Bigrams And Trigrams, Sequential and parallel implementations

Andrea Simioni
andreasimioni5@gmail.com

Abstract

This project is about the implementation of an algorithm for compute and estimate the occurrence of bigrams and trigrams in two versions: 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. Through computational time and speed up, by varying the number of threads in the parallel version, it was possibile to evaluate the performances of each versions. All the tests were performed on a machine having intel i7 quad-core.

Future Distribution Permission

The author(s) of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses.

1. Introduction

In the fields of computational linguistics and probability, an n-gram is a contiguous sequence of n items from a given sample of text or speech. The items can be phonemes, syllables, letters, words or base pairs according to the application. The n-grams typically are collected from a text or speech corpus. Using Latin numerical prefixes, an n-gram of size 1 is referred to as a unigram; size 2 is a bigram (or, less commonly, a digram); size 3 is a trigram.

n	Letters	Names
1	p	unigram
2	pa	bigram
3	par	trigram

Table 1: Examples of n-grams.

The main goal is to compute and estimate occurrences of Bigrams and Trigrams in a certain text. It was used a wikipedia extract.

Bigram	Occurrence
aa	98
ab	56
ac	277

Table 2: Occurrences of bigrams.

Trigram	Occurrence
cce	18
cci	13
cch	12

Table 3: Occurrences of trigrams.

1.1. Computation

depending on the version to implement, the computation of bigrams and trigrams was formulated in two ways.
The sequential version has 2 types of data:

- *n*: number of grams, 2 for bigram and 3 for trigram
- *filestring*: contains the character from the txt file

The parallel version has the same structure but uses more information, depending on how many threads will be used:

- *id*: contains the thread ID
- *k*: dimension of txt for each thread, calculated as *floor(text.length/nThread)*
- *begin*: start character, calculated as (k * i)

Algorithm 1: Sequential version

```
Data: n, filestring
for i = 0 to filestring.length-n+1 do
    key = "";
    for j = 0 to n - 1 do
        key = key + filestring[i+j];
    end
    // insert key in a data structure;
end
```

- *stop*: end character, calculated as (i + 1) * k + ((n - 1) -1)

Algorithm 2: Parallel version

```
Data: id, k, n, begin, stop, filestring
for i = this.begin to (this.stop - this.n+1) do
    key = "";
    for j = 0 to this.n - 1 do
        key = key + filestring[i+j];
    end
    // insert key in a data structure;
end
```

2. Implementation

As told before the implementation was developed using Java language. For the parallel version it was used respectively Java Thread.

2.1. Sequential

2.1.1 Text Preprocessing

Firstly, to compute the occurrences of trigrams and bigrams, text must be cleaned from characters defined invalid. The function implemented with this behavior is called *readAndCleanText()* from *Utils* class. In the algorithm it was used 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.

Algorithm 3: readAndCleanText()

```
Data: yourPathString
path = Paths.get(yourPathString);
lines = Files.lines(path);
str = lines.collect(Collectors.joining());
file = str.replaceAll("charToRemove").toCharArray();
for i = 0 to file.length-1 do
    if isUpperCase(filestring[i]) then
        file[i] = toLowerCase(file[i]);
    end
end
return file;
```

2.1.2 Data structure

HashMaps are the selected data structure to store bigrams and trigrams. Reason of this choice is that HashMaps provide the 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 collection since Java 1.2 and it provides the basic implementation of the Map interface of Java. Data are stored in (Key, Value) pairs and for having access to a value one must know its key. Hashmaps make no guarantees as to the order of the map, but for the purpose of the project order is not required.

2.1.3 Computation

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

- *int n*: 2 or 3, it identifies bigram or trigram.
- *char[] filestring*: the cleaned text to analyse.

