**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)**

(шифр, прізвище, ім'я, по батькові)

*ІП-01 Корнієнко В.С.*

**Перевірив(ла)**

(прізвище, ім'я, по батькові)

*Стеценко І. В.*

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.**

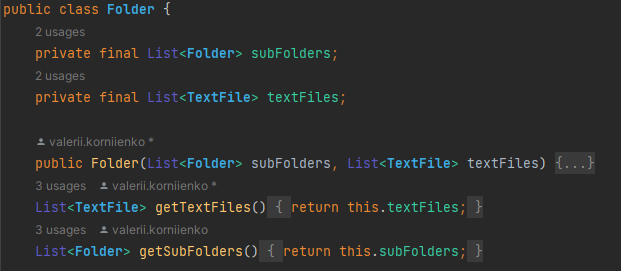
**Хід роботи:**

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

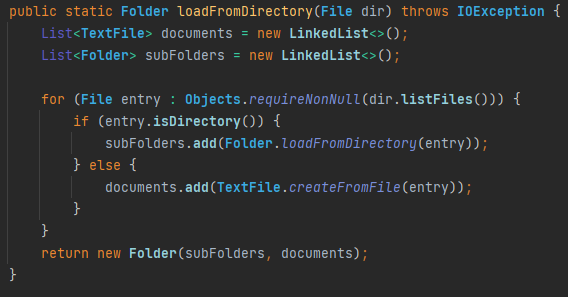
Оскільки завдання 1, 3, 4 орієнтовані на роботу з файловою системою, а саме на роботу з текстовими файлами та папками, створимо класи TextFile та Folder відповідно.

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:

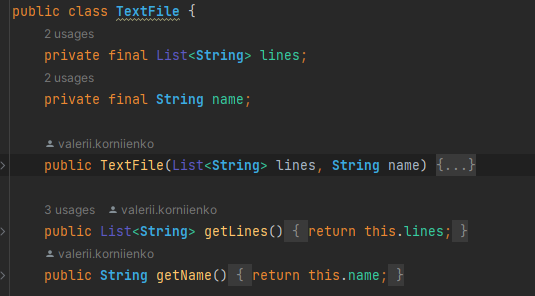


Статична фабрика Folder:

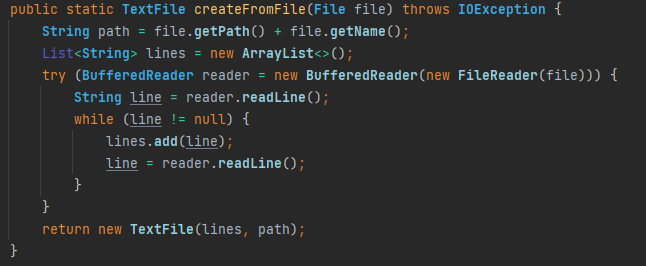


В свою чергу клас TextFile містить в собі всі лінії текстового файлу та його ім’я. Також, аналогічно класу Folder, даний клас має статичну фабрику

Клас TextFile має наступну структуру:



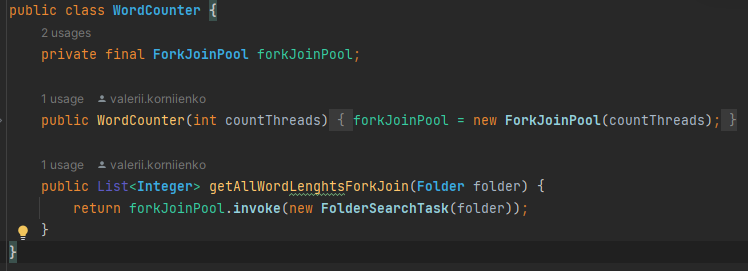
Статична фабрика TextFile:



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.

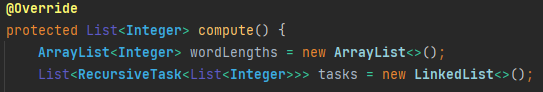
Клас WordCounter має наступний вигляд:



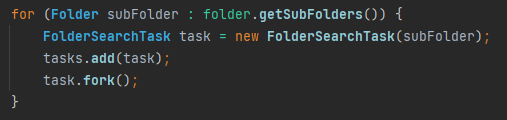
Далі були створені описані вище класи FolderSearchTask та TextFileSearchTask, що наслідуються від RecursiveTask<List<Integer>>.

