getLines() { return this.lines; }
    valerii.korniienko
    public String getName() { return this.name; }
```

Статична фабрика TextFile:

```
public static TextFile createFromFile(File file) throws IOException {
    String path = file.getPath() + file.getName();
    List<String> lines = new ArrayList<>();
    try (BufferedReader reader = new BufferedReader(new FileReader(file))) {
        String line = reader.readLine();
        while (line != null) {
            lines.add(line);
            line = reader.readLine();
        }
    }
    return new TextFile(lines, path);
}
```

Далі був створений клас для безпосередньо рахування довжин слів в символах. Даний клас має конструктор, що ініціалізує ForkJoinPool з певною кількістю потоків та метод, який запускає в роботу пошук по папкам FolderSearchTask(folder) extends RecursiveTask<List<Integer>>.

По суті даний метод додає в forkJoinPool нове завдання типу FolderSearchTask, яке в свою чергу буде також рекурсивно додавати завдання типу FolderSearchTask, TextFileSearchTask в залежності від вмісту папки, яку ми даємо в якості вхідного аргументу.

Клас WordCounter має наступний вигляд:

```java
public class WordCounter {
    2 usages
    private final ForkJoinPool forkJoinPool;

    1 usage  valerii.korniienko
    public WordCounter(int countThreads) { forkJoinPool = new ForkJoinPool(countThreads); }

    1 usage  valerii.korniienko
    public List<Integer> getAllWordLenghtsForkJoin(Folder folder) {
        return forkJoinPool.invoke(new FolderSearchTask(folder));
    }
}
```

Далі були створені описані вище класи FolderSearchTask та TextFileSearchTask, що наслідуються від RecursiveTask<List<Integer>>.

Клас FolderSearchTask має в собі єдиний метод compute(), який має наступну структуру:

Спочатку ми ініціалізуємо структури для зберігання кількості слів, рекурсивних завдань:

```java
@Override
protected List<Integer> compute() {
    ArrayList<Integer> wordLengths = new ArrayList<>();
    List<RecursiveTask<List<Integer>>> tasks = new LinkedList<>();
```

Далі ми перебираємо всі вкладені папки та створюємо завдання для кожної з них, рекурсивно запускаємо їх на паралельне виконання за допомогою методу fork()

```java
for (Folder subFolder : folder.getSubFolders()) {
    FolderSearchTask task = new FolderSearchTask(subFolder);
    tasks.add(task);
    task.fork();
}
```

Потім робимо аналогічну операцію для текстових файлів

```java
for (TextFile textFile : folder.getTextFiles()) {
    TextFileSearchTask task = new TextFileSearchTask(textFile);
    tasks.add(task);
    task.fork();
}
```

Наприкінці методу ми збираємо результати з усіх завдань та додаємо їх в наш список довжин слів, повертаємо результат

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

Клас TextFileSearchTask містить в собі метод compute(), який рахує кількості всіх слів в файлі та повертає їх. Варто зазначити що в цьому методі не відбувається паралельних рекурсивних викликів, оскільки операція рахування слів в файлі не потребує розпаралелювання

Метод compute() має наступний вигляд:

```
@Override
protected List<Integer> compute() {
    return getAllWordLengths(textFile);
}
```

```
private static List<Integer> getAllWordLengths(TextFile textFile) {
    List<Integer> wordLengths = new ArrayList<>();
    for (String line : textFile.getLines()) {
        for (String word : getAllWordsInLine(line)) {
            wordLengths.add(word.length());
        }
    }
    return wordLengths;
}
```

Приклад результату роботи програми:

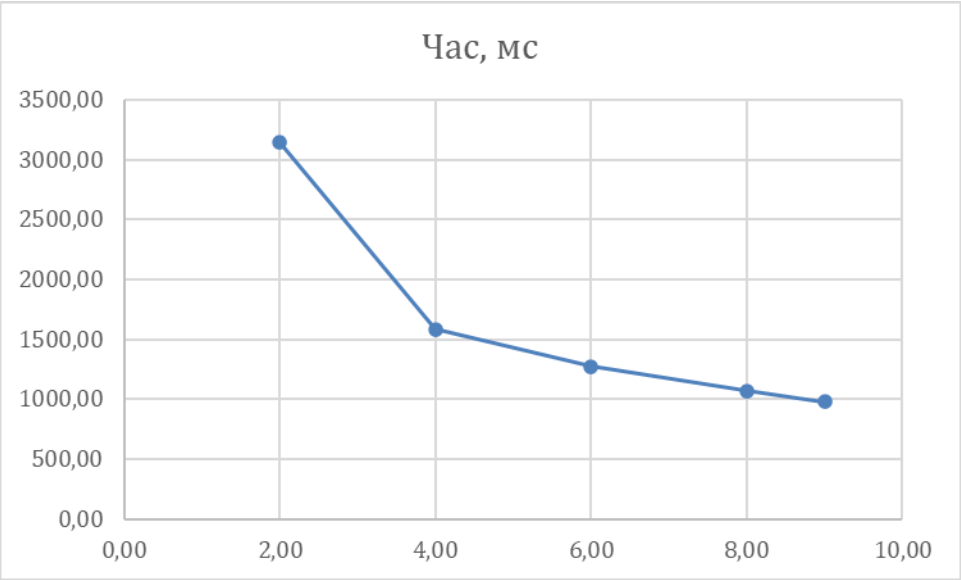
```
Count: 1903868
Mean length: 4.457752323165262
Dispersion: 7.785106346848348
Mean square deviation: 2.7901803430689474

