Sequential and parallel implementation of ngrams in Java.

Armand Palla: 7042287

Abstract:

We present an algorithm for computing and estimating the occurrence of ngrams(**bigrams and trigrams**) in two different implementations: sequential and parallel. The chosen language is JAVA for both implementations and the system makes use of parallelisms in computation via the JAVA thread programming model. We study the different implementation through computational time and speed up, vary depending on number of threads in the parallel version. The tests were performed on intel i5 dual-core and the results show how the parallel version differ from the sequential version.

Introduction:

People read texts. The texts consist of sentences and also sentences consist of words. Human beings can understand linguistic structures and their meanings easily, but machines are not successful enough on natural language comprehension yet. So, we try to teach some languages to machines like we do for an elementary school kid. This is the main concept: words are basic, meaningful elements with the ability to represent a different meaning when they are in a sentence. By this point, we keep in mind that sometimes word groups provide more benefits than only one word when explaining the meaning.
Here is our sentence "**I read a book about the history of America.**"
The machine wants to get the meaning of the sentence by separating it into small pieces. How should it do that?
1. It can regard words one by one. This is ***unigram***; each word is a gram.
"I", "read", "a", "book", "about", "the", "history", "of", "America"
2. It can regard words two at a time. This is ***bigram*** (*digram*); each two adjacent words create a bigram.
"I read", "read a", "a book", "book about", "about the", "the history", "history of", "of America"
3. It can regard words three at a time. This is ***trigram****;* each three adjacent words create a trigram.

In our example we have done the same thing ,using different sizes of texts,but ngrams are used by separating it into **letters** and not into words.For simplicity, I have used bigrams and trigrams examples in order to implement the main goal.

Text and computation:

The main goal is to estimate occurrences of bigrams and trigrams in a certain text. We have used a wikipedia text that talks about **Gutenberg project of Australia**.

To compute bigrams and trigrams we have formulated a pseudo-algorithm in two different versions: sequential and parallel.

Sequential:

- n: number of grams, 2 for bigrams, 3 for trigrams.
- file: contains the characters from the text file.
- file.length: represents the length of the file,number of characters in the file.
- key: a variable of type string with length equal to n.

Let's see now the pseudo code of the sequential version:

```
Data: n, file
1.  for i = 0 to file.length-n+1 do
2.      key = "";
3.      for j = 0 to n - 1 do
4.          key = key + file[i+j];
5.      end
6.  end
```

Parallel:

The parallel version has the same structure but uses more information, depending on how many threads we use:

- id: contains the thread ID.
- k: dimension of text for each thread, calculated as floor(fileLen/realThreads).
- start: start character, calculated as (i *k).
- stop: end character, calculated as (i + 1) * k + ((n - 1) -1).

```
Data:   i =idthreads,
        n = 2(bigrams) or 3(trigrams),
        start = (i * k), stop = (i + 1) * k + ((n - 1) -1),
        file = text
for i = this.start + this.n – 1 to this.stop do
    key="";
    for j = this.n – 1 downto 0 do
        key = key + this.file[i-j];
    end
end
```

Implementation:

The sequential approach:

Firstl,we will introduce with the the sequential approach to provide a better understanding for the parallel one. Before compute bigrams and trigrams, we have to read the text to analyse and replace opportunely those characters defined invalid, to prevent errors in bigrams and trigrams composition. The function implemented with this behavior is called readText().

```java
public static char[] readText() {
    Path path = Paths.get("/Users/Armando/eclipse-workspace/bigrams_trigrams/src/1-text50KB.txt");

    try {
        //Read all rows from a file as a Stream.
        Stream<String> rows = Files.lines(path);

        // Returns a Collector that concatenates the input elements into a String and converts this string to a new character array.
        char[] file = (rows.collect(Collectors.joining())).replaceAll("[ ;:&'(),.]", "").toCharArray();

        for(int i = 0; i < file.length - 1; ++i) {
            if (Character.isUpperCase(file[i])) {
                file[i] = Character.toLowerCase(file[i]);
            }
        }
        return file;
    }

    catch (IOException e) {
        System.out.println(e);
        System.exit(1); //bad command-line, can't find file, unsuccessful termination.
        return null;
    }
} // public static char[]
```

In the algorithm we use some function of *collectors* and *string* library:

- replaceAll(): change specified character to another.
- toCharArray(): copies each character in a string to a character array.
- isUpperCase(): boolean variable that tells us if the charater is Upper case.
- toLowerCase(): change the character from Upper case to Lower case.
- Collectors.joining(): Returns a Collector that concatenates the input elements into a String, in encounter order.

Data Structure:

The selection of ideal data structure is not quite easy. There i salso need to account the type of data,which will be stored simultaneously with the data structure concept. *HashMaps* are the selected data structure to store bigrams and trigrams . Reason of this choice is that a HashMap store items in “key/value” pairs and we can accesss them by an index of another type (eg. a String).One object is used as a key(index) to another object(value).A HashMap is similar with a hashtable but it is unsynchronized,it means that it can’t be shared between many threads without proper synchronization code. HashMap is generally preferd over HashTable if thread synchronization is not needed and HashMaps provide the constant time performance for the basic operations such as get() and put() for large sets. Hashmaps make no guarantees as to the order of the map.

