Київський політехнічний інститут імені Ігоря Сікорського

Інститут прикладного системного аналізу

Кафедра Системного проектування

**Курсова робота**

# З дисципліни «Паралельні обчислення»

# На тему: «Створення інвертованого індексу»

Виконала:

Студентка 3-го курсу

Групи ДА-81

Козінцева А. О.

Варіант №13

Керівник:

Ас. Яременко В. С.

Київ – 2021

**Завдання:**

Завданням курсової роботи є розробка програм для побудови інвертованого індексу, а також – для користування індексом.

* При побудові варто зробити можливість пришвидшити час рішення, варіюючи кількість потоків.
* При користуванні індексом необхідно використовувати структуру даних з паралельним доступом, одночасно звертаючись до нього декількома потоками.
* При цьому, для отримання найвищих балів варто звертатись до даної структури з різних процесів, використовуючи мережеві сокети.

**Основні пункти:**

1. Вивчити теорію з глави 4 «Побудова індексу» [1].
2. Прочитати всі пункти завдання, зробити аналіз вимог та розробити модель системи для побудови індексу в паралельному середовищі, заклавши можливість варіювати кількість потоків (від 1 до N).
3. Завантажити вхідні текстові дані згідно до варіанту [2].
4. Створити проект Github (або у будь-якому іншому репозиторії) [3]. Назвою може бути наступна `course\_work\_parallel\_computing`.
5. Створити програмну реалізацію моделі з п.2 [4][5], правильно використовуючи систему контролю версій (бажано, щоб один commit був однією атомарною зміною в проекті); описувати кожен commit в одному стилі, наприклад як вказано тут [6].
6. Виконати тестування розробленої системи.
7. Зробити графічне порівняння часу роботи системи, задаючи різну кількість потоків (від 1 до N).
8. Перевірити правильність результату роботи послідовного та паралельних рішень.
9. Зробити форматування коду в одному стилі, наприклад [7][8].
10. Зробити файл з інструкцією як зібрати та запустити проект, додати його в репозиторій.
11. Зробити протокол з результатами роботи, що має містити інформацію з попередніх пунктів.

*Структура файлів текстових даних для обробки:*

1. aclImdb\test\neg – N=12500 файлів
2. aclImdb\test\pos – N=12500 файлів
3. aclImdb\train\neg – N=12500 файлів
4. aclImdb\train\pos – N=12500 файлів
5. aclImdb\train\unsup – N=50000 файлів

Згідно до варіанту, треба обрати по 250 файлів з 1-4 директорій, та 1000 файлів з директорії 5.

**Вибір файлів:**

В кожній директорії файли проіндексовані в порядку зростання.

Початковий індекс згідно до варіанту: N / 50 \* (V – 1)

Кінцевий індекс згідно до варіанту: N / 50 \* V

*Для варіанту №13:*

1. test\neg: [3000, 3250]
2. test\pos: [3000, 3250]
3. train\neg: [3000, 3250]
4. train\pos: [3000, 3250]
5. train\unsup: [12000, 13000]

**Теоретичні відомості**

**Інвертований індекс** — [структура даних](https://uk.wikipedia.org/wiki/%D0%A1%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%B0_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85), в якій для кожного слова колекції документів у відповідному списку перераховані всі документи в колекції — в яких воно зустрілося. Інвертований індекс використовується для пошуку за текстами.

Є два варіанти інвертованого індексу:

* індекс, який містить лише список документів для кожного слова,
* індекс, додатково включає позицію слова в кожному документі.

Опишемо, як вирішується завдання знаходження документів, в яких зустрічаються всі слова з пошукового запиту. При обробці однослівного пошукового запиту відповідь вже є в інвертованому індексі — достатньо взяти список, відповідний слову із запиту. При обробці багатослівного запиту беруться списки, відповідні кожному зі слів запиту і пересічні.

**Існує декілька способів для побудови інвертованого індексу:**

* Блочна індексація

Спочатку виконується прохід по колекції та збір усіх пар «термін-docID». Потім ці пари сортуються, причому в якості головного ключа використовується термін, а в якості вторинного ключа – ідентифікатор документу docID. На заключному етапі ідентифікатори документу для кожного терміну заносяться до інвертованого списку та обчислюются статичні характеристики, наприклад, частота терміна в документі (TF) та документна частота (DF). Для невеликих колекцій це можна здійснити в оперативній пам’яті. Використовується алгорим BSBI (blocked sort-based indexing).

