НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ім. І. СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра автоматики та управління в технічних системах

Лабораторна робота №4

із дисципліни «Технології паралельних та розподілених обчислень»

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

Підготував:

Студент ФІОТ, гр. ІТ-83,

Троян О.С,

Перевірив:

пос. Дифучина О. Ю.

Київ 2020

Мета роботи: Розробити паралельну програму з використанням пулів потоків, екзекьюторів та ForkJoinFramework

Завдання:

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

   20 балів.

2. 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. Розробити та реалізувати алгоритм пошуку спільних слів в текстових документах з використанням ForkJoinFramework.

   20 балів.

5. Розробити та реалізувати алгоритм пошуку текстових документів, які відповідають заданим ключовим словам (належать до області «Інформаційні технології»), з використанням ForkJoinFramework.

   30 балів.

Хід роботи

1. Я створив алгоритми що визначає характеристику випадкової величини кількості символів в словах. Для порівняння швидкодії один з алгоритмів працює в одному потоці, а інший використовує технологію ForkJoinFramework.

   1.1. Одиночний потік

```java
public RandomVariableCharacteristics countWordsLengthInSingleThread(Document document) {
    Hashtable<Integer, Integer> wordsLengths = new Hashtable<>();

    for (String line : document.getLines()) {
        for (String word : Helper.wordsIn(line)) {
            Integer count = wordsLengths.getOrDefault(word.length(), defaultValue: 0);
            wordsLengths.put(word.length(), ++count);
        }
    }

    int wordsSum = wordsLengths.values().stream().mapToInt(i -> i).sum();
    final double mathExpected = wordsLengths.entrySet().stream()
            .mapToDouble(i -> (double) i.getKey() * i.getValue()).sum() / wordsSum;
    final double disperse = wordsLengths.entrySet().stream()
            .mapToDouble(i -> Math.pow((i.getKey() - mathExpected), b: 2) * i.getValue()).sum() / wordsSum;

    return new RandomVariableCharacteristics(wordsLengths, mathExpected, disperse, Math.sqrt(disperse));
}
```

Використовується простий цикл для перебору рядків, а потім слів в них. Далі з отриманих параметрів розраховується дисперсія.

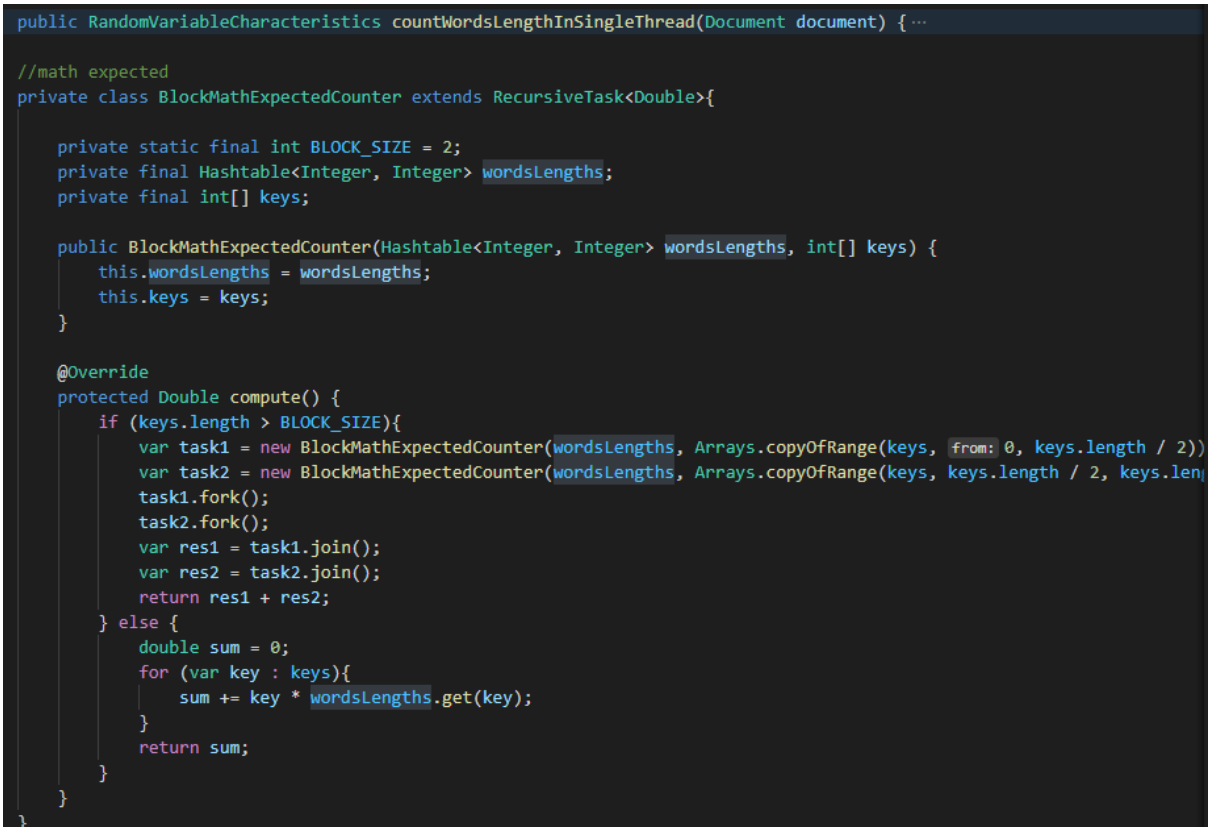
Результат:

```
Single thread
Time: 360
Length 18: 1
Length 17: 2
Length 15: 12
Length 14: 34
Length 13: 67
Length 12: 572
Length 11: 430
Length 10: 1390
Length 9: 2342
Length 8: 3981
Length 7: 6251
Length 6: 8323
Length 5: 10894
Length 4: 15274
Length 3: 21318
Length 2: 11254
Length 1: 2921
Length 0: 5131
Math expected: 4.197955586105968
Disperse: 5.6809705756297655
Deviation: 2.383478671108631
```

