PC-2019/20 Bigrams And Trigrams

Lorenzo Pratesi Mariti
Matricola - 7037171
lorenzo.pratesi@stud.unifi.it

Abstract

This paper presents an algorithm to compute and estimate occurrences of Bigrams and Trigrams in two different implementations: sequential and parallel. Particular attention will be paid to the study of the two different implementations through computational time and speed up, vary depending on number of threads in the parallel version. The chosen language is Java for the sequential version and the thread introduction for the parallel version. The tests were performed on intel i7 hexa core and the results show how the parallel version nearly halves the sequential times.

Future Distribution Permission

1. Introduction

Bigrams and Trigrams are used in one of the most successful language models for speech recognition. They are a special case of N-grams. A N-gram is a sequence of N adjacent elements from a string of tokens, which are typically letters, syllables, or words. Bigrams and Trigrams are widely used in statistical ”natural language” processing, and most significantly play a vital role in Hidden Markov models. Applications abound, but include context-sensitive spell checking, voice-to-text systems and grammar checking. Bigram analysis typically uses a corpus of text to learn the probability of various word pairs, and these probabilities are later used in recognition. However, in this project we are only interested in the data collection phase of bigram usage.
In this paper we have analyzed the frequency of letters related to Bigrams and Trigrams using both serial and parallel approaches. Finally, we have compared these two applications observing the execution time by varying the length of the text, expressed in Megabyte.

1.1. Main goal

The main goal is to compute and estimate occurrences of Bigrams and Trigrams in a certain text. More in general:

Sample sequence
`Parallel`
unigram
`P-a-r-a-l-l-e-l`
bigram
`Pa-ar-ra-al-ll-le-el`
trigram
`Par-ara-ral-all-lle-lel`
4-gram sequence
`Para-aral-rall-alle-llel`

Table 1. N-gram characters examples sequences

Table 1 shows an example sequence and the corresponding *unigram*, *bigram*, *trigram* and *4-gram* sequences.

1.2. Computation of N-grams

To compute bigrams and trigrams we have formulated two ways, in particular depending on the version to implement. The sequential version has 2 types of data:

- `fileString`: contains a list of character from the txt file

- `n`: number of grams, 2 for bigram and 3 for trigram

Algorithm 1: Sequential version

```
Input: fileString, n
Function computeNGrams ($fileString, n$):
    for $i$ = 0 to fileString.length-n do
        ngram = ””
        for $j$ = 0 to n-1 do
            ngram = ngram + fileString[i+j]
        end
    end
```

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 txt for each thread, calculated as *floor(text.length / nThread)*
- `start`: start character, calculated as *(k * i)*
- `stop`: end character, calculated as *((i + 1) * k) + (n - 1) -1)*

Algorithm 2: Parallel version

```
Input: id, k, n, start, stop, n, fileString
Function computeNGrams ($id, k, n, start, stop, n, fileString$):
    for $i$ = this.start to (this.stop - this.n+1) do
        ngram = ””
        for $j$ = 0 to this.n-1 do
            ngram = ngram + filestring[i+j]
        end
    end
```

2. Implementation

Next up are shown the sequential and parallel implementation, using Java language. For the parallel version we also use respectively Java Thread.

2.1. The sequential approach

First of all, we will introduce the sequential approach to provide a better understanding of the parallel one.

2.1.1 Data Processing

For simplicity, the calculation process only accepts text strings without special characters and white spaces, therefore, before compute bigrams and trigrams, the input text is been analyzed and cleaned up. This operation is also done to prevent computation errors in bigrams and trigrams composition. The function implemented with this behavior is called *readTextFromFile()* from *Utils* class. In the algorithm we use some function of collectors and string library:

```java
public static List<Character> readTextFromFile() {
    final Path path = Paths.get(”Your path”);

    try (Stream<String> fileStream = Files.lines(path))
     {
        return fileStream
            .flatMap(Utils::trimAndMapToChars)
            .collect(Collectors.toList());

    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }
}

private static Stream<Character> trimAndMapToChars(
     String s) {
    return s.toLowerCase()
        .replaceAll(”[^a–zA–Z]”, ””)
        .chars()
        .mapToObj(c –> (char) c);
}
```

Listing 1. Read text string from file.

2.1.2 Data Structure

The selection of ideal data structure is not quite easy task. Mainly there is also 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 HashMaps provide a constant time performance for the basic operations such as *get()* and *put()* for large sets (*O(n)* where *n* is the number of elements). HashMap is a part of Java's collection since Java 1.2. It provides the basic implementation of the Map interface of Java.

`Class HashMap<K,V>`

```
java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.HashMap<K,V>
```

It stores the data in (Key, Value) pairs. To access a value one must know its key. Hashmaps make no guarantees as to the order of the map. Order for our problem is not required.