* Однопрохідна індексація

Блочна індексація, заснована на сортуванні, має чудові можливості для масштабування, але для її реалізації кожному терміну потрібно поставити у відповідність ідентифікатор termID. Для дуже великих колекцій необхідна структура даних може не поміститися у пам’яті. Більш ефективним з точки зору масштабування є однопрохідна індексація в пам’яті (SPIMI – single-pass in-memory indexing). Цей алгоритм використовує терміни, а не їхні ідентифікатори, записує словник кожного блоку на диск, а потім для нового блоку починає створювати новий словник. За наявності достатньої кількості об’єму пам’яті на диску, за допомогою алгоритму SPIMI можна проіндексувати колекцію довільного розміру.

* Розподілена індексація

Формує розподілений індекс, розділений між декількома машинами, - або за термінами, або за документами.

* Динамічна індексація

Використовується у випадку, коли колекції часто змінюються, при цьому додаються нові документи, інші видаляються або оновлюються. Дана індексація періодично перебудовує індекс повторно.

* та інші.

**Практична частина**

**Принцип роботи програми:**

Спочатку проходимо по директоріях та зчитуємо шляхи до файлів та назви самих файлів у списки filePath та fileName відповідно. Після цього трансформуємо отримані списки до масивів sPath[], sName[], щоб з даними було зручніше працювати.

List<String> filePath = new ArrayList<>(); //список адрес усіх файлів  
List<String> fileName = new ArrayList<>(); //список назв усіх файлів  
  
File dirPath1 = new File("D:/PO/Kursova/datasets/test\_neg");  
File dirPath2 = new File("D:/PO/Kursova/datasets/test\_pos");  
File dirPath3 = new File("D:/PO/Kursova/datasets/train\_neg");  
File dirPath4 = new File("D:/PO/Kursova/datasets/train\_pos");  
File dirPath5 = new File("D:/PO/Kursova/datasets/train\_unsup");  
  
//Зчитуємо дані для файлів за відповідною адресою  
File files1[] = dirPath1.listFiles();  
for (int i = 0; i < files1.length; i++) {  
 filePath.add(files1[i].getPath());  
 fileName.add(files1[i].getName());  
}  
File files2[] = dirPath2.listFiles();  
for (int i = 0; i < files2.length; i++) {  
 filePath.add(files2[i].getPath());  
 fileName.add(files2[i].getName());  
}

**Для послідовного варіанту** алгоритму створюємо об’єкт класу IndexSample, який має 2 методи: buildIndex() та findWord().

* **buildIndex()** відповідає за створення хешмапи, у якій в якості ключів будуть зберігатися слова з файлів, а в якості значень – назви файлів, у яких ці слова зустрічаються.

public void buildIndex(String[] files, String[] names) {  
 for (int i = 0; i < files.length; i++) {  
 try (BufferedReader file = new BufferedReader(new FileReader(files[i]))) {  
 String line;  
 while ((line = file.readLine()) != null) {   
 String[] words = line.split("\\W+");   
 for (String word : words) {  
 word = word.toLowerCase();  
 if (!index.containsKey(word)) {   
 index.put(word, new HashSet<String>());   
 }  
 index.get(word).add(names[i]);   
 }  
 }  
 } catch (IOException e) {  
 System.*out*.println("File " + files[i] + " not found.");  
 }  
 }

Для побудови індексу зчитуємо кожен файл з масиву та записуємо результат у line. Розділяємо line на окремі слова методом split() та записуємо їх до масиву words[]. Перетворюємо слова так, щоб вони починалися з маленької літери. Якщо в хешмапі index ще немає такого слова, записуємо його до цієї хешмапи як ключ та створюємо новий HashSet() для запису назв файлів у якості значень до даного ключа. Відповідно, якщо такого ключа немає, записуємо і ключ, і назви файлів за відповідним ключем.