1.2. ForkJoin

```java
public RandomVariableCharacteristics countWordsLengthInRealParallel(Document document) {
    var wordsLengths = forkJoinPool.invoke(new BlockWordsCounterTask(document.getLines().toArray(new String[0])));

    var keys = Collections.list(wordsLengths.keys()).stream().mapToInt(i -> i).toArray();
    int wordsSum = wordsLengths.values().stream().mapToInt(i -> i).sum();
    var mathExpecte ForkJoinPool forkJoinPool BlockMathExpectedCounter(wordsLengths, keys)) / wordsSum;
    var disperse = forkJoinPool.invoke(new BlockDisperseCounter(wordsLengths, keys, mathExpected)) / wordsSum;

    return new RandomVariableCharacteristics(wordsLengths, mathExpected, disperse, Math.sqrt(disperse));
}
```

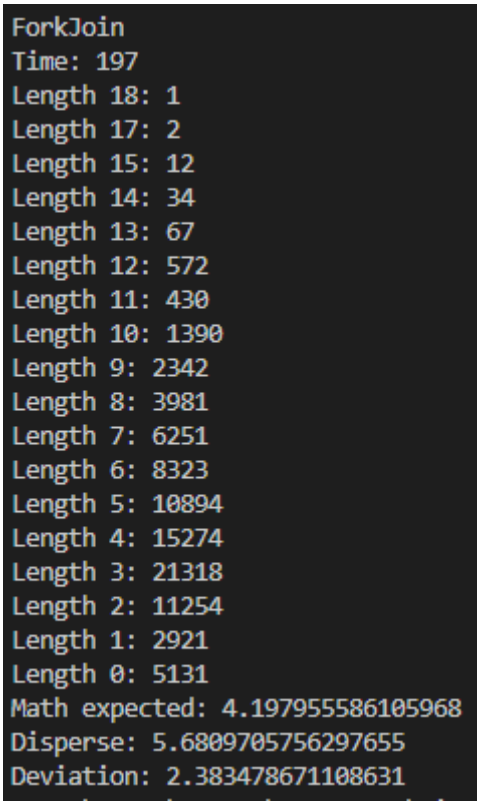
Для пошуку слів, розрахунку дисперсії та математичного очікування використовується ForkJoinFramework,
Для прикладу пошук слів:



```
public RandomVariableCharacteristics countWordsLengthInSingleThread(Document document) {···

//math expected
private class BlockMathExpectedCounter extends RecursiveTask<Double>{

    private static final int BLOCK_SIZE = 2;
    private final Hashtable<Integer, Integer> wordsLengths;
    private final int[] keys;

    public BlockMathExpectedCounter(Hashtable<Integer, Integer> wordsLengths, int[] keys) {
        this.wordsLengths = wordsLengths;
        this.keys = keys;
    }

    @Override
    protected Double compute() {
        if (keys.length > BLOCK_SIZE){
            var task1 = new BlockMathExpectedCounter(wordsLengths, Arrays.copyOfRange(keys, from: 0, keys.length / 2))
            var task2 = new BlockMathExpectedCounter(wordsLengths, Arrays.copyOfRange(keys, keys.length / 2, keys.len
            task1.fork();
            task2.fork();
            var res1 = task1.join();
            var res2 = task2.join();
            return res1 + res2;
        } else {
            double sum = 0;
            for (var key : keys){
                sum += key * wordsLengths.get(key);
            }
            return sum;
        }
    }
}
```

