Bigrams and Trigrams

Elena Sesoldi
elena.sesoldi@stud.unifi.it

Giulia Pellegrini
giulia.pellegrini1@stud.unifi.it

Abstract

N-gram is a contiguous sequence of N items from a sequence of text. N-grams of texts are extensively used in text mining and natural language processing tasks.
In this paper we present some different implementations of an algorithm that compute the number of N-gram occurrences: the first is the sequential version and the other represent parallel version in order to appreciate the efficiency of parallelization; those all were implemented on Java.
For testing all versions we use the same dataset of books downloaded from the Gutenberg project looking over the different time of execution, in particular we compare the speed up.

1. Introduction

N-grams are used for a variety of different task. For example, when developing a language model, N-grams are used to develop not just unigram models but also bigram and trigram models. *Google* and *Microsoft* have developed web scale N-gram models that can be used in a variety of tasks such as spelling correction, word breaking and text summarization. Another use of N-grams is for developing features for supervised Machine Learning models such as SVMs, MaxEnt models, Naive Bayes, etc. The idea is to use tokens such as bigrams in the feature space instead of just unigrams.
We decided to consider only Bigrams and Trigrams (two and three length N-gram respectively) of letters and words. In particulary word 3-grams can be used for implementation of text prediction in mobile devices. To understanding the benefits of parallelization we implemented a sequential version and a multithread version.

2. Dataset

The *Gutenberg Project*[1](PG) began in 1971 by Michael Hart as a community project to make books available freely to all and it is the oldest digital library. The releases are availlable in plain text but can be included other formats such as HTML, PDF, EPUB, MOBI, Plucker.
Today PG offers over 54,000 free e-books availllable for downloading or online reading.
We decided to download some books in English and other in Italian, in order to identify the most frequent bigrams and trigrams of both languages.
We decided to test our programs with different dataset incrementing dimension in order to evaluate the speed up improvement.

3. Implementation

We decided to use *Java* language to implement the project task. We use *Java Threads* for the parallel version implementation.
The program produces two files with *.txt* extension: one reports the results concerning the *"Italian language"* and the other the *"English language"*.
To reach this goal, we create a list of Italian books and another of English books, both containing books taken from the *Gutenberg Project*.
The results report the number of occurrences of bigrams and trigrams of letters and words. Each result is shown in decreasing order so that is easy

[1](https://www.gutenberg.org/)

to see the most frequent bigrams and trigrams for both languages.

3.1. Results structure

Each N-gram and its number of occurrences are considered as a key-value couple and inserted in a map, so that at the end we obtain four different maps for each language considered.

3.2. Sequential Implementation

The program is dived in two parts that compute the same work on the two different datasets.
First of all it reads all the Italian books from a given path in a *BufferReader* and save all the text lines as *String* in a *BlockingQueue<String>*. Then the program creates the four Italian result maps implemented as *ConcurrentHashMap<String,Integer>*.
The use of *BlockingQueue* and *ConcurrentHashMap* is not necessary because in the sequential implementation there is no problem of concurrency, but we want the same data structures for both implementations.
To create these maps the algorithm loops on each book producing partial result maps containg the N-grams found in the analized book and then merges the partial result with the ones already created.
As soon as this part ends the algorithm does the same for the English datasets: creates the book list and for each element produces the partial and the result maps.
Once obtained all the maps the program sorts these in order of the decreasing values (number of N-gram occurencies) using a *TreeMap<String,Integer>*.
At the end the program builds two files named *"Italian Results.txt"* and *"English Results.txt"* containg the corrisponding *TreeMap*.

3.3. Parallel Implementation

We decided to use *Java Threads* with the combination of two patterns for parallel programming: the *Reduction Pattern* and the *Producer and Consumer Pattern*.

Reduction Pattern

Reduction combines all the elements in a collection into one using an associative two-input, one-output operator. Given n elements in the collection, using the operator on two adjacent elements can be chosen and combined into one to give n-1 elements. This process can be repeated until there is only one element. If the operator is addition, then a reduction computes the sum of all the elements in a collection. If it is maximum, then the reduction computes the largest value in the collection.