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

Algorithm 4: computeNGrams()

```
Data: n,filestring
hashMap = new hashMap();
for i = 0 to filestring.length-n+1 do
    builder = new StringBuilder();
    for j = 0 to n do
        builder.append(filestring[i + j]);
    end
    key = builder.toString();
    if !hashMap.containsKey(key) then
        hashMap.put(builder.toString() , 1);
    end
    else
        hashMap.put(builder.toString() ,
          hashMap.get(key) + 1);
    end
end
return hashMap;
```

2.2. Parallel

Dividing the text in input as many parts as are the thread istances and leave the search of bigrams and trigrams on a single part to a single thread provides a parallelization of the main task. Thanks to this approach the computational times are significantly reduced compared to sequential times.

2.2.1 Java Thread

For the parallelization in Java, it was used the Java threadering model, in particular java.util.concurrent package that provides tools to create concurrent applications. For the purpose of the project the features used 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*: a complete solution for asynchronous processing. It manages an in-memory queue and schedules submitted tasks based on thread availability. To use ExecutorService, you need to create one Runnable or Callable class.

2.2.2 Text Preprocessing

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

2.2.3 Data structure

For the purpose it was used ConcurrentHashMap, an hash table supporting full concurrency of retrievals and high expected concurrency for updates. it is part of java.util.concurrent package and 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 is only necessary to lock the bucket that is being altered. It is basically lock-free on reads. Like the normal HashMap, data are stored in (Key, Value) pairs.

2.2.4 Computation

1. The first step is to declare a *ParallelThread* class that implements *Callable* interface. The reason behind this choice is because thread must have a return value, which in this case is the concurrentHashMap generated from the text portion assigned.
2. The call method In *ParallelThread* class allows to compute bigrams or trigrams and create the hashMap.
3. Lastly, the *merge()* function from *HashMerge* class merges the *ConcurrentHashMaps* returned from different threads. This function

adds new bigrams or trigrams, or updates the occurrences. In this first approach, the task of merging was executed in sequential version; subsequently it will be analyzed a further parallel approach where also the merge is implemented in the thread.

Algorithm 5: merge()

```
Data: map, finalMap
for map.first to map.end do
    newvalue = map.getValue ();
    existingvalue = finalMap.getValue (map.getKey());
    if existingvalue != NULL then
        newvalue = newvalue + existingvalue;
    end
    finalMap.put (map.getKey () , newvalue);
end
return finalMap;
```

4. The final step is to instantiate a *futuresArray*, that it'll contains all the threads, and an *ExecutorService* specifying the thread-pool size. Once the *ExecutorService* is created, it can use it to submit new threads and add them to the *futureArray* list. When the threads have finished their job merge function is called for all map created by the threads, thanks to the Future method *.get()*.

Algorithm 6: JAVA thread execution

```
Data: nThread, fileLength
finalMap = new ConcurrentHashMap ();
futuresArray = new ArrayList<>();
executor = Executors.newFixedThreadPool (nThread);
k = Math.floor (fileLength/nThread);
for i = 0 to nThread do
    Future f = executor.submit (new ParallelThread
      ("constructor values"));
    futuresArray.add(f);
end
for Future f : futuresArray do
    map = f.get ();
    HashMerge (map , finalMap);
end
awaitTerminationAfterShutdown(executor);
```

2.3. different parallel approaches

in order to avoid contention between threads, it was implemented a different approach where the merge function is computed directly in the thread itself, instead of being processed externally. The final concurrentHashmap, is passed as a parameter when the thread is instantiated; Subsequently when this one generates the partial HashMap (a normal Hashmap is used, not concurrent) of the part of the text assigned, instead of returning the value, merges directly the result with the final concurrentHashmap.
A major difference from the previous one is that the *ParallelThread* class implements *Runnable* and no more *Callable*, because the thread hasn't to return any value. A further consequence of this is that the feature *Future* has not been used in this version.