* **findWord()** відповідає за пошук у хешмапі назв файлів, у яких наявні вказані слова.

public void findWord (String phrase) {  
 String[] words = phrase.split("\\W+"); //розділяємо введений користувачем рядок на окремі слова  
 if (words.length > 1) {  
 HashSet<String> answer = new HashSet<String>(index.get(words[0].toLowerCase()));  
 for (String word : words) {  
 answer.retainAll(index.get(word)); //залишаємо лише ті файли, у яких є всі перелічені слова  
 }  
 if (answer.size() == 0) {  
 System.*out*.println("Not found");  
 } else {  
 System.*out*.println("Intersection of words: ");  
 for (String ans : answer) {  
 System.*out*.println("\t" + ans);  
 }  
 }  
 }  
 if (words.length >= 1) {  
 for (String word : words) {  
 System.*out*.println("Files for '" + word + "' are: " + index.get(word));  
 }  
 }  
}

До answer записуються ті назви файлів, які містять усі слова, введені користувачем. Спочатку додаємо до answer файли, які відповідають першому слову. Далі використовуємо функцію retainAll(), яка перевіряє, чи є такі файли за ключем наступного слова. Якщо є – залишає їх, а решту до answer не записує.

**У випадку паралельного алгоритму** було створено клас IndexParallel. В залежності від кількості потоків обробляється відповідна частина файлів. Наприклад, при threads=5 перший потік обробляє файли під індексом від 0 до 20 з масиву sPath[]. Другий – від 20 до 40 і т.д. За це відповідає наступна функція:

for (int i = start; i < finish; i++) { //обирається, які саме файли буде оброблено  
 try (BufferedReader file = new BufferedReader(new FileReader(arrayPath[i]))) {  
 String line;  
 while ((line = file.readLine()) != null) {  
 String[] words = line.split("\\W+");  
 for (String word : words) {  
 word = word.toLowerCase();  
 if (!tempInd.containsKey(word)) {  
 tempInd.put(word, new HashSet<String>());  
 }  
 tempInd.get(word).add(arrayName[i]);  
 }  
 }  
 } catch (IOException e) {  
 System.*out*.println("File " + arrayPath[i] + " not found.");  
 }  
}

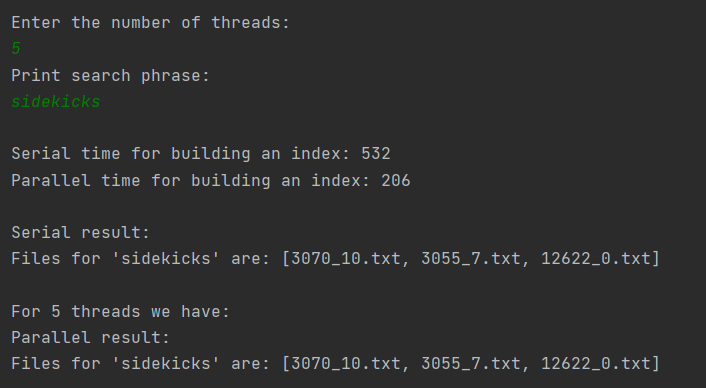
На відміну від послідовного алгоритму, тут створюється n хешмап (де n – кількість потоків). Після проведення паралельних обчислень в головному потоці отримані хешмапи об’єднуються в одну – parallelIndex – наступною функцією:

for (int i = 0; i < threads; i++) {  
 //об'єднуємо усі хешмапи в одну  
 for (Map.Entry<String, HashSet<String>> temp : ThreadArray[i].getResult().entrySet()) {  
 String key = temp.getKey();  
 HashSet<String> value = temp.getValue();  
 if (parallelIndex.containsKey(key)) {  
 parallelIndex.computeIfAbsent(key, w -> new HashSet<>()).addAll(value);  
 } else {  
 parallelIndex.put(key, value);  
 }  
 }  
}

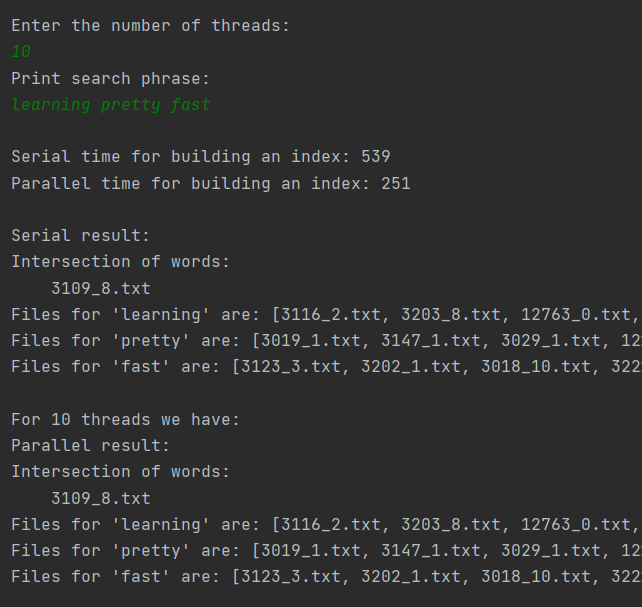
Де проходимо по усіх записах хештаблиць, отриманих кожним потоком, та в залежності від того чи є такий ключ у результуючій хешмапі parallelIndex додаємо записи, яких за цим ключем ще не було. Або ж створюється новий ключ та з тимчасової хешмапи переносяться значення безпосередньо до результуючої хешмапи parallelIndex.