Time: 573 ms

Process finished with exit code 0
```

1.1. Дослідіть побудований алгоритм аналізу текстових документів на ефективність експериментально.

Дослідимо ефективність алгоритму в залежності від кількості потоків. Всі тести будемо проводити на 1000 книгах та вкладеністю папок до 7. Всі файли разом мають приблизно 18_000_000 слів

Маємо наступні результати:

Кількість потоків	Час, мс
2	3143
4	1584
6	1275
8	1070
9	976



Як бачимо зі збільшенням кількості потоків, програма починає працювати значно швидше. Найбільший пік такого бусту припадає на збільшення з 2 до 4 потоків, оскільки це фактичне збільшення обчислюваного ресурсу, якого не вистачає, в 2 рази.

Консольний вивід програми:

```
------------------------
Threads count: 2
C:\Projects\ParrallelComputing\TPO_labs\lab4_1\TestFolder1
Count: 17928579
Mean length: 4.714728590592706
Dispersion: 9.189165813899752
Mean square deviation: 3.0313636888205533

Time: 3143 ms
------------------------
Threads count: 4
C:\Projects\ParrallelComputing\TPO_labs\lab4_1\TestFolder1
Count: 17928579
Mean length: 4.714728590592706
Dispersion: 9.189165813899752
Mean square deviation: 3.0313636888205533

Time: 1574 ms
------------------------
Threads count: 6
C:\Projects\ParrallelComputing\TPO_labs\lab4_1\TestFolder1
Count: 17928579
Mean length: 4.714728590592706
Dispersion: 9.189165813899752
Mean square deviation: 3.0313636888205533

Time: 1275 ms
------------------------
Threads count: 8
C:\Projects\ParrallelComputing\TPO_labs\lab4_1\TestFolder1
Count: 17928579
Mean length: 4.714728590592706
Dispersion: 9.189165813899752
Mean square deviation: 3.0313636888205533

Time: 1070 ms
------------------------
Threads count: 9
C:\Projects\ParrallelComputing\TPO_labs\lab4_1\TestFolder1
Count: 17928579
Mean length: 4.714728590592706
Dispersion: 9.189165813899752
Mean square deviation: 3.0313636888205533

Time: 976 ms

Process finished with exit code 0
```

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

В даній лабораторній роботі, в якості алгоритму для оптимізації з використанням ForkJoinFramework був використаний алгоритм Фокса з 2 лабораторної роботи. Даний алгоритм був обраний через його швидкодію, можливість модифікацій.

Для лабораторної роботи було створено 2 класа: FoxAlgorithmForkJoin та FoxAlgorithmTask.

FoxAlgorithmForkJoin клас має лише одну задачу: створення ForkJoinPool-а, його запуск з FoxAlgorithmTask для матриць, які необхідно перемножити. Клас має наступний вигляд:

```
public class FoxAlgorithmForkJoin implements IMatrixMultiplicationAlgorithm {
    2 usages
    private final ForkJoinPool forkJoinPool;

    1 usage  valerii.korniienko
    public FoxAlgorithmForkJoin(int countThreads) { forkJoinPool = new ForkJoinPool(countThreads); }
    6 usages  new *
    @Override
    public Result multiply(Matrix matrixA, Matrix matrixB) {
        long startTime = System.currentTimeMillis();

        return new Result(forkJoinPool.invoke(new FoxAlgorithmTask(matrixA, matrixB)),
                totalTime: System.currentTimeMillis() - startTime);
    }
}
```

В свою чергу клас FoxAlgorithmTask наслідується від RecursiveTask<Matrix>. В класі міститься конструктор та метод compute().

Метод compute() починається з умови виходу з рекурсії. В даному випадку умовою виходу з рекурсії є розмір матриці, менший за мінімальний заданий (matrixSizeLimit). При настанні цієї умови метод повертає результат послідовного множення цих матриць

```
@Override
public Matrix compute() {
    if (matrix1.getColumnsCount() <= matrixSizeLimit) {
        return matrix1.multiply(matrix2);
    }
```

Якщо умова виходу з рекурсії не справджується, ми починаємо дії, аналогічні зі звичайним алгоритмом Фокса, а саме розбиття початкових матриць на менші матриці, створення результуючих матриць та їх

ініціалізація. Єдиною різницею є те, що в даному випадку ми завжди розбиваємо на рівну кількість частин(в даному випадку splitFactor == 2)

```
Matrix[][] matrixM1 = FoxAlgorithm.splitMatrixIntoSmallerMatrices(matrix1, splitFactor);
Matrix[][] matrixM2 = FoxAlgorithm.splitMatrixIntoSmallerMatrices(matrix2, splitFactor);

int internalMatrixSize = matrixM1[0][0].getColumnsCount();
Matrix[][] resultMatrixM = new Matrix[splitFactor][splitFactor];

for (int i = 0; i < splitFactor; i++) {
    for (int j = 0; j < splitFactor; j++) {
        resultMatrixM[i][j] = new Matrix(internalMatrixSize, internalMatrixSize);
    }
}
```

Далі ми запускаємо рекурсивно викликаємо FoxAlgorithmTask для підматриць