2.1.3 Computation

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

- `List<Character> fileString`: the text to analyse.
- `int n`: identifies bigram or trigram.

The function behavior follows the computation for sequen- tial version explained in 1.2, and returns the HashMap with bigrams or trigrams calculated.

```java
public Map<String, Integer> computeNGrams(List<Character> fileString, int n) {
    Map<String, Integer> hashMap = new HashMap<>();

    for (int i = 0; i < fileString.size()-n+1; ++i) {
        StringBuilder builder = new StringBuilder();

        for (int j = 0; j < n; ++j) {
            builder.append(fileString.get(i + j));
        }

        String key = builder.toString();
        if (!hashMap.containsKey(key)) {
            hashMap.put(builder.toString(), 1);
        } else if (hashMap.containsKey(key)) {
            hashMap.put(builder.toString(),
                        hashMap.get(key) + 1);
        }
    }
    return hashMap;
}
```

Listing 2. Compute ngrams from a given list of characters.

2.2. The parallel approach

For the reconstruction of this just described serial code into a parallel one, the main idea is to parallelize the sequential behavior, the text it's been divided in as many parts as are the thread instances 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.

2.2.1 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 under the hood for revenant tasks, 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, etc.
- *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 in-memory queue and schedules submitted tasks based on thread avail- ability. 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.

2.2.2 Data preprocessing

To read and replace invalid characters is used the same function *readTextFromFile()* of the sequential version.

2.2.3 Data structure

For our needs, 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. It is basically lock-free on reads.

```
Class ConcurrentHashMap<K,V>
    java.lang.Object
        java.util.AbstractMap<K,V>
            java.util.concurrent.
                ConcurrentHashMap<K,V>
```

It stores the data in (Key, Value) pairs. To access a value one must know its key.

2.3. 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 in 1.2 (Parallel version) 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().

```
private void hashMerge(ConcurrentMap<String, Integer>
        nGrams, ConcurrentMap<String, Integer> finalGrams)
        {
    nGrams.forEach((k, v) -> finalGrams.merge(k, v,
        Integer::sum));

}
```

Listing 3. Merging hashmaps.

3. Experimental Results

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

$$S_p = \frac{ts}{tp}$$

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

Computer specifics

- 6-Core Intel Core i7
- 16 GB RAM
- SSD storage

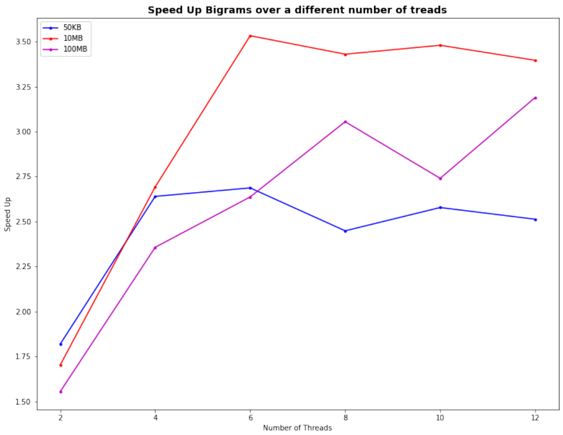
The corpus of test programs consisted of a Java benchmarks suite, the Java Microbenchmark Harness Suite (JMH). There are several benefits of using this approach, one of those is that JVM performs all sorts of optimizations internally, such as dead code elimination or loop unrolling, so we can focus only on the real behavior of the application test. Another important benefits is the availability of using a set of parameters with range values, this allows you to use conditions that may appear differently during an actual program run.
With this tests case we have set two parameters:

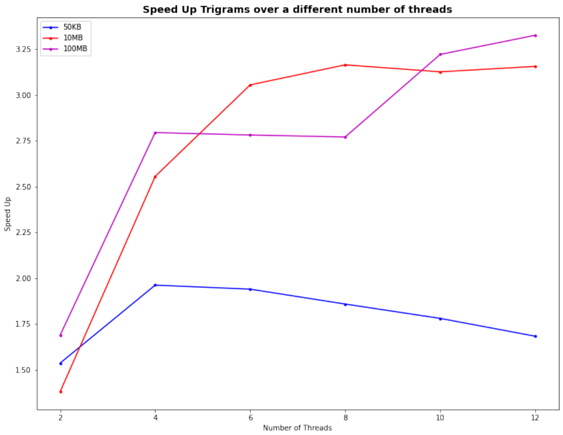
- *numOfThreads*: 2, 4, 6, 8, 10 and 12 (for parallel version)
- *files*: 50KB, 500KB, 10MB, 50MB, 100MB, 150MB

Before starting each test, JMH will do five runs of the code snippet to warm up the environment and initialize it. After doing that, executes the test for five times and finally collect the statistics needed.

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.