**Результати роботи програми:**

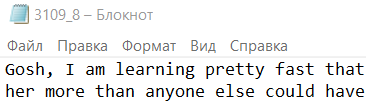
*Для одного слова:*

****

*Для декількох слів:*

**

*Перевірка:*

**

**Залежність швидкості роботи програми від кількості потоків**

|  |  |  |
| --- | --- | --- |
| **Кількість потоків** | **Час на побудову індексу** | **Кількість оброблених файлів одним потоком** |
| 1 | 519 | 2000 |
| 2 | 255 | 1000 |
| 5 | 209 | 400 |
| 10 | 216 | 200 |
| 15 | 257 | 133 |
| 20 | 289 | 100 |
| 25 | 291 | 80 |
| 50 | 293 | 40 |
| 100 | 343 | 20 |
| 250 | 459 | 8 |
| 500 | 475 | 4 |
| 1000 | 620 | 2 |
| 2000 | 657 | 1 |

**Графік залежності швидкості роботи програми**

**від кількості потоків**

**Інструкція для запуску проекту:**

1. Використовуючи Git Bash та посилання на репозиторій (Code -> HTTPS -> copy), ввести команду git clone та клонувати репозиторій:

git clone https://github.com/gannusia/course\_work\_parallel\_computing.git

2. Запустити проект у середовищі Intellij IDEA.

3. Скомпілювати проект, після чого у консолі з’явиться інструкція щодо подальших дій: ввести кількість потоків та фразу для пошуку.

4. Як результат користувач отримує:

* час, за який здійснюється побудова індексу у послідовному варіанті програми;
* час, за який здійснюється побудова індексу у паралельному варіанті програми;
* файли, у яких зустрічаються вказані користувачем слова.

**Лістинг**

**IndexParallel:**

import java.io.BufferedReader;  
import java.io.FileReader;  
import java.io.IOException;  
import java.util.HashMap;  
import java.util.HashSet;  
import java.util.Map;  
  
public class IndexParallel extends Thread {  
  
 String[] arrayPath; //список з адресами  
 String[] arrayName; //список з назвами файлів  
 int start; //початковий індекс для обробки  
 int finish; //кінцевий індекс для обробки  
  
 Map<String, HashSet<String>> tempInd = new HashMap<>(); //хешмапа для результату роботи кожного потоку  
  
 public IndexParallel(String[] arrayPath, String[] arrayName, int startRow, int finishRow) {  
 this.arrayPath = arrayPath;  
 this.arrayName = arrayName;  
 this.start = start;  
 this.finish = finish;  
 }  
  
 public Map<String, HashSet<String>> getResult() {  
 return tempInd;  
 }  
  
 @Override  
 public void run() {  
 for (int i = start; i < finish; i++) { //обирається, які саме файли буде оброблено  
 try (BufferedReader file = new BufferedReader(new FileReader(arrayPath[i]))) {  
 String line;  
 while ((line = file.readLine()) != null) {  
 String[] words = line.split("\\W+");  
 for (String word : words) {  
 word = word.toLowerCase();  
 if (!tempInd.containsKey(word)) {  
 tempInd.put(word, new HashSet<String>());  
 }  
 tempInd.get(word).add(arrayName[i]);  
 }  
 }  
 } catch (IOException e) {  
 System.*out*.println("File " + arrayPath[i] + " not found.");  
 }  
 }  
 }  
}

**IndexSample:**

import java.io.\*;  
import java.util.ArrayList;  
import java.util.HashMap;  
import java.util.Map;  
import java.util.HashSet;  
import java.util.List;  
  
public class IndexSample {  
  
 Map<String, HashSet<String>> index;  
  