Algorithm 7: Parallel thread: run()

```
Data: n,filestring, finalMap
hashMap = new hashMap();
for i = 0 to filestring.length-n+1 do
    builder = new StringBuilder();
    for j = 0 to n do
        builder.append(filestring[i + j]);
    end
    key = builder.toString();
    if !hashMap.containsKey(key) then
        hashMap.put(builder.toString() , 1);
    end
    else
        hashMap.put(builder.toString() ,
          hashMap.get(key) + 1);
    end
end
for map.first to map.end do
    newvalue = map.getValue ();
    existingvalue = finalMap.getValue (map.getKey());
    if existingvalue != NULL then
        newvalue = newvalue + existingvalue;
    end
    finalMap.put (map.getKey () , newvalue);
end
```

The last approach considered is a kind of hybrid between the two previous versions. The implementation is identical to the first one described, but the thread instead to use a concurrentHashmap as partial map to calculate occurrences of bigrams/trigrams in the part of text assigned, uses a normal HashMap. However, the final result of each merger is saved into a concurrentHashmap. Like the first one, *ParallelThread* class implements *Callable*, and *Future* is used to save the result of each threads.

3. result

To test the sequential version, computational time has been collected varying the text dimension (50KB, 500KB, 10MB, 50MB, 100MB,150MB, 200MB and 300MB). To ensure greater precision, every result is calculated on the average of 10 runs. The same procedure was used for the two parallel versions, but in addition to varying the size of the test or to be analyzed, the number of threads was also changed. 2,4, and 8 threads were used for evaluate performances. Over the maximum number of threads chosen, computational times do not improve more significantly. Lastly, it was also calculated the speedup as follow:

$$S_p = \frac{ts}{tp} \tag{1}$$

to evaluate speed increase using parallel versions with 8 threads and 4 threads, compared to the sequential one. The results are shown below through plots for both bigrams and trigrams. The speedup analysis had a further study by comparing the results of each parallel version.

As shown in the figures, the parallel approaches do not bring a substantial improvement for the short texts. On the other hand, already with medium size texts up to the large ones, the use of the parallel versions resulted in a significant reduction of computational time, and consequently an increase in the speedup value.