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

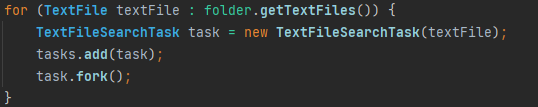
Спочатку ми ініціалізуємо структури для зберігання кількості слів, рекурсивних завдань:



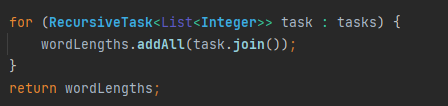
Далі ми перебираємо всі вкладені папки та створюємо завдання для кожної з них, рекурсивно запускаємо їх на паралельне виконання за допомогою методу fork()



Потім робимо аналогічну операцію для текстових файлів

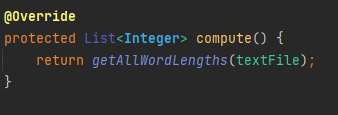


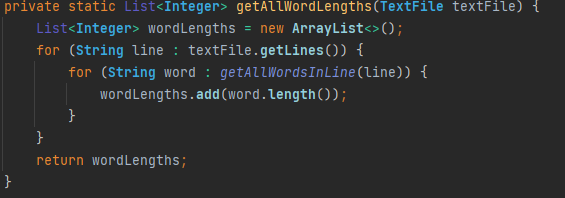
Наприкінці методу ми збираємо результати з усіх завдань та додаємо їх в наш список довжин слів, повертаємо результат



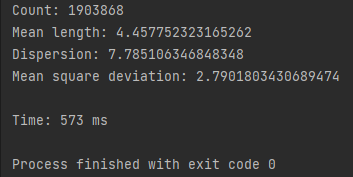
Клас TextFileSearchTask містить в собі метод compute(), який рахує кількості всіх слів в файлі та повертає їх. Варто зазначити що в цьому методі не відбувається паралельних рекурсивних викликів, оскільки операція рахування слів в файлі не потребує розпаралелювання

Метод compute() має наступний вигляд:





Приклад результату роботи програми:

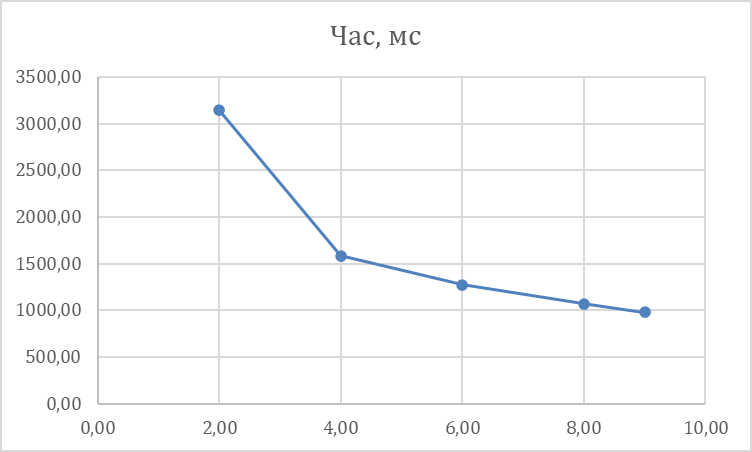


* 1. **Дослідіть побудований алгоритм аналізу текстових документів на ефективність експериментально.**