 IndexSample() {  
 index = new HashMap<String, HashSet<String>>();  
 }  
  
 public void buildIndex(String[] files, String[] names) {  
 for (int i = 0; i < files.length; i++) {  
 try (BufferedReader file = new BufferedReader(new FileReader(files[i]))) { //відкриваємо для читання поточний файл  
 String line;  
 while ((line = file.readLine()) != null) { //зчитуємо вміст файлу у змінну line  
 String[] words = line.split("\\W+"); //розділяємо line на окремі слова та записуємо їх до масиву  
 for (String word : words) {  
 word = word.toLowerCase();  
 if (!index.containsKey(word)) { //якщо такого слова ще немає у хешмапі  
 index.put(word, new HashSet<String>()); //додаємо слово до хешмапи  
 }  
 index.get(word).add(names[i]); //додаємо назву файлу за відповідним ключем  
 }  
 }  
 } catch (IOException e) {  
 System.*out*.println("File " + files[i] + " not found.");  
 }  
 }  
 }  
  
 public void findWord (String phrase) {  
 String[] words = phrase.split("\\W+"); //розділяємо введений користувачем рядок на окремі слова  
 if (words.length > 1) {  
 HashSet<String> answer = new HashSet<String>(index.get(words[0].toLowerCase()));  
 for (String word : words) {  
 answer.retainAll(index.get(word)); //залишаємо лише ті файли, у яких є всі перелічені слова  
 }  
 if (answer.size() == 0) {  
 System.*out*.println("Not found");  
 } else {  
 System.*out*.println("Intersection of words: ");  
 for (String ans : answer) {  
 System.*out*.println("\t" + ans);  
 }  
 }  
 }  
 if (words.length >= 1) {  
 for (String word : words) {  
 System.*out*.println("Files for '" + word + "' are: " + index.get(word));  
 }  
 }  
 }  
  
 public static void main (String[] args) throws IOException, InterruptedException {  
 //проходимо по усіх файлах та додаємо їх до списків  
 List<String> filePath = new ArrayList<>(); //список адрес усіх файлів  
 List<String> fileName = new ArrayList<>(); //список назв усіх файлів  
  
 File dirPath1 = new File("D:/PO/Kursova/datasets/test\_neg");  
 File dirPath2 = new File("D:/PO/Kursova/datasets/test\_pos");  
 File dirPath3 = new File("D:/PO/Kursova/datasets/train\_neg");  
 File dirPath4 = new File("D:/PO/Kursova/datasets/train\_pos");  
 File dirPath5 = new File("D:/PO/Kursova/datasets/train\_unsup");  
  
 //Зчитуємо дані для файлів за відповідною адресою  
 File files1[] = dirPath1.listFiles();  
 for (int i = 0; i < files1.length; i++) {  
 filePath.add(files1[i].getPath());  
 fileName.add(files1[i].getName());  
 //System.out.println(files1[i].getName());  
 }  
 File files2[] = dirPath2.listFiles();  
 for (int i = 0; i < files2.length; i++) {  
 filePath.add(files2[i].getPath());  
 fileName.add(files2[i].getName());  
 }  
 File files3[] = dirPath3.listFiles();  
 for (int i = 0; i < files3.length; i++) {  
 filePath.add(files3[i].getPath());  
 fileName.add(files3[i].getName());  
 }  
 File files4[] = dirPath4.listFiles();  
 for (int i = 0; i < files4.length; i++) {  
 filePath.add(files4[i].getPath());  
 fileName.add(files4[i].getName());  
 }  
 File files5[] = dirPath5.listFiles();  
 for (int i = 0; i < files5.length; i++) {  
 filePath.add(files5[i].getPath());  
 fileName.add(files5[i].getName());  
 }  
  
 //перетворюємо отримані списки у масиви  
 String[] sPath = new String[filePath.size()];  
 String[] sName = new String[fileName.size()];  
 sPath = filePath.toArray(sPath);  
 sName = fileName.toArray(sName);  
  
 //користувач задає кількість потоків  
 System.*out*.println("\nEnter the number of threads: ");  
 BufferedReader readerTwo = new BufferedReader(new InputStreamReader(System.*in*));  
 String threadsStr = readerTwo.readLine();  
 int threads = Integer.*parseInt*(threadsStr);  
  
 System.*out*.println("Print search phrase: ");  
 BufferedReader in = new BufferedReader(new InputStreamReader(System.*in*));  
 String phrase = in.readLine();  
  
  
 //-------------------| Послідовний алгоритм |----------------------//  
  
 long timeSerialStart = System.*currentTimeMillis*();  
 IndexSample index = new IndexSample();  
 index.buildIndex(sPath, sName);  
 long timeSerialFinish = System.*currentTimeMillis*() - timeSerialStart;  
  