Весь текст постійно ділиться на 2 блоки допоки розмір блоку не буде менше за вказаний, ця операція відбувається рекурсивно, клолиж умова відбувається рекурсія переривається і починається розрахунок кількості символів в слові.

За аналогічним принципом відбувається і розрахунок математичних значень.



Як видно з результатів математичне очікування, дисперсія та середньоквадратичне відхилення співпадають.

1.3. При великій кількості тексту ForkJoin швидше виконує свою роботу, але якщо тексту буде мало то звичайне однопоточне виконання буде швидшим адже ForkJoin буде витрачати більше часу на переключення між потоками ніж на розрахунок.

2. Для порівняння алгоритмів я використав стрічковий алгоритм множення матриць, та написав цей алгоритм з використанням ForkJoinFramework.

```java
protected void compute() {
    int difference = toRow - fromRow;
    if (difference > 0){
        int midRow = fromRow + (difference / 2);
        var task1 = new ForkJoinTapeWorker(fromRow, midRow, parameters);
        var task2 = new ForkJoinTapeWorker(midRow + 1, toRow, parameters);
        task1.fork();
        task2.fork();
        task1.join();
        task2.join();
    } else {
        int row = toRow;
        int column = getCorrectColumn(row);
        result[row][column] = multiplyRowAndColumn(row, column);
    }
}
```

Кожна ітерація це розрахунок одного рядка результуючої матриці, завдяки рекурсії розраховується кожен елемент результуючої матриці (в рамках рядка) окремо.

Результат:

```
Size test: 500 matrix rank
Tape method: 2304
Fork join tape method: 493
Speed up: 4.67342799188641

Size test: 1000 matrix rank
Tape method: 14540
Fork join tape method: 6606
Speed up: 2.2010293672419015

Size test: 1500 matrix rank
Tape method: 36320
Fork join tape method: 25371
Speed up: 1.4315557132158765

Size test: 2000 matrix rank
Tape method: 71431
Fork join tape method: 55953
Speed up: 1.2766250245741961
```

ForkJoin швидше ніж звичайний стрічковий алгоритм.

3. Пошук слів в папках відбувається за допомогою ForkJoin

3.1. Пошук всіх папок відбувається рекурсивно

Початкову папку вказуємо самостійно

```java
public static void main(String[] args) throws IOException {
    var folder = Folder.fromDirectory(new File( pathname: "Lab4/src/documents"));
    var browser = new FilesBrowser();

    Set<String> searchedWords = ConcurrentHashMap.newKeySet();
    searchedWords.addAll(Arrays.asList( ...a: "code","programming"));

    var result = browser.findFilesByTheme(folder, searchedWords);

    for (String file : result){
        System.out.println(file);
    }
}
```

Далі через цикл йде перевірка наявних папок, якщо такі є то рекурсивно повторюється пошук папок.

```java
@Override
protected List<String> compute() {
    List<String> files = new LinkedList<>();
    List<RecursiveTask<List<String>>> tasks = new LinkedList<>();

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

    for (Document document : folder.getDocuments()) {
        DocumentSearchTask task = new DocumentSearchTask(document, searchedWords);
        tasks.add(task);
        task.fork();
    }

    for (RecursiveTask<List<String>> task : tasks) {
        List<String> result = task.join();
        if (result != null){
            files.addAll(result);
        }
    }

    return files;
}
```