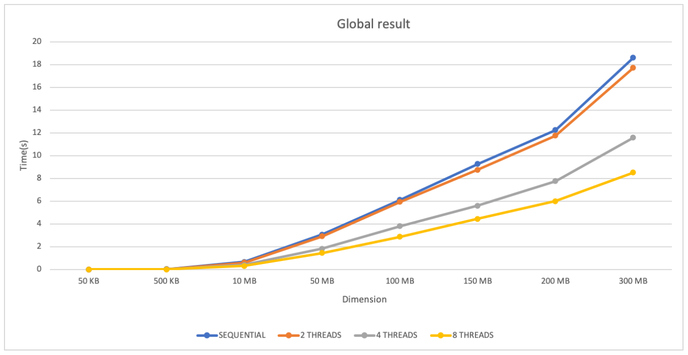

Figure 1: Bigrams result

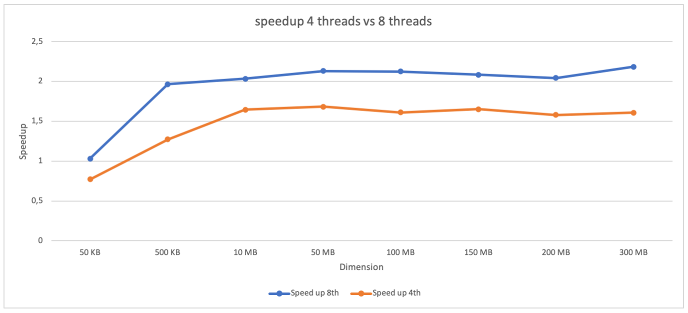

Figure 2: Speedup bigrams: parallel 4/8 threads

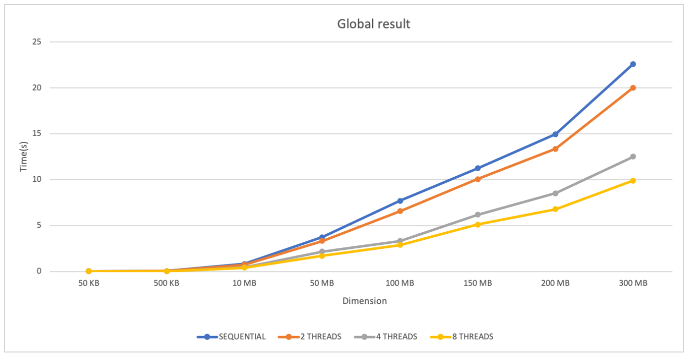

Figure 3: Trigrams result

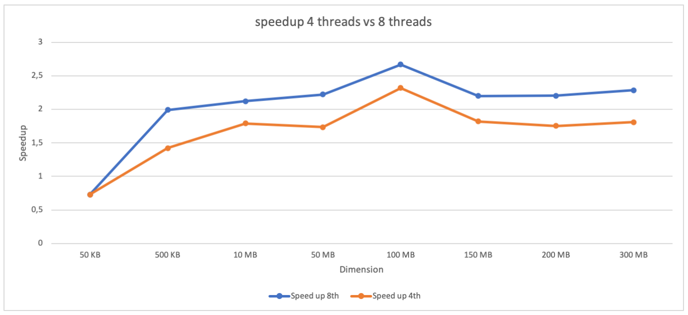

Figure 4: Speedup trigrams: parallel 4/8 threads

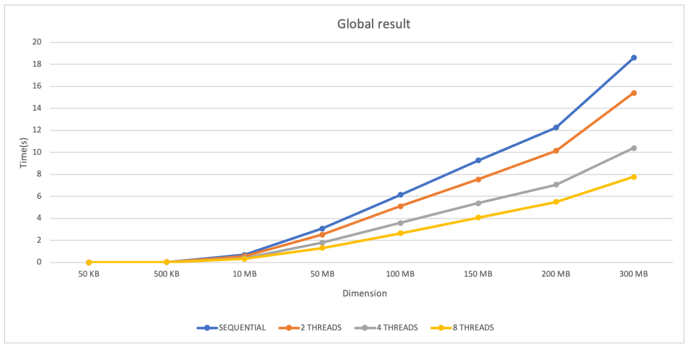

Figure 5: Bigrams result: second parallel approach

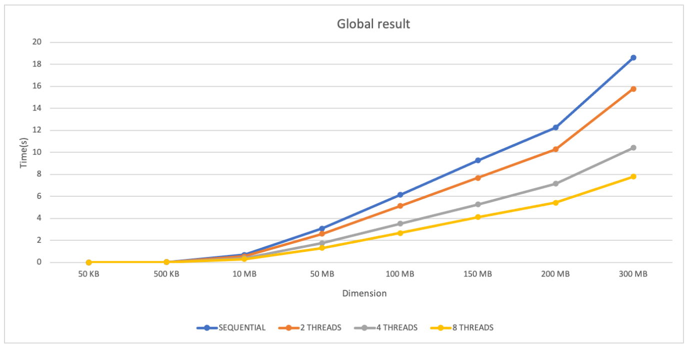

Figure 9: Bigrams result: third parallel approach