Producer-Consumer Pattern

Producer-Consumer is a pattern based on two processes, the producer and the consumer, who share a common fixed size buffer used a queue. The producer's task is to put data into the buffer if it is not full, otherwise it waits until another process removes data from the buffer. The consumer's task is to extract data from the buffer if it is not empty, otherwise it waits someone that inserts element in it.

We use two different classes that implements *Runnable* interface of *Java* , one named *Producer* and the other *Consumer*.
The *Producer* loads each book's line from the directory in a *BlockingQueue*, we use two producers in order to keep books of the two languages in different buffers.
The *Consumer* takes the lines from the same *BlockingQueue* one by one and compute the partial results of the current line, which are four *HashMap* for the N-grams of every type. The program expects more than one consumer for each dataset to improve the performace. The *Consumer* threads share four *ConcurrenHashMap* and as soon as the *BlockingQueue* is empty and the *Producer* has completed its task, the partial results are completed and the consumers can add them in the corrisponding shared *ConcurrentHashMap*. The main program creates first the Italian Producer and some consumers

for the Italian datasets then it waits them with the *join* method. After that the main program does the same for the English version. *"Italian Results.txt"* and *"English Results.txt"* are created as in the Sequential implementation.

ConcurrentHashMap

ConcurrenHashMap is a map with atomic operation.
This class implements versions of methods corresponding to each method of *HashMap* and supports full concurrency. It is thread safe without synchronizing the whole map in fact it doesnt throw a *ConcurrentModificationException* if one thread tries to modify it while another is iterating over it.
ConcurrenHashMap uses multitude of locks, in fact it synchronizes only the portions of the map that are modified and it is lock-free in lecture.

In this program the use of *ConcurrentHashMap* is necessary to guarantee the concurrent access at the maps containing total results from Consumer threads without make use of explicit lock.

BlockingQueue

BlockingQueue is a queue that additionally supports operations that wait for the queue to become non-empty when retrieving an element,and wait for space to become available in the queue when storing an element.
Thanks of the use of *BlockingQueue* it was not necessary to synchronize the thread insertion and removal operations of the elements of the buffer.

4. Experiments

We tested our programs with datasets of different sizes, starting with one of 2.7 MegaByte and doubling up every time to reach a dimension of 86.4 MegaByte. This values represent the sum of the Italian and English datasets.
We tried more parallel versions with different number of consumers to see how execution times change increasing comsumer threads.

In Table 1 we show the execution times of sequential version and parallel ones. In the first column there are the sizes of the datasets in MegaByte. The other columns report the execution times in milliseconds of sequential, in the second one, and parallel implementation, in the last ones.
For the parallel versions the number of producer is always one for English datasets and one for the Italian ones. The number of consumers instead varies from one to four for each dataset. For example, in the third column, named *Par1*, we have a producer and a consumer for the Italian datasets and a producer and a consumer for the English ones.

Table 1. Table that reports the execution times

Size	Seq	Par1	Par2	Par3	Par4
2.7	2902	3168	2661	2483	2573
5.4	5564	5518	4249	4203	4185
10.8	10563	10430	7479	7334	6177
21.6	22269	21109	13539	11326	11121
43.2	54458	41346	27533	22001	21988
86.4	156008	83835	55353	43905	42066

Once the times have been detected we compute the relative speedups with the formula:

$$Speedup = \frac{Time_{seq}}{Time_{par}}$$

The speedup is a number that measures the relative performance of two systems processing the same problem. In our case the two systems are the sequential and the parallel implementation.