3.2. Після того як всі папки було знайдено починається пошук файлів та пошук слів в них.

```java
class DocumentSearchTask extends RecursiveTask<List<String>> {
    private final Document document;
    private final Set<String> searchedWords;

    public DocumentSearchTask(Document document, Set<String> searchedWords) {
        this.document = document;
        this.searchedWords = searchedWords;
    }

    @Override
    protected List<String> compute() {

        for (String line : document.getLines()) {
            for (String word : Helper.wordsIn(line)) {
                if (searchedWords.contains(word.toLowerCase(Locale.ROOT))){
                    return Collections.singletonList(document.getPath());
                }
            }
        }
        return null;
    }
}
```

Якщо слово присутнє в файлі ми зберігаємо його шлях.

3.3. Виводимо результати пошуку по всім папкам та файлам всередині них.

```java
for (RecursiveTask<List<String>> task : tasks) {
    List<String> result = task.join();
    if (result != null){
        files.addAll(result);
    }
}

return files;
```

Результат:

```
Lab4\src\documents\files\additional\codeInfo
Lab4\src\documents\files\News Lines
Lab4\src\documents\files\Top 7 books
Lab4\src\documents\Program.java
```

4. Пошук схожих слів за допомогою ForkJoin.

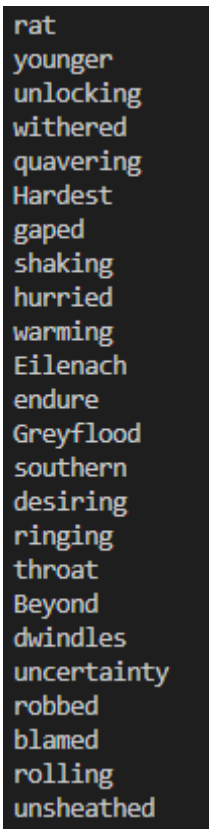
За основу я взяв 2 книги одного автора.

Пошук відбувається рекурсивно

```java
@Override
protected Result compute() {
    if (firstTextLines.length > BLOCK_SIZE && secondTextLines.length > BLOCK_SIZE){
        return forkTasks(Arrays.copyOfRange(firstTextLines, from: 0, firstTextLines.length / 2),
                Arrays.copyOfRange(firstTextLines, firstTextLines.length / 2, firstTextLines.length),
                Arrays.copyOfRange(secondTextLines, from: 0, secondTextLines.length / 2),
                Arrays.copyOfRange(secondTextLines, secondTextLines.length / 2, secondTextLines.length));
    } else if (firstTextLines.length > BLOCK_SIZE){
        return forkTasks(Arrays.copyOfRange(firstTextLines, from: 0, firstTextLines.length / 2),
                Arrays.copyOfRange(firstTextLines, firstTextLines.length / 2, firstTextLines.length),
                secondTextLines,
                secondTextLines);
    } else if (secondTextLines.length > BLOCK_SIZE){
        return forkTasks(firstTextLines,
                firstTextLines,
                Arrays.copyOfRange(secondTextLines, from: 0, secondTextLines.length / 2),
                Arrays.copyOfRange(secondTextLines, secondTextLines.length / 2, secondTextLines.length));
    } else {
        Set<String> firstTextWords = ConcurrentHashMap.newKeySet();
        Set<String> secondTextWords = ConcurrentHashMap.newKeySet();

        for (String line : firstTextLines) {
            firstTextWords.addAll(Arrays.asList(Helper.wordsIn(line)));
        }

        for (String line : secondTextLines) {
            secondTextWords.addAll(Arrays.asList(Helper.wordsIn(line)));
        }

        Set<String> result = ConcurrentHashMap.newKeySet();
        result.addAll(firstTextWords);
        result.retainAll(secondTextWords);

        firstTextWords.removeAll(result);
        secondTextWords.removeAll(result);

        return new Result(firstTextWords, secondTextWords, result);
    }
}
```

обидва документи починають ділится на 2 блоки до тих пір поки кожен з них не досягне заданого розміру. Далі знаходятся спільні слова цих блоків.

Результат:



Висновок: Під час виконання лабораторної роботи ми розробили паралельні програми з використанням з використанням пулів потоків, екзекьюторів та ForkJoinFramework.