```
List<FoxAlgorithmTask> tasks = new ArrayList<>();
List<Matrix> calculatedSubBlocks = new ArrayList<>();

for (int i = 0; i < splitFactor; i++) {
    for (int j = 0; j < splitFactor; j++) {
        var task = new FoxAlgorithmTask(
                matrixM1[i][(i + k) % splitFactor],
                matrixM2[(i + k) % splitFactor][j]);

        tasks.add(task);
        task.fork();
    }
}
```

І після цього збираємо результати всіх викликів разом, формуємо результат

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

for (int i = 0; i < splitFactor; i++) {
    for (int j = 0; j < splitFactor; j++) {
        resultMatrixM[i][j].add(calculatedSubBlocks.get(i * splitFactor + j));
    }
}
```

Наприкінці алгоритму нам залишається лише зібрати результуючу матрицю з частин докупи, це і буде результатом виконання методу

```
return FoxAlgorithm.combineMatrixMatricesToMatrix(
        resultMatrixM, matrix1.getRowsCount(), matrix2.getColumnsCount());
```

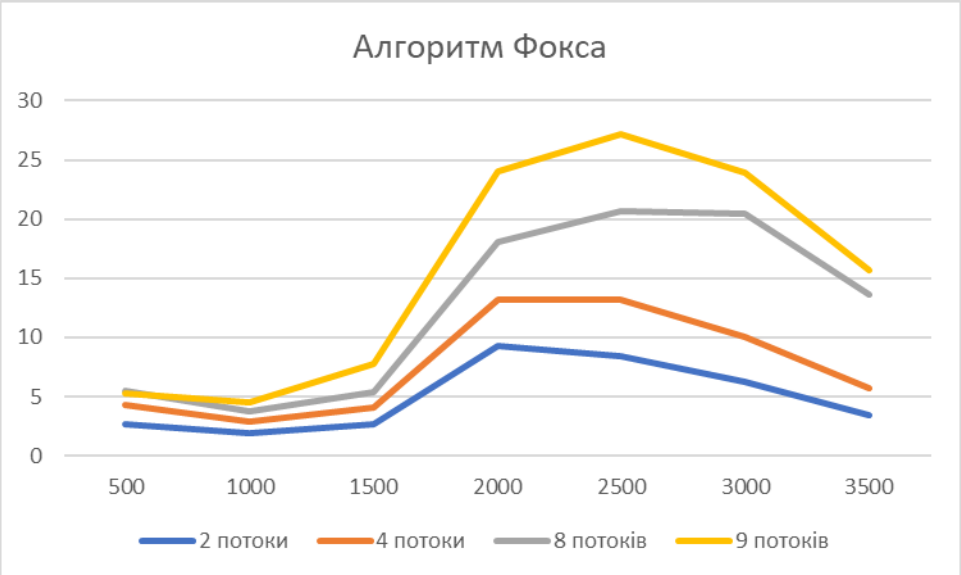
Порівняння швидкодії алгоритму зі стандартною версією алгоритму Фокса

Зробимо аналіз швидкодії стандартного алгоритму Фокса та його модифікації з використанням ForkJoinPool. Для цього запустимо алгоритм на виконання, варіюючи розмірність матриць та кількість потоків.

Маємо такі результати:

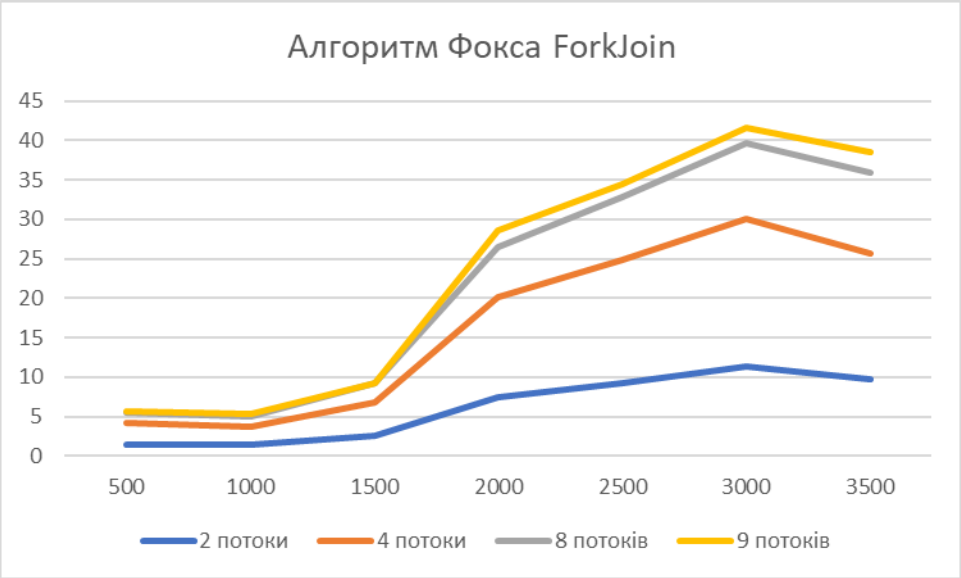
Алгоритм Фокса

Розмір матриці	Послідовний алгоритм	Алгоритм Фокса							
		2 потоки		4 потоки		8 потоків		9 потоків	
		Час, мс	Прискорення	Час, мс	Прискорення	Час, мс	Прискорення	Час, мс	Прискорення
500	158	60	2,633	37	4,270	29	5,448275862	30	5,266666667
1000	1104	580	1,903	379	2,913	298	3,704697987	245	4,506122449
1500	6083	2242	2,713	1483	4,102	1138	5,345342707	779	7,80872914
2000	46338	5002	9,264	3519	13,168	2563	18,07959423	1926	24,05919003
2500	104459	12473	8,375	7900	13,223	5055	20,6644906	3843	27,18162894
3000	207429	33156	6,256	20745	9,999	10124	20,4888384	8668	23,93043378
3500	335823	96995	3,462	59251	5,668	24661	13,61757431	21356	15,72499532