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

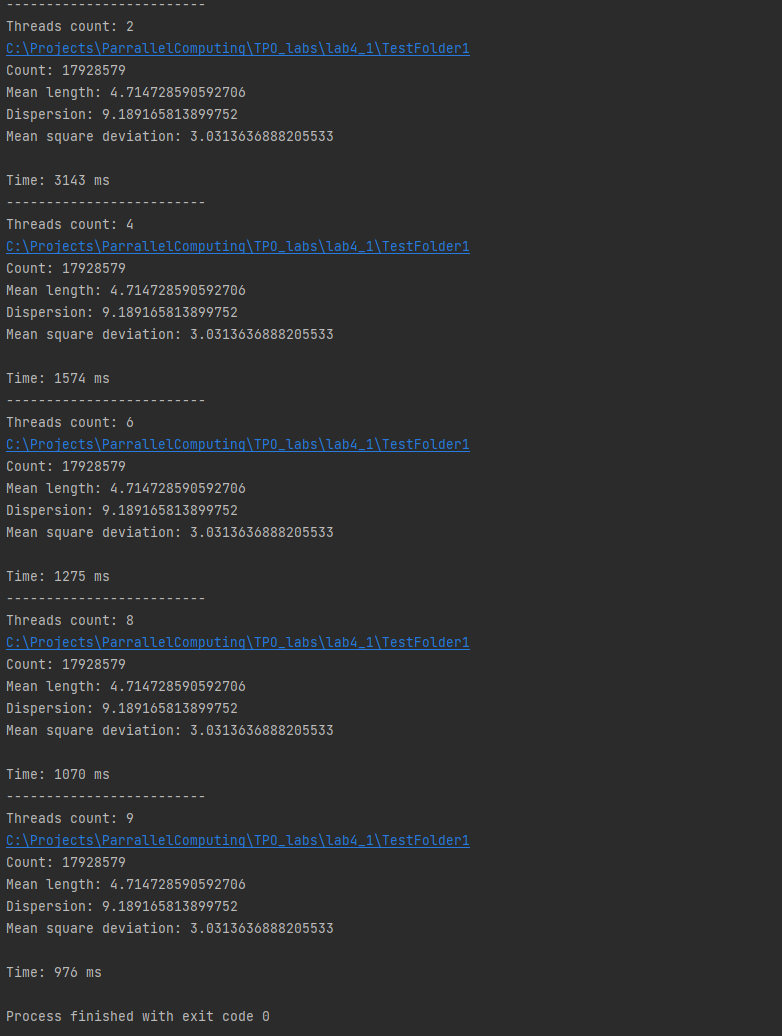
Маємо наступні результати:





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:



1. **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.

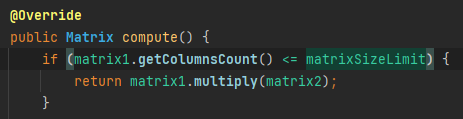
Для лабораторної роботи було створено 2 класа: FoxAlgorithmForkJoin та FoxAlgorithmTask.

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

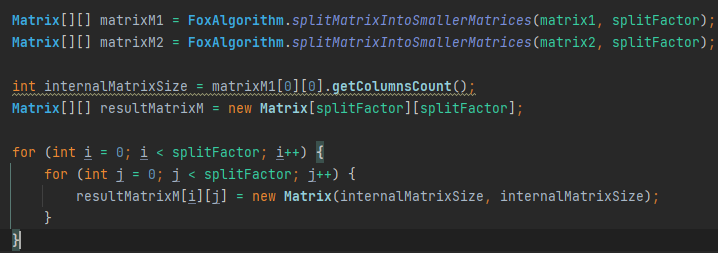


В свою чергу клас FoxAlgorithmTask наслідується від RecursiveTask<Matrix>. В класі міститься конструктор та метод compute().

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



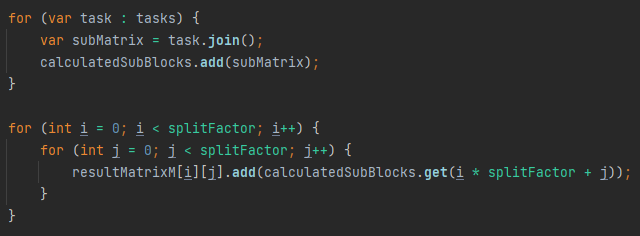
Якщо умова виходу з рекурсії не справджується, ми починаємо дії, аналогічні зі звичайним алгоритмом Фокса, а саме розбиття початкових матриць на менші матриці, створення результуючих матриць та їх ініціалізація. Єдиною різницею є те, що в даному випадку ми завжди розбиваємо на рівну кількість частин(в даному випадку splitFactor == 2)



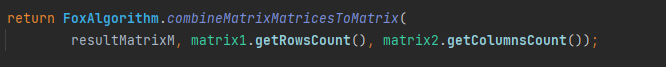
Далі ми запускаємо рекурсивно викликаємо FoxAlgorithmTask для підматриць



І після цього збираємо результати всіх викликів разом, формуємо результат



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



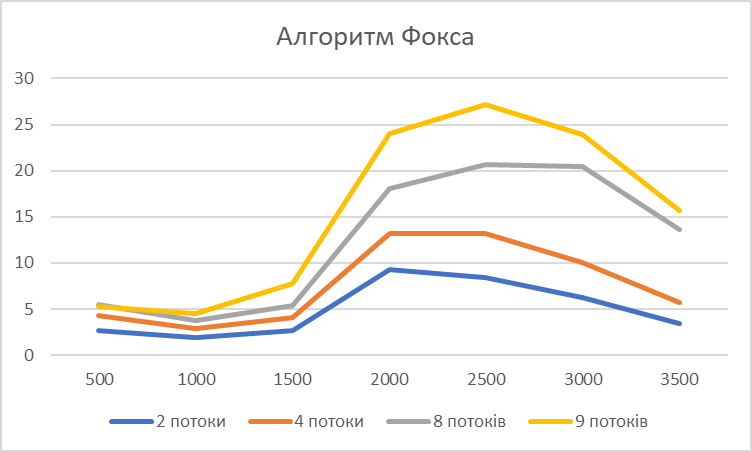
**Порівняння швидкодії алгоритму зі стандартною версією алгоритму Фокса**