Computation:

Bigrams and trigrams are calculated in the body of function ngrams(). This function gets as input parameters:

- int n: identifies bigram or trigram.
- char[] file: as the text to analyse.

The function behavior follows the computation for sequential version and returns the HashMap with bigrams or trigrams calculated.

```
36⊖    public static HashMap<String, Integer> ngrams(int n, char[] file) {
37         HashMap<String, Integer> hashMap = new HashMap();
38
39         for(int i = 0; i < file.length - n + 1; ++i) {
40
41             //A string buffer is like a String, but can be modified.String buffers are safe for use by multiple threa
42             StringBuffer buffer = new StringBuffer();
43             //StringBuilder builder = new StringBuilder();
44             for(int j = 0; j < n; ++j) {
45                 buffer.append(file[i + j]);
46             }
47
48             String key = buffer.toString();
49             //Control if the key is added one time in the hashMap
50             if (!hashMap.containsKey(key)) {
51                 hashMap.put(key, 1);
52             }
53             //if it is added before increment the value with one
54             else if (hashMap.containsKey(key)) {
55                 hashMap.put(key, hashMap.get(key) + 1);
56             }
57         }
58
59         return hashMap;
```

The parallel approach:

For the reconstruction of this the idea to parallelize the sequential behavior is to divide the text in as many parts as are the thread istances and leave the search of bigrams and trigrams on a single part to a single thread. In this way we'll see in results how the computational times are significantly reduced compared to sequential times.

Java Thread:

The Concurrency API introduces the concept of an ExecutorService as a higher level replacement for working with threads directly. Executors are capable of running asynchronous tasks and typically manage a pool of threads, so we don't have to create new threads manually. All threads of the internal pool will be reused, so we can run as many concurrent tasks as we want throughout the life-cycle of our application with a single executor service. The features used of this package are:

- Future: is used to represent the result of an asynchronous operation. It comes with methods for checking if the asynchronous operation is completed or not, getting the computed result.
- Executor: an interface that represents an object which executes provided tasks. One point to note here is that Executor does not strictly require the task

execution to be asynchronous. In the simplest case, an executor can invoke the submitted task instantly in the invoking thread.

- ExecutorService: is a complete solution for asynchronous processing. It manages an inmemory queue and schedules submitted tasks based on thread availability. To use ExecutorService, we need to create one Runnable or Callable class.
- CompletionService is service that decouples the production of new asynchronous tasks from the consumption of the results of completed tasks. Instead of trying to find out which task has completed (to get the results), it just asks the CompletionService instance to return the results as they become available.

Data Structure:

To read and replace invalid characters is used the same function readText() of the sequential version. We have used ConcurrentHashMap. It allows concurrent modifications of the Map from several threads without the need to block them. The difference with the synchronizedMap (java.util.Collections) is that instead of needing to lock the whole structure when making a change, it's only necessary to lock the bucket that's being altered. ConcurrentHashMap class is thread-safe,multiple threads can operate on a single object without any compilations. The object is divided into a number of segments according to the concurrency level.The underlined data structure for ConcurrentHashMap is HashTable. It stores the data in (Key, Value) pairs. To access a value one must know its key.

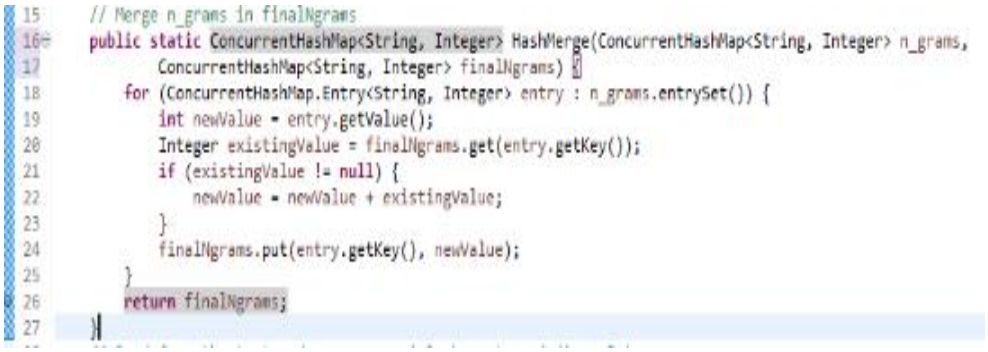
Computation:

• The first step is to declare a ParallelThread class which implements Callable because threads must have a return value, as in our case a ConcurrentHashMap, and because threads are called from an ExecutorService object that requires an Callable class.

• Implement the call method as described that allows to compute bigrams or trigrams and create the HashMap.

• Implement a HashMerge function that merges the ConcurrentHashMaps returned from different threads. This function adds new bigrams or trigrams, or updates the occurrences.