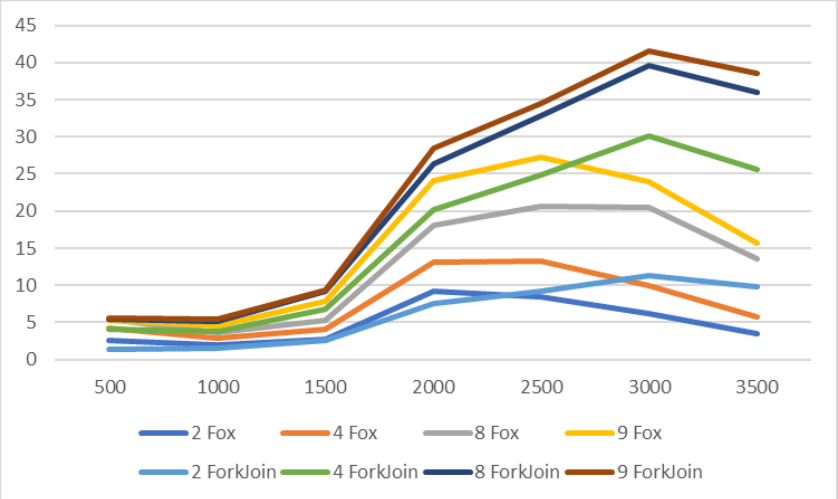


Алгоритм Фокса з використанням ForkJoinPool

Розмір матриці	Послідовний алгоритм	Алгоритм Фокса ForkJoin							
		2 потоки		4 потоки		8 потоків		9 потоків	
		Час, мс	Прискорення	Час, мс	Прискорення	Час, мс	Прискорення	Час, мс	Прискорення
500	158	110	1,436363636	38	4,157894737	29	5,448275862	28	5,642857143
1000	1104	754	1,464190981	294	3,755102041	217	5,087557604	205	5,385365854
1500	6083	2300	2,644782609	887	6,85794814	657	9,258751903	654	9,301223242
2000	46338	6172	7,507777058	2298	20,16449086	1754	26,41847206	1625	28,51569231
2500	104459	11294	9,249070303	4193	24,91271166	3177	32,87976078	3036	34,40678524
3000	207429	18359	11,2984912	6895	30,08397389	5240	39,58568702	4991	41,5606091
3500	335823	34445	9,749542749	13105	25,62556276	9337	35,96690586	8729	38,47210448



Збірний результат



Як бачимо, модифікація алгоритму Фокса пройшла успішно і ми бачимо значний приріст в прискоренні алгоритму. Він особливо помітний при великих значеннях матриць, оскільки там відбувається велика кількість рекурсивних викликів, що і є сильною стороною ForkJoin. Оновлений алгоритм програє стандартному лише при кількості потоків 2, оскільки програмі просто недостатньо ресурсу щоб виконати велику кількість задач лише на 2 потоках

3. **Розробіть та реалізуйте алгоритм пошуку спільних слів в текстових документах з використанням ForkJoinFramework**

Дане завдання має схожий характер до завдання 4.1. про аналіз довжини слів. Аналогічно до завдання 4.1. в алгоритмі були використані класи Folder та TextFile, вони були детально роз'яснені в описі до завдання 4.1.

Для цього завдання було створено декілька класів. Першим з них є CommonWordCounter, в якому ми ініціалізуємо ForkJoinPool з заданою кількістю потоків та викликаємо з використанням методу invoke FolderSearchTask для папки в якій буде проходити пошук спільних слів. Клас CommonWordCounter має наступний вигляд:

```
public class CommonWordCounter {
    2 usages
    private final ForkJoinPool forkJoinPool;

    1 usage  valerii.korniienko
    public CommonWordCounter(int countThreads) { forkJoinPool = new ForkJoinPool(countThreads); }

    1 usage  valerii.korniienko
    public HashSet<String> findCommonWordsForkJoin(Folder folder) {
        return forkJoinPool.invoke(new FolderSearchTask(folder));
    }
}
```

Далі були створені описані вище класи FolderSearchTask та TextFileSearchTask, що наслідуються від RecursiveTask<HashSet<String>>.

Клас FolderSearchTask має в собі єдиний метод compute(). Аналогічно до класу з такою ж назвою в заданні 4.1, цей клас спочатку рекурсивно викликає FolderSearchTask для всіх підпапок, потім ствоює та викликає завдання для всіх текстовиї файлів папки. Наприкінці своєї роботи метод збирає дані з усіх викликаних завдань та виводить результат. compute() має наступний вигляд:

```java
@Override
protected HashSet<String> compute() {
    HashSet<String> commonWords;
    List<RecursiveTask<HashSet<String>>> tasks = new ArrayList<>();

    for (Folder subFolder : folder.getSubFolders()) {
        FolderSearchTask task = new FolderSearchTask(subFolder);
        tasks.add(task);
        task.fork();
    }

    for (TextFile textFile : folder.getTextFiles()) {
        TextFileSearchTask task = new TextFileSearchTask(textFile);
        tasks.add(task);
        task.fork();
    }

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

    return commonWords;
}
```