Increasing the text size, the scenario changes completely. In the tests done in a medium length text and in a long length text SpeedUp has reached

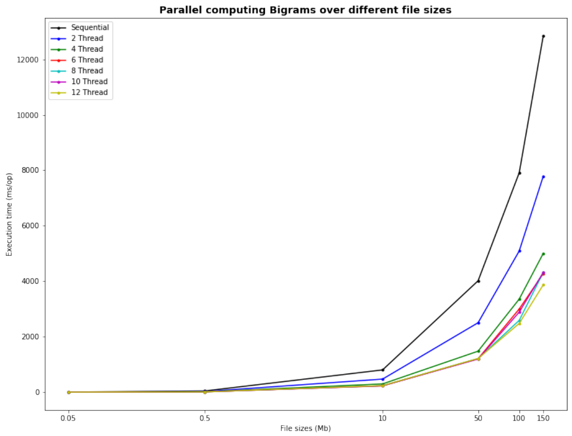


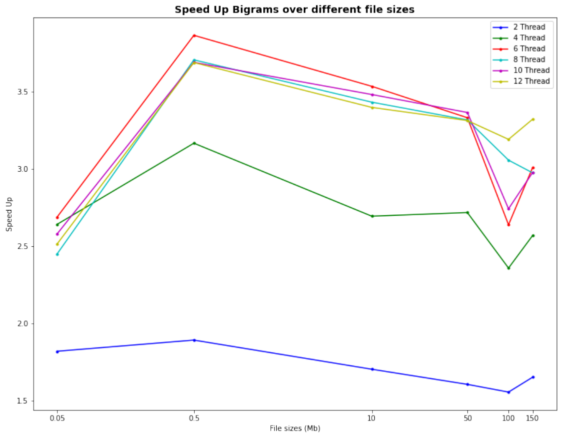


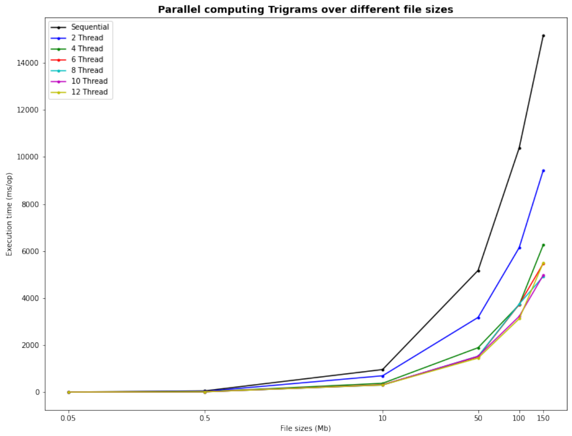
the peak, in terms of performance, when we have used a number of thread nearly equal to the number of computer's core (6 core), and then has decreased its value again, with an ondulatory behavior, as the number of threads has got larger. However, in these cases it could be possible that SpeedUp reaches a new peak greater than before with a number of threads higher than the number of computer's core: if it exists, this new peak usually has a value nearly the one found with the first peak. If we take in consideration a even higher number of threads, then we go back to the sequential time and, in some cases, we exceed this time.

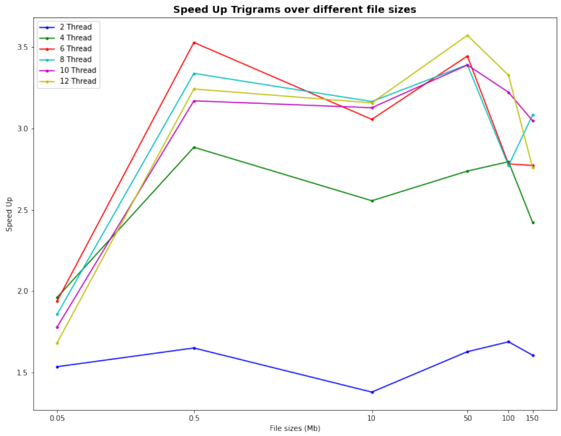
References

[1] n-gram - Wikipedia. `https://en.wikipedia.org/wiki/N-gram`


Parallel computing Bigrams over different file sizes
Sequential
2 Thread
4 Thread
6 Thread
8 Thread
10 Thread
12 Thread
Execution time (ms/op)
File sizes (Mb)


Speed Up Bigrams over different file sizes
2 Thread
4 Thread
6 Thread
8 Thread
10 Thread
12 Thread
Speed Up
File sizes (Mb)


Parallel computing Trigrams over different file sizes
Sequential
2 Thread
4 Thread
6 Thread
8 Thread
10 Thread
12 Thread
Execution time (ms/op)
File sizes (Mb)


Speed Up Trigrams over different file sizes
2 Thread
4 Thread
6 Thread
8 Thread
10 Thread
12 Thread
Speed Up
File sizes (Mb)