• Instantiate a Future array and an ExecutorService specifying the thread-pool size. Once the executor is created, we can use it to submit the compute method and get the results thanks to the Future method .get().



```
// Merge n_grams in finalNgrams
public static ConcurrentHashMap<String, Integer> HashMerge(ConcurrentHashMap<String, Integer> n_grams,
        ConcurrentHashMap<String, Integer> finalNgrams) {
    for (ConcurrentHashMap.Entry<String, Integer> entry : n_grams.entrySet()) {
        int newValue = entry.getValue();
        Integer existingValue = finalNgrams.get(entry.getKey());
        if (existingValue != null) {
            newValue = newValue + existingValue;
        }
        finalNgrams.put(entry.getKey(), newValue);
    }
    return finalNgrams;
}
```

Results:

In the tests of the application, we have studied the behavior of SpeedUp.

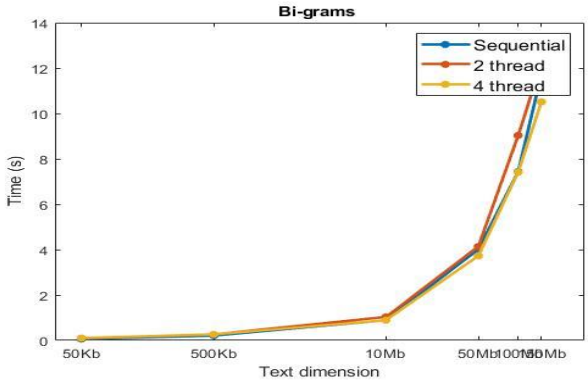
$$Sp = ts / tp$$

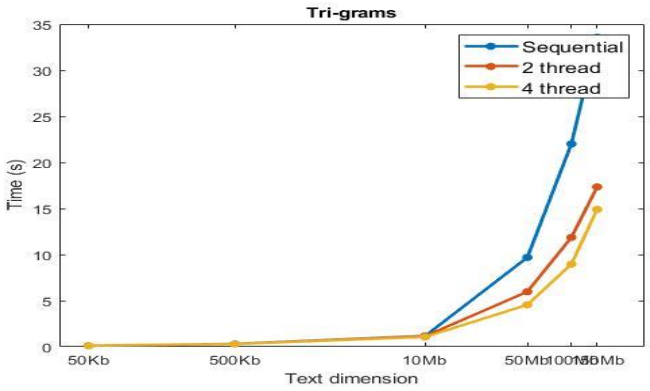
This behavior is studied increasing the number of threads, taking in consideration the parallel program, and the length of the text.

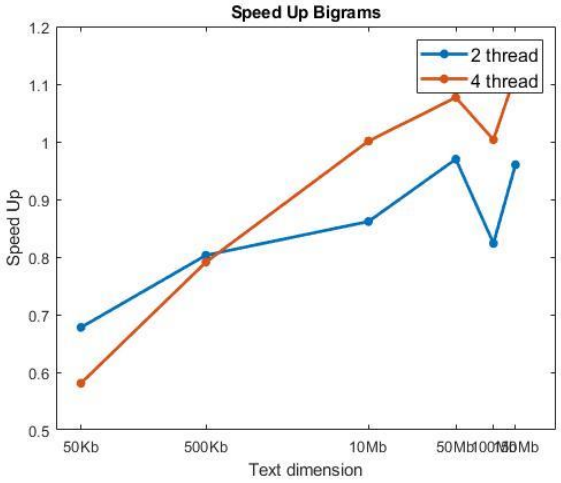
The results of this program depends on two main things:

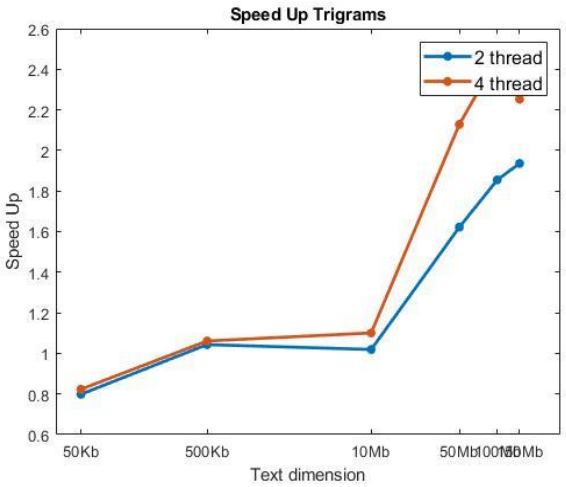
- Number of threads: 2 or 4
- Size of files: 50KB, 500KB, 10MB, 50MB, 100MB, 150MB

For every thread the computational time has been collected so many times and averaged. As shown in the figures, the parallel approach does not bring a substantial improvement in terms of computational time for the short texts; indeed, increasing the number of threads we have noticed that SpeedUp steeply decreased towards zero.









Conclusions:

For smaller size of texts the speed up in bigrams is more higher with 2 threads rather than 4 threads.

For bigger size of texts the speed up in bigrams is more higher with 4 threads rather than 2 threads.

For trigrams the speed up for smaller size of texts, related to threads is approximately the same .

For larger sizd of texts the speed up is more higher with 4 threads rather than threads.

So, when we increase the size of text it is more valuable to use more threads. The speed up will reach the peak(in terms of performance) when the number of threads equal to the number of computer's core.