Клас TextFileSearchTask, який також наслідується від RecursiveTask<HashSet<String>> виконує пошук унікальних слів по тексту. При цьому в даному класі не виконуються рекурсивні виклики, оскільки операція пошуку по файлу вже не потребує поділу на підзадачі. Метод compute() цього класу має наступний вигляд:

```java
@Override
protected HashSet<String> compute() {
    return getUniqueWordsInTextFile(textFile);
}

1 usage  new *
private static HashSet<String> getUniqueWordsInTextFile(TextFile textFile) {
    HashSet<String> uniqueWords = new HashSet<>();
    for (String line : textFile.getLines()) {
        for (String word : getWordsIn(line)) {
            uniqueWords.add(word.toLowerCase());
        }
    }
    return uniqueWords;
}

1 usage  new *
private static String[] getWordsIn(String line) { return line.trim().split( regex: "(\\s|\\p{Punct})+"); }
```

Приклад роботи програми:

```
Common words: [, introduces, functions, chips, your, slower, without, theorists, mflops, these, equations, introduced, would, slows, t
Count: 2251

Time: 1289 ms

Process finished with exit code 0
```

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

Для виконання даного завдання був використаний алгоритм пошуку конкретних ключових слів по файлам, директоріям.

В процесі виконання завдання був використаний підхід аналогічний завданню 4.1 та 4.3

Маємо клас RequiredWordsChecker, головною задачею якого є ініціалізація ForkJoinPool та запуск FolderSearchTask з використанням методу invoke

```
public class RequiredWordsChecker {
    2 usages
    private final ForkJoinPool forkJoinPool;

    1 usage  valerii.korniienko
    public RequiredWordsChecker(int countThreads) { forkJoinPool = new ForkJoinPool(countThreads); }

    1 usage  valerii.korniienko
    public HashMap<String, List<String>> findCommonWordsForkJoin(Folder folder, List<String> requiredWords) {
        List<String> wordsToLowerCase = new ArrayList<>();
        for (String word : requiredWords) {
            wordsToLowerCase.add(word.toLowerCase());
        }
        return forkJoinPool.invoke(new FolderSearchTask(folder, wordsToLowerCase));
    }
}
```

Також маємо клас FolderSearchTask extends RecursiveTask<HashMap<String, List<String>>>, головною задачею якого є запуск задач FolderSearchTask для всіх підпапок та TextFileSearchTask для всіх текстових файлів. Також даний клас агрегує результати всіх задач, які він запустив на виконання.

```
@Override
protected HashMap<String, List<String>> compute() {
    HashMap<String, List<String>> filesWithRequiredWords = new HashMap<>();
    List<RecursiveTask<HashMap<String, List<String>>>> tasks = new ArrayList<>();

    for (Folder subFolder : folder.getSubFolders()) {
        FolderSearchTask task = new FolderSearchTask(subFolder, wordsMustExist);
        tasks.add(task);
        task.fork();
    }

    for (TextFile textFile : folder.getTextFiles()) {
        TextFileSearchTask task = new TextFileSearchTask(textFile, wordsMustExist);
        tasks.add(task);
        task.fork();
    }

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

    return filesWithRequiredWords;
}
```

Наступним класом є TextFileSearchTask який також наслідується від RecursiveTask<HashMap<String, List<String>>>. Його задачею є безпосередня перевірка наявності ключових слів в файлах

```java
@Override
protected HashMap<String, List<String>> compute() {
    return checkExistWords(textFile, wordsMustExist);
}

1 usage  new *
private static HashMap<String, List<String>> checkExistWords(TextFile textFile, List<String> requiredWords) {
    HashSet<String> uniqueWords = new HashSet<>();
    for (String line : textFile.getLines()) {
        for (String word : getWordsInLine(line)) {
            uniqueWords.add(word.toLowerCase());
        }
    }

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

    HashMap<String, List<String>> map = new HashMap<>();
    map.put(textFile.getName(), matchedWords);

    return map;
}

1 usage  new *
private static String[] getWordsInLine(String line) { return line.trim().split( regex: "(\\s|\\p{Punct})+"); }
```

Приклад результату роботи програми:

```
C:\Projects\ParrallelComputing\TPO_labs\lab4_1\TestFolder1\New folder - Copy (7)\parallelPrinciples - Copy (3).txtparallelPrinciples - Copy (3).txt=[algorithm, parallel, computer]
C:\Projects\ParrallelComputing\TPO_labs\lab4_1\TestFolder1\New folder - Copy (8)\parallelPrinciples.txtparallelPrinciples.txt=[algorithm, parallel, computer]
C:\Projects\ParrallelComputing\TPO_labs\lab4_1\TestFolder1\New folder - Copy (10)\parallelPrinciples - Copy (13).txtparallelPrinciples - Copy (13).txt=[algorithm, parallel, computer]
C:\Projects\ParrallelComputing\TPO_labs\lab4_1\TestFolder1\New folder\New folder - Copy (11)\parallelPrinciples - Copy (3).txtparallelPrinciples - Copy (3).txt=[algorithm, parallel, computer]
C:\Projects\ParrallelComputing\TPO_labs\lab4_1\TestFolder1\New folder\New folder - Copy (10)\parallelPrinciples - Copy (8).txtparallelPrinciples - Copy (8).txt=[algorithm, parallel, computer]
C:\Projects\ParrallelComputing\TPO_labs\lab4_1\TestFolder1\New folder\New folder - Copy (12)\parallelPrinciples - Copy (12).txtparallelPrinciples - Copy (12).txt=[algorithm, parallel, computer]
C:\Projects\ParrallelComputing\TPO_labs\lab4_1\TestFolder1\New folder - Copy (9)\parallelPrinciples - Copy (7).txtparallelPrinciples - Copy (7).txt=[algorithm, parallel, computer]
C:\Projects\ParrallelComputing\TPO_labs\lab4_1\TestFolder1\New folder - Copy (6)\parallelPrinciples - Copy (10).txtparallelPrinciples - Copy (10).txt=[algorithm, parallel, computer]
C:\Projects\ParrallelComputing\TPO_labs\lab4_1\TestFolder1\New folder\New folder - Copy (2)\parallelPrinciples - Copy (7).txtparallelPrinciples - Copy (7).txt=[algorithm, parallel, computer]

Don't have required words: 0
Have not all required words: 420
Have all required words: 18

Time: 1487 ms
```