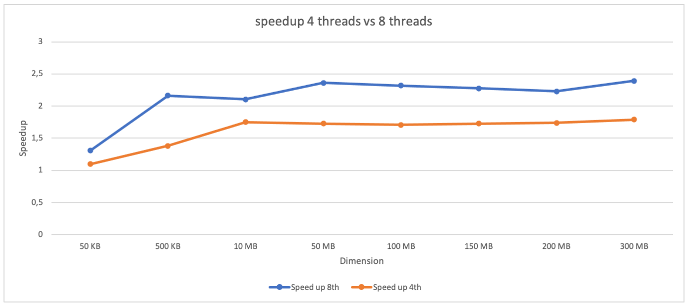

Figure 6: Speedup bigrams: second parallel approach using 4/8 threads

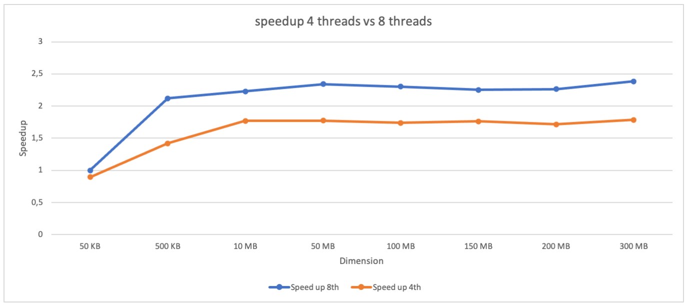

Figure 10: Speedup bigrams: third parallel approach using 4/8 threads

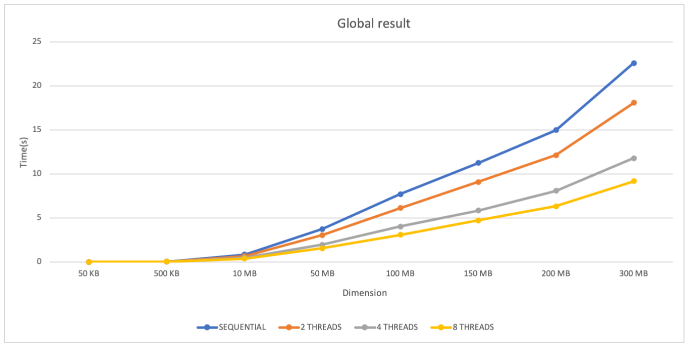

Figure 7: Trigrams result: second parallel approach

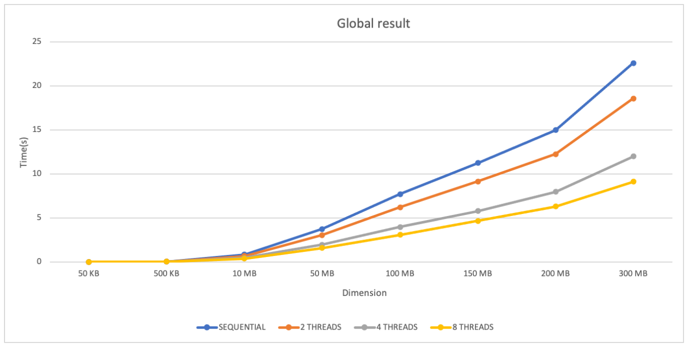

Figure 11: Trigrams result: third parallel approach

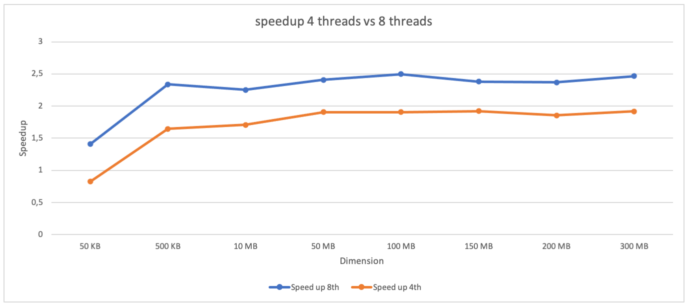

Figure 8: Speedup trigrams: second parallel approach using 4/8 threads