  
 //-------------------| Паралельний алгоритм |---------------------//  
  
 long timeParallelStart = System.*currentTimeMillis*();  
 Map<String, HashSet<String>> parallelIndex = new HashMap<>();  
 IndexParallel ThreadArray[] = new IndexParallel[threads];  
  
 for (int i = 0; i < threads; i++) { //розподіляємо роботу між потоками  
 ThreadArray[i] = new IndexParallel(sPath, sName,sPath.length / threads \* i, i == (threads - 1) ? sPath.length : sPath.length / threads \* (i + 1));  
 ThreadArray[i].start();  
 }  
 for (int i = 0; i < threads; i++) {  
 ThreadArray[i].join();  
 }  
 for (int i = 0; i < threads; i++) {  
 //об'єднуємо усі хешмапи в одну  
 for (Map.Entry<String, HashSet<String>> temp : ThreadArray[i].getResult().entrySet()) { //перебір елементів хештаблиці  
 String key = temp.getKey(); //зчитуємо ключ для цього запису в Map  
 HashSet<String> value = temp.getValue(); //зчитуємо значення для цього запису в Map  
 if (parallelIndex.containsKey(key)) {  
 parallelIndex.computeIfAbsent(key, w -> new HashSet<>()).addAll(value); //додаємо значення за вказаним ключем  
 } else {  
 parallelIndex.put(key, value);  
 }  
 }  
 }  
 long timeParallelFinish = System.*currentTimeMillis*() - timeParallelStart;  
  
  
 //---------------------| Виведення результатів |----------------------//  
  
 System.*out*.println("\nSerial time for building an index: "+ (timeSerialFinish));  
 System.*out*.println("Parallel time for building an index: "+ (timeParallelFinish));  
  
 System.*out*.println("\nSerial result: ");  
 index.findWord(phrase);  
  
 System.*out*.println("\nFor " + threads + " threads we have: ");  
 System.*out*.println("Parallel result: ");  
 String[] words = phrase.split("\\W+"); //розділяємо введений користувачем рядок на окремі слова  
 if (words.length > 1) {  
 HashSet<String> answer = new HashSet<String>(parallelIndex.get(words[0].toLowerCase()));  
 for (String word : words) {  
 answer.retainAll(parallelIndex.get(word)); //залишаємо лише ті файли, у яких є всі перелічені слова  
 }  
 if (answer.size() == 0) {  
 System.*out*.println("Not found");  
 } else {  
 System.*out*.println("Intersection of words: ");  
 for (String ans : answer) {  
 System.*out*.println("\t" + ans);  
 }  
 }  
 }  
 if (words.length >= 1) {  
 for (String word : words) {  
 System.*out*.println("Files for '" + word + "' are: " + parallelIndex.get(word));  
 }  
 }  
 }  
}

**Висновок:**

У курсовій роботі було розроблено програму на мові Java для побудови інвертованого індексу. Дана програма будує індекс за допомогою послідовного та паралельного алгоритмів. На основі результатів, отриманих при роботі програми, було з’ясовано, що більша кількість потоків дозволяє швидше будувати індекс, однак при занадто великій їхній кількості витрачається багато часу на створення цих потоків, що потребує навіть більше часу ніж саме виконання програми у послідовному порядку. Наприклад, маємо 2000 записів. Два потоки зможуть паралельно обробити по 1000 записів кожен, 5 потоків – по 400, 10 – по 200. Однак при кількості потоків рівній, наприклад, 20 і більше починається спад ефективності, оскільки час на створення потоків вже є суттєвішим, а кількість файлів, які потік обробляє, могла б бути більшою для кращої ефективності.

**Література**

1. Маннинг К. Д. Введение в информационный поиск / К. Д. Маннинг, П. Рагхаван, Х. Шютце., 2014. – 528 с.
2. <http://ai.stanford.edu/~amaas/data/sentiment/>
3. <https://github.com/>
4. Goetz B. Java concurrency in practice / Brian Goetz., 2006. – 386 с.
5. Уильямс Э. Параллельное программирование на С++ в действии / Энтони Уильямс., 2012.
6. <https://www.conventionalcommits.org/en/v1.0.0/>
7. <https://google.github.io/styleguide/>
8. Мартин Р. Чистый код. Создание, анализ и рефакторинг / Роберт Мартин., 2012. – 464 с. – (Библиотека программиста).