Table 2. Table that reports the speedups

Size	Sp1	Sp2	Sp3	Sp4
2.7	0.9160	1.0906	1.1687	1.1279
5.4	1.0083	1.3095	1.3238	1.3295
10.8	1.0128	1.4124	1.4403	1.7101
21.6	1.0055	1.6448	1.9662	2.0024
43.2	1.3171	1.9779	2.4753	2.4767
86.4	1.8609	2.8184	3.5533	3.7086

In Table 2 we show the speedups of the different parallel versions. In the first column there are the sizes of the datasets in MegaByte. The other columns report the speedups.
For example, in the second column, named *Sp1*, we have the ratio between the sequential time and the parallel time with a producer and a consumer for the Italian dataset and a producer and a consumer for the English one.

The two tables point out that the value of the speedup grows with dataset size and number of consumer increasing. In fact the best speedup in our experiments is obtained with the biggest tested dataset (of 86.4 MegaByte) and the maximum number of consumers (four for Italian dataset and four for the English one, for a total of eight) with a value of 3.7086.

The only case of loss of performance found is the test with the smallest dataset (2.7 MegaByte) and a consumer for each language. This could depend of the cost for thread creation.
These experiments have been done using one *Intel Core i5-7200U Processor* that has 2 fisical cores and 2 logic ones, with a clock frequency of 2.500GHz; regarding the specific software we work with *Ubuntu* 17.04 at 64 bit.

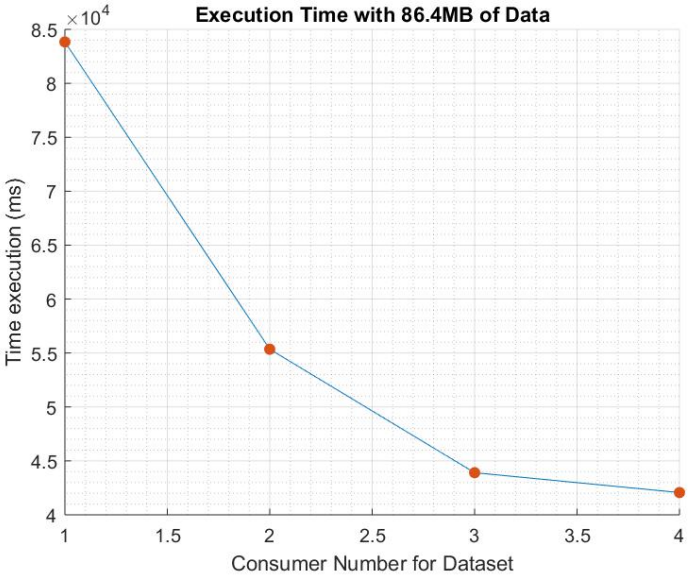

Figure 1. Execution Time with 86.4MB of Data

5. Results

In this section we present in two tables the most common bigrams and trigrams found by our algorithm. In the first table(Table 3) there are the results for the English language and in the second (Table 4) for the Italian. The occurence numbers refer to the biggest tested dataset.

Table 3. Table that reports English Result

Letter2grams	Letter3grams	Word2grams	Word3grams
th	the	of-the	one-of-the
2062144	1284320	99776	3392
he	and	in-the	i-do-not
1866208	566816	71328	3392
in	ing	to-the	out-of-the
1300448	510048	41952	3232

Table 4. Table that reports English Result

Letter2grams	Letter3grams	Word2grams	Word3grams
on	ent	e-di	per-lo-piu
1063040	435072	15328	1472
er	ell	per-la	poco-a-poco
963808	415040	12000	1472
en	ion	che-si	a-poco-a
914400	332768	11840	1472

6. Conclusion

In this work we propose two different implemetations of N-grams occurence calculus: bigrams and trigrams of letters and words. Both versions are implemented in *Java*.
The first implementation is the sequential one: it takes one line of a book at a time and processes it. The second one is the parallel version with explicit *Java Threads* that using *Reduction Pattern* and the *Producer and Consumer Pattern* that computes the same work but the execution time is quicker. In this way we show the efficiency of parallelization; with the increasing of the dimension of the dataset the execution time of the parallel version is three times less of sequential one, indeed the speedup has a value of 3.7086.