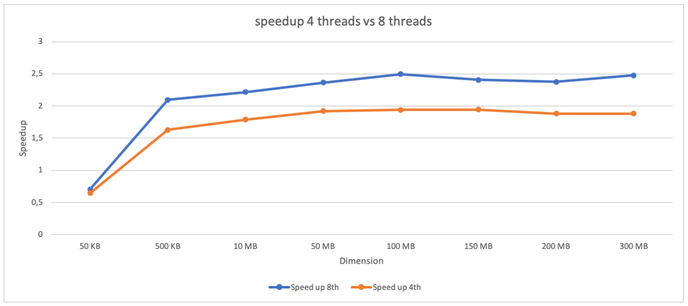

Figure 12: Speedup trigrams: third parallel approach using 4/8 threads

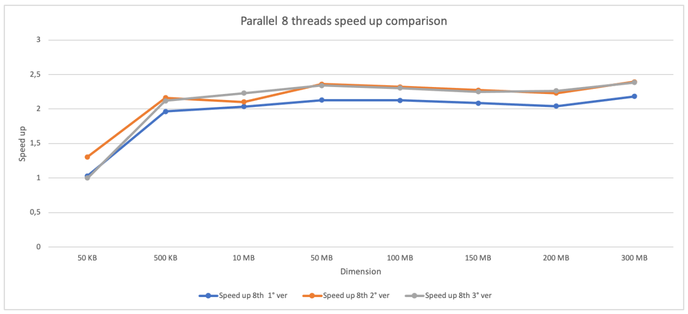

Figure 13: 8 threads speedup bigrams comparison

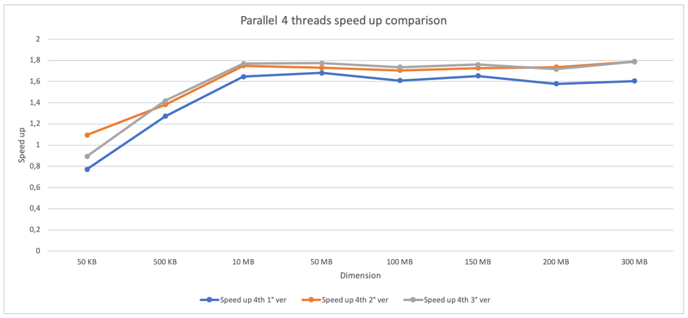

Figure 14: 4 threads speedup bigrams comparison

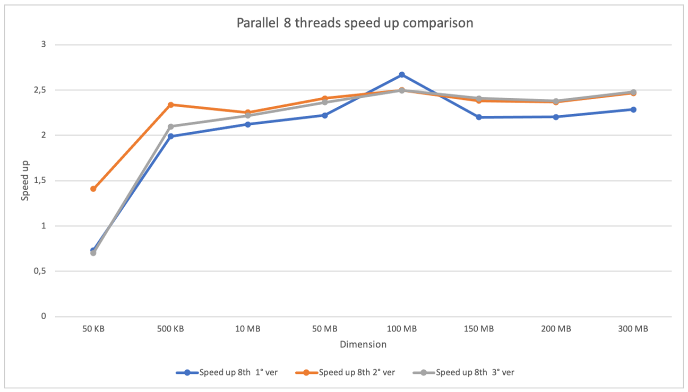

Figure 15: 8 threads speedup trigrams comparison

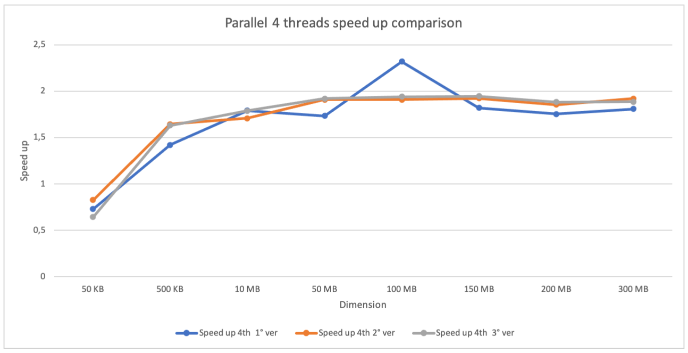

Figure 16: 4 threads speedup trigrams comparison

References

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