Лістинг коду програми

Спільні файли:

Folder.java

```
import java.io.File;
import java.io.IOException;
import java.util.LinkedList;
import java.util.List;
import java.util.Objects;

public class Folder {
    private final List<Folder> subFolders;
    private final List<TextFile> textFiles;

    public Folder(List<Folder> subFolders, List<TextFile>
textFiles) {
        this.subFolders = subFolders;
        this.textFiles = textFiles;
    }
    List<TextFile> getTextFiles() {
        return this.textFiles;
    }
    List<Folder> getSubFolders() {
        return this.subFolders;
    }

    public static Folder loadFromDirectory(File dir) throws
IOException {
        List<TextFile> documents = new LinkedList<>();
        List<Folder> subFolders = new LinkedList<>();

        for (File entry :
Objects.requireNonNull(dir.listFiles())) {
            if (entry.isDirectory()) {

subFolders.add(Folder.loadFromDirectory(entry));
            } else {
                documents.add(TextFile.createFromFile(entry));
            }
        }
        return new Folder(subFolders, documents);
    }
}
```

TextFile.java

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
```

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

public class TextFile {
    private final List<String> lines;
    private final String name;

    public TextFile(List<String> lines, String name) {
        this.lines = lines;
        this.name = name;
    }

    public List<String> getLines() {
        return this.lines;
    }
    public String getName() {
        return this.name;
    }

    public static TextFile createFromFile(File file) throws
IOException {
        String path = file.getPath() + file.getName();
        List<String> lines = new ArrayList<>();
        try (BufferedReader reader = new BufferedReader(new
FileReader(file))) {
            String line = reader.readLine();
            while (line != null) {
                lines.add(line);
                line = reader.readLine();
            }
        }
        return new TextFile(lines, path);
    }
}
```

Task 4.1

FolderSearchTask.java

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

class FolderSearchTask extends RecursiveTask<List<Integer>> {
    private final Folder folder;

    FolderSearchTask(Folder folder) {
        this.folder = folder;
    }

    @Override
    protected List<Integer> compute() {
        ArrayList<Integer> wordLengths = new ArrayList<>();
        List<RecursiveTask<List<Integer>>> tasks = new
LinkedList<>();

        for (Folder subFolder : folder.getSubFolders()) {
            FolderSearchTask task = new
FolderSearchTask(subFolder);
            tasks.add(task);
            task.fork();
        }

        for (TextFile textFile : folder.getTextFiles()) {
            TextFileSearchTask task = new
TextFileSearchTask(textFile);
            tasks.add(task);
            task.fork();
        }

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

TextFileSearch.java

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

class TextFileSearchTask extends RecursiveTask<List<Integer>>
{
```

```
    private final TextFile textFile;

    TextFileSearchTask(TextFile textFile) {
        this.textFile = textFile;
    }

    @Override
    protected List<Integer> compute() {
        return getAllWordLengths(textFile);
    }

    private static List<Integer> getAllWordLengths(TextFile
textFile) {
        List<Integer> wordLengths = new ArrayList<>();
        for (String line : textFile.getLines()) {
            for (String word : getAllWordsInLine(line)) {
                wordLengths.add(word.length());
            }
        }
        return wordLengths;
    }
    private static String[] getAllWordsInLine(String line) {
        return line.trim().split("(\\s|\\p{Punct})+");
    }
}
```

WordCounter.java

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

public class WordCounter {
    private final ForkJoinPool forkJoinPool;

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

    public List<Integer> getAllWordLenghtsForkJoin(Folder
folder) {
        return forkJoinPool.invoke(new
FolderSearchTask(folder));
    }
}
```

Main.java

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

```java
public class Main {
    public static void main(String[] args) throws IOException
{
        File file = new
File("C:\\Projects\\ParrallelComputing\\TPO_labs\\lab4_1/TestF
older");
        System.out.println(file.getAbsolutePath());
        Folder folder = Folder.loadFromDirectory(file);
        WordCounter wordCounter = new WordCounter(2);

        long startTime = System.currentTimeMillis();
        List<Integer> wordLengths =
wordCounter.getAllWordLenghtsForkJoin(folder);
        long totalTime = System.currentTimeMillis() -
startTime;

        int totalLength = 0;
        int count = 0;
        for (var length : wordLengths) {
            totalLength += length;
            count++;
        }
        double meanLength = (double) totalLength / count;

        int totalSquaredLength = 0;
        for (var length : wordLengths) {
            totalSquaredLength += Math.pow(length,2);
        }

        double D = ((double) totalSquaredLength /count) -
Math.pow(meanLength,2);
        double G = Math.sqrt(D);

        System.out.println("Count: " + count);
        System.out.println("Mean length: " + meanLength);
        System.out.println("Dispersion: " + D);
        System.out.println("Mean square deviation: " + G);
        System.out.println("\nTime: " + totalTime + " ms");

    }
}
```