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:

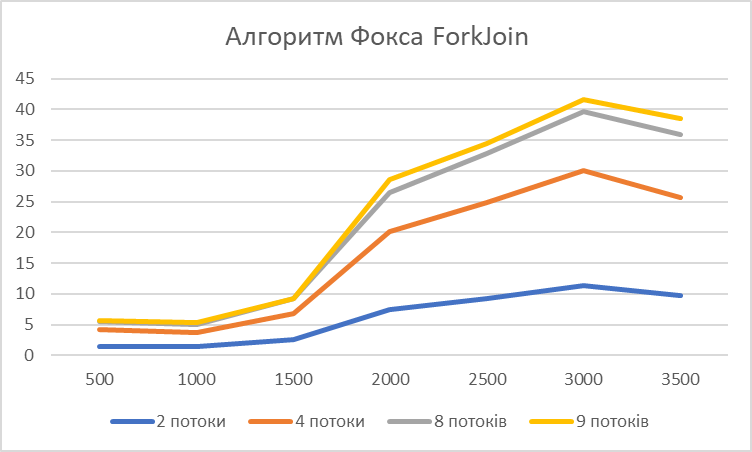
Алгоритм Фокса



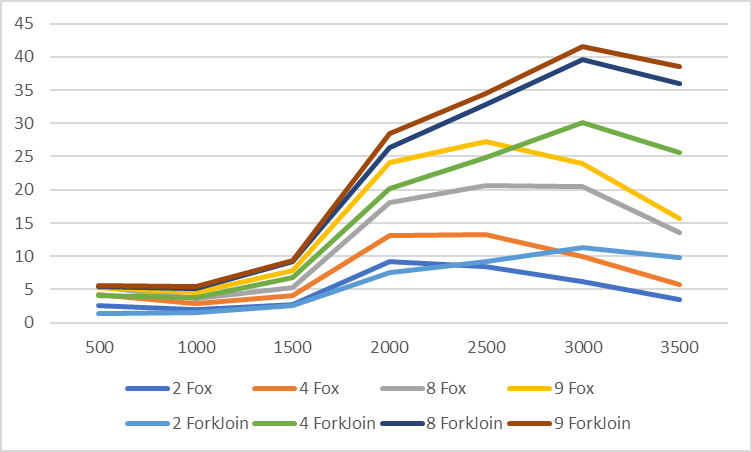


Алгоритм Фокса з використанням ForkJoinPool





Збірний результат

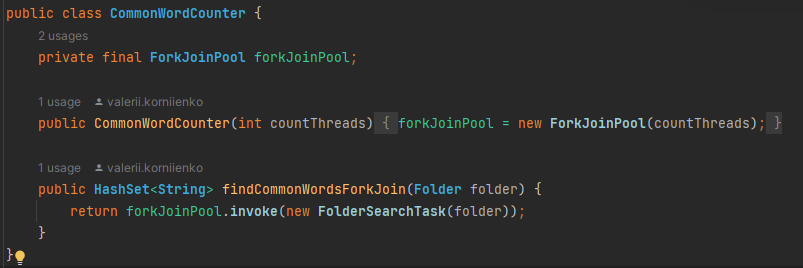


Як бачимо, модифікація алгоритму Фокса пройшла успішно і ми бачимо значний приріст в прискоренні алгоритму. Він особливо помітний при великих значеннях матриць, оскільки там відбувається велика кількість рекурсивних викликів, що і є сильною стороною ForkJoin. Оновлений алгоритм програє стандартному лише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

1. **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:



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

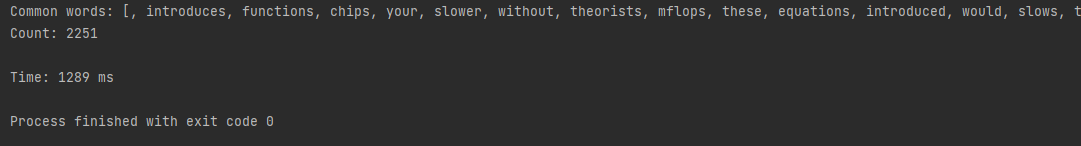
Клас FolderSearchTask має в собі єдиний метод compute(). Аналогічно до класу з такою ж назвою в заданні 4.1, цей клас спочатку рекурсивно викликає FolderSearchTask для всіх підпапок, потім ствоює та викликає завдання для всіх текстовиї файлів папки. Наприкінці своєї роботи метод збирає дані з усіх викликаних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



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:



Приклад роботи програми:

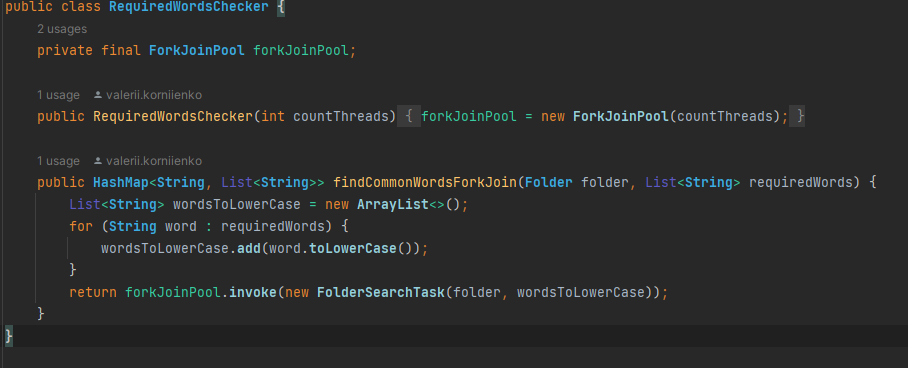


1. **Розробіть та реалізуйте алгоритм пошуку текстових документів, які відповідають заданим ключовим словам (належать до області «Інформаційні технології»), з використанням ForkJoinFramework.**

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

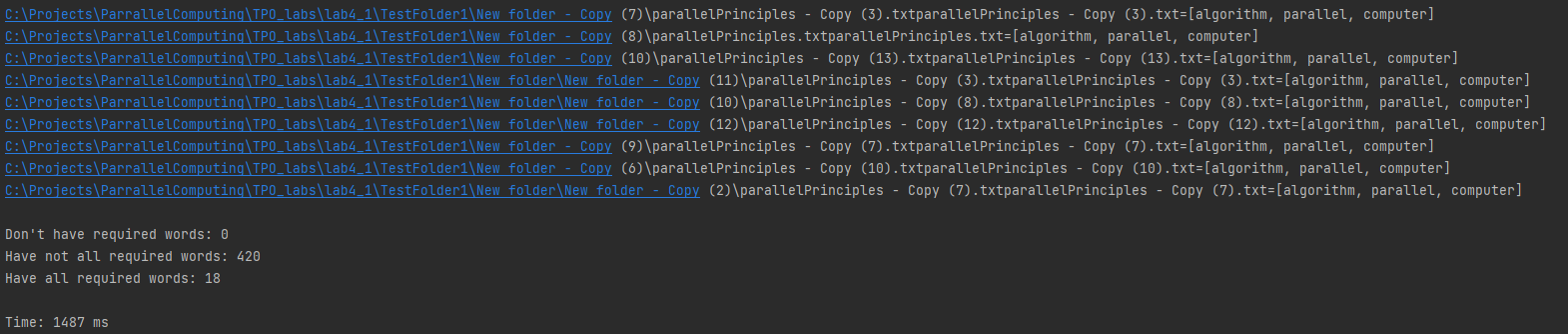


Також маємо клас FolderSearchTask extends RecursiveTask<HashMap<String, List<String>>>, головною задачею якого є запуск задач FolderSearchTask для всіх підпапок та TextFileSearchTask для всіх текстових файлів. Також даний клас агрегує результати всіх задач, які він запустив на виконання. 

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



Приклад результату роботи програми:



**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**;  
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;  
  
public class **Main** {  
 public static void main(**String**[] args) throws **IOException** {  
 **File** file = new **File**(**"C:\\Projects\\ParrallelComputing\\TPO\_labs\\lab4\_1/TestFolder"**);  
 **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**

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/TestFolder"**);  
 **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**

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**

import **java.io.File**;  
import **java.io.IOException**;  
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/TestFolder"**);  
 **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"**);  
 }  
}