Task 4.2

FoxAlgorithmForkJoin.java

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

public class FoxAlgorithmForkJoin implements
IMatrixMultiplicationAlgorithm {
    private final ForkJoinPool forkJoinPool;

    public FoxAlgorithmForkJoin(int countThreads) {
        forkJoinPool = new ForkJoinPool(countThreads);
    }
    @Override
    public Result multiply(Matrix matrixA, Matrix matrixB) {
        long startTime = System.currentTimeMillis();

        return new Result(forkJoinPool.invoke(new
FoxAlgorithmTask(matrixA, matrixB)),
                System.currentTimeMillis() - startTime);
    }
}
```

FoxAlgorithmTask.java

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

public class FoxAlgorithmTask extends RecursiveTask<Matrix> {

    private Matrix matrix1;
    private Matrix matrix2;
    private final int matrixSizeLimit = 100;
    private final int splitFactor = 2;

    public FoxAlgorithmTask(Matrix matrix1, Matrix matrix2) {
        this.matrix1 = matrix1;
        this.matrix2 = matrix2;
    }

    @Override
    public Matrix compute() {
        if (matrix1.getColumnsCount() <= matrixSizeLimit) {
            return matrix1.multiply(matrix2);
        }

        Matrix[][] matrixM1 =
FoxAlgorithm.splitMatrixIntoSmallerMatrices(matrix1,
splitFactor);
        Matrix[][] matrixM2 =
```

```
FoxAlgorithm.splitMatrixIntoSmallerMatrices(matrix2,
splitFactor);

        int internalMatrixSize =
matrixM1[0][0].getColumnsCount();
        Matrix[][] resultMatrixM = new
Matrix[splitFactor][splitFactor];

        for (int i = 0; i < splitFactor; i++) {
            for (int j = 0; j < splitFactor; j++) {
                resultMatrixM[i][j] = new
Matrix(internalMatrixSize, internalMatrixSize);
            }
        }

        for (int k = 0; k < splitFactor; k++) {
            List<FoxAlgorithmTask> tasks = new ArrayList<>();
            List<Matrix> calculatedSubBlocks = new
ArrayList<>();

            for (int i = 0; i < splitFactor; i++) {
                for (int j = 0; j < splitFactor; j++) {
                    var task = new FoxAlgorithmTask(
                            matrixM1[i][(i + k) %
splitFactor],
                            matrixM2[(i + k) %
splitFactor][j]);

                    tasks.add(task);
                    task.fork();
                }
            }

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

            for (int i = 0; i < splitFactor; i++) {
                for (int j = 0; j < splitFactor; j++) {

resultMatrixM[i][j].add(calculatedSubBlocks.get(i *
splitFactor + j));
                }
            }
        }

        return FoxAlgorithm.combineMatrixMatricesToMatrix(
                resultMatrixM, matrix1.getRowsCount(),
matrix2.getColumnsCount());
```

```
    }
}
```

Main.java

```
public class Main {
    public static void main(String[] args) {
        int[] matrixSizes = {500, 1000, 1500, 2000, 2500,
3000, 3500};
        int[] threadsCounts = {2, 4, 8, 9};

        testAlgorithmsSpeed(matrixSizes, threadsCounts);
    }
    private static void testAlgorithmsSpeed(int[] matrixSizes,
int[] threadsCounts) {
        for (int matrixSize : matrixSizes) {
            Matrix matrixA =
MatrixHelper.generateRandomMatrix(matrixSize);
            Matrix matrixB =
MatrixHelper.generateRandomMatrix(matrixSize);
            System.out.println("-------------------------");
            System.out.println("Matrix size: " + matrixSize);

            long sequentialTime = checkAlgorithmSpeed(matrixA,
matrixB, new SequentialAlgorithm(), 1);
            System.out.println("\nSequential algorithm: " +
sequentialTime + " ms");

            for (int threads : threadsCounts) {
                System.out.println("\nThreads count: " +
threads);

                long foxForkJoinTime =
checkAlgorithmSpeed(matrixA, matrixB, new
FoxAlgorithmForkJoin(threads), 5);
                long foxTime = checkAlgorithmSpeed(matrixA,
matrixB, new FoxAlgorithm(threads), 5);

                System.out.println("\tFox algorithm with " +
threads + " threads: " + foxTime + " ms");
                System.out.println("\tFox algorithm with " +
threads + " threads and ForkJoin: " + foxForkJoinTime + "
ms");
            }
        }
    }
    static long checkAlgorithmSpeed(Matrix matrixA, Matrix
matrixB, IMatrixMultiplicationAlgorithm
multiplicationAlgorithm, int iterations) {
        long sum = 0;
        for (int i = 0; i < iterations; i++) {
```

```
            sum += multiplicationAlgorithm.multiply(matrixA,
matrixB).getTotalTime();
        }
        return sum / iterations;
    }
}
```

Task 4.3

CommonWordCounter.java

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

public class CommonWordCounter {
    private final ForkJoinPool forkJoinPool;

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

    public HashSet<String> findCommonWordsForkJoin(Folder
folder) {
        return forkJoinPool.invoke(new
FolderSearchTask(folder));
    }
}
```

FolderSearchTask.java

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

class FolderSearchTask extends RecursiveTask<HashSet<String>>
{
    private final Folder folder;

    FolderSearchTask(Folder folder) {
        this.folder = folder;
    }

    @Override
    protected HashSet<String> compute() {
        HashSet<String> commonWords;
        List<RecursiveTask<HashSet<String>>> tasks = new
ArrayList<>();

        for (Folder subFolder : folder.getSubFolders()) {
            FolderSearchTask task = new
FolderSearchTask(subFolder);
            tasks.add(task);
            task.fork();
        }

        for (TextFile textFile : folder.getTextFiles()) {
```

```
            TextFileSearchTask task = new
TextFileSearchTask(textFile);
            tasks.add(task);
            task.fork();
        }

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

        return commonWords;
    }
}
```

TextFileSearchTask.java

```
import java.util.HashSet;
import java.util.concurrent.RecursiveTask;

class TextFileSearchTask extends
RecursiveTask<HashSet<String>> {
    private final TextFile textFile;

    TextFileSearchTask(TextFile textFile) {
        this.textFile = textFile;
    }

    @Override
    protected HashSet<String> compute() {
        return getUniqueWordsInTextFile(textFile);
    }

    private static HashSet<String>
getUniqueWordsInTextFile(TextFile textFile) {
        HashSet<String> uniqueWords = new HashSet<>();
        for (String line : textFile.getLines()) {
            for (String word : getWordsIn(line)) {
                uniqueWords.add(word.toLowerCase());
            }
        }
        return uniqueWords;
    }

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

}
```

Main.java

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

public class Main {
    public static void main(String[] args) throws IOException
{
        File file = new
File("C:\\Projects\\ParrallelComputing\\TPO_labs\\lab4_1/TestF
older");
        Folder folder = Folder.loadFromDirectory(file);
        CommonWordCounter commonWordSearcher = new
CommonWordCounter(2);

        long startTime = System.currentTimeMillis();
        HashSet<String> commonWords =
commonWordSearcher.findCommonWordsForkJoin(folder);
        long time = System.currentTimeMillis() - startTime;

        System.out.println("Common words: " + commonWords);
        System.out.println("Count: " + commonWords.size());
        System.out.println("\nTime: " + time + " ms");
    }
}
```

Task 4.4

RequiredWordsChecker.java

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

public class RequiredWordsChecker {
    private final ForkJoinPool forkJoinPool;

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

    public HashMap<String, List<String>>
findCommonWordsForkJoin(Folder folder, List<String>
requiredWords) {
        return forkJoinPool.invoke(new
FolderSearchTask(folder, requiredWords));
    }
}
```

FolderSearchTask.java

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.concurrent.ForkJoinPool;

public class RequiredWordsChecker {
    private final ForkJoinPool forkJoinPool;

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

    public HashMap<String, List<String>>
findCommonWordsForkJoin(Folder folder, List<String>
requiredWords) {
        List<String> wordsToLowerCase = new ArrayList<>();
        for (String word : requiredWords) {
            wordsToLowerCase.add(word.toLowerCase());
        }
        return forkJoinPool.invoke(new
FolderSearchTask(folder, wordsToLowerCase));
    }
}
```

TextFileSearchTask.java

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

class TextFileSearchTask extends RecursiveTask<HashMap<String,
List<String>>> {
    private final TextFile textFile;
    private final List<String> wordsMustExist;

    TextFileSearchTask(TextFile textFile, List<String>
wordsMustExist) {
        this.textFile = textFile;
        this.wordsMustExist = wordsMustExist;
    }

    @Override
    protected HashMap<String, List<String>> compute() {
        return checkExistWords(textFile, wordsMustExist);
    }

    private static HashMap<String, List<String>>
checkExistWords(TextFile textFile, List<String> requiredWords)
{
        HashSet<String> uniqueWords = new HashSet<>();
        for (String line : textFile.getLines()) {
            for (String word : getWordsInLine(line)) {
                uniqueWords.add(word.toLowerCase());
            }
        }

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

        HashMap<String, List<String>> map = new HashMap<>();
        map.put(textFile.getName(), matchedWords);

        return map;
    }

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

Main.java

```java
import java.io.File;
import java.io.IOException;
```

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

public class Main {
    public static void main(String[] args) throws IOException
{
        File file = new
File("C:\\Projects\\ParrallelComputing\\TPO_labs\\lab4_1/TestF
older");
        Folder folder = Folder.loadFromDirectory(file);
        RequiredWordsChecker requiredWordsChecker = new
RequiredWordsChecker(2);

        List<String> words = new ArrayList<>();
        words.add("Algorithm");
        words.add("Java");
        words.add("Networking");
        words.add("Database");
        words.add("Computer");

        long startTime = System.currentTimeMillis();
        HashMap<String, List<String>> fileAndExistWords =
requiredWordsChecker.findCommonWordsForkJoin(folder, words);
        long time = System.currentTimeMillis() - startTime;

        int dontHaveRequiredWords = 0;
        int haveNotAllRequiredWords = 0;
        int haveAllRequiredWords = 0;

        for (var item: fileAndExistWords.entrySet()){
            System.out.println(item);
            if (item.getValue().size() == 0){
                dontHaveRequiredWords++;
            }
            else if (item.getValue().size() < words.size()){
                haveNotAllRequiredWords++;
            }
            else{
                haveAllRequiredWords++;
            }
        }

        System.out.println("\nDon't have required words: " +
dontHaveRequiredWords);
        System.out.println("Have not all required words: " +
haveNotAllRequiredWords);
        System.out.println("Have all required words: " +
haveAllRequiredWords);
        System.out.println("\nTime: " + time + " ms");
```

```
        }
}
```