Sequential and parallel implementation of bigrams and trigrams, Java and C++

Giulio Calamai
Matricola - 7006568
E-mail Address
giulio.calamai1@stud.unifi.it

Marco Loschiavo
Matricola - 7017247
E-mail address
marco.loschiavo1@stud.unifi.it

Abstract

We present an algorithm for compute and estimate the occurrence of bigrams and trigrams in two different implementations: sequential and parallel. The chosen languages are C++ and JAVA for both implementations and the system makes use of parallelisms in computation via the JAVA thread programming model and the C++ pthreads. 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 i7 quad-core and the results show how the parallel version nearly halves the sequential times.

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" [1].

1.1. Text

The main goal is to compute and estimate occurrences of bigrams and trigrams in a certain text. We used a wikipedia extract [2].

1.2. Computation of n-grams

To compute bigrams and trigrams we have formulated a pseudo-algorithm in two different versions.
The sequential version [1] has 2 types of data:

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

Algorithm 1: Sequential version

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

The parallel version [2] 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)*
- *begin*: start character, calculated as (k * i)
- *stop*: end character, calculated as (i + 1) * k + ((n - 1) -1)

Algorithm 2: Parallel version

```
   Data: id, k, n, begin, stop, filestring
1  for i = this.begin to (this.stop - this.n+1) do
2  |   key = "";
3  |   for j = 0 to this.n - 1 do
4  |   |   key = key + filestring[i+j];
5  |   end
6  end
```

2. Implementation

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

2.1. Sequential

2.1.1 JAVA

Data preprocessing

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 readTextFromFile() [3]. 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.

Algorithm 3: readTextFromFile()

```
   Data: no input data
1  path = Paths.get("yourPath");
2  lines = Files.lines(path);
3  str = lines.collect(Collectors.joining());
4  file =
     str.replaceAll("charToRemove").toCharArray();
5  for i = 0 to file.length-1 do
6  |  if isUpperCase(filestring[i]) then
7  |  |  file[i] = toLowerCase(file[i]);
8  |  end
9  end
10 return file;
```

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).

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

Type Parameters:
K - the type of keys maintained by this map.
V - the type of mapped values.

Hashmaps make no guarantees as to the order of the map. Order for our problem isn't required.

Computation

Bigrams and trigrams are calculated in the body of function *computeNgrams()* [4]. This function gets as input parameters:

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

The function behavior follows the computation for sequential version explained in [1], and returns the HashMap with bigrams or trigrams calculated.

Algorithm 4: computeNGrams()

```
   Data: n,filestring
1  hashMap = new hashMap();
2  for i = 0 to filestring.length-n+1 do
3  |  builder = new StringBuilder();
4  |  for j = 0 to n do
5  |  |  builder.append(filestring[i + j]);
6  |  end
7  |  key = builder.toString();
8  |  if !hashMap.containsKey(key) then
9  |  |  hashMap.put(builder.toString() , 1);
10 |  end
11 |  else
12 |  |  hashMap.put(builder.toString() ,
   |  |    hashMap.get(key) + 1);
13 |  end
14 end
15 return hashMap;
```

2.1.2 C++

Data preprocessing

Like in Java, before computing, is necessary read the text and replace invalid characters.
In C++, however, we need to do some preliminary stuff to avoid read and memory error. First, the function *readTextFromFile()* [6]:

- Open the file and check if it was found.
- Determine the size of the file.
- Allocate space in the memory.

After this operation we can copy the file characters in the allocated memory and proceed with the replacement of the invalid characters, thanks to the function *removeChar()* [5].

Algorithm 5: removeChar()

```
   Data: s, charToRemove
1  *copy = s;
2  *temp = s;
3  while *copy do
4  |   if *copy != charToRemove then
5  |   |   *temp++ = *copy;
6  |   end
7  |   copy++;
8  end
9  *temp = 0;
```

Algorithm 6: readTextFromFile()

```
   Data: no input data
1  pFile = fopen ("text.txt" , "r");
2  if pFile == NULL then
3  |   fputs("File not found");
4  |   exit(1);
5  end
6  fseek(pFile , 0 , SEEKEND);
7  size = ftell(pFile);
8  rewind(pFile);
9  buffer = (char*) malloc (sizeof(char)*size);
10 if buffer == NULL then
11 |   fputs("Memory error");
12 |   exit(2);
13 end
14 result = fread(buffer , 1 , size , pFile);
15 if result != size then
16 |   fputs("Reading error");
17 |   exit(3);
18 end
19 for i = 0 to size do
20 |   c = tolower(buffer[i]);
21 |   buffer[i] = c;
22 end
23 blacklist[] = "invalid char list";
24 for i = 0 to strlen(blacklist) do
25 |   removeChar(buffer, blacklist[i]);
26 end
27 fclose(pFile);
28 return buffer;
```

Data structure

Unordered Maps are the chosen structure to replace the Java HashMaps [5]. They are associative containers that store elements formed by the combination of a key value and a mapped value, which allows for fast retrieval of individual elements based on their keys. Internally, the elements in the unordered maps are not sorted in any particular order with respect to either their key or mapped values, but organized into buckets depending on their hash values to allow for fast access to individual elements directly by their key values. Unordered map containers are faster than map containers to access individual elements by their key.

Computation

Like in Java code, *computeNgrams()* [7] is the implemented function, but we need to use the chosen data structure:

Algorithm 7: computeNGrams()

```
   Data: n, filestring
1  fileStr = filestring;
2  fileStr.erase(remove(fileStr.begin(),fileStr.end,"
     "),fileStr.end());
3  for i = 0 to fileStr.length() + n - 1 do
4  |   key = "";
5  |   for j = 0 to n - 1 do
6  |   |   key = key + fileStr[i + j];
7  |   end
8  |   if map.find(key) == map.end() then
9  |   |   map.insert(key,1);
10 |   end
11 |   else
12 |   |   map[key] += 1;
13 |   end
14 end
15 return map;
```

2.2. Parallel

The idea to parallelize the sequantial 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.

2.2.1 JAVA Thread

To parallelize in Java language, we used the Java's threadering model, in particular *java.util.concurrent* package that provides tools to create concurrent applications. 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 availability. To use ExecutorService, we need to create one Runnable or Callable class.

Data preprocessing

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

Type Parameters:
K - the type of keys maintained by this map.
V - the type of mapped values.

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 (ConcurrentHashMap in our case) and because they are called from an *ExecutorService* object that requires a *Callable* class.

2. In *ParallelThread* class, we have implemented the call method described in [2] using the algorithm [4] that allows to compute bigrams or trigrams and create the hashMap.

3. Implement an *HashMerge* function [8] that merges the *ConcurrentHashMaps* returned from different threads. This function adds new bigrams or trigrams, or updates the occurrences. We choose to do this task in sequential version because after doing some measurements we have saw that the computational time of HashMerge in both parallel and sequential is irrelevant to the complessive time of execution.

Algorithm 8: HashMerge ()

```
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, we can use it to submit new threads and add them to the *futureArray* list. When the threads have finished their job we call the HashMerge function for all map created by the threads, thanks to the Future method *.get()* [9].

Algorithm 9: Thread generator and execution in JAVA

```
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"));
    existingvalue = finalMap.getValue
      (map.getKey());
    futuresArray.add(f);
end
for Future f : futuresArray do
    map = f.get ();
    HashMerge (map , finalMap);
end
awaitTerminationAfterShutdown(executor);
```

2.2.2 C++ Thread - PThread

Pthreads API are used to parallelize the C++ version. We have created a .h file which implements the Thread class including <pthread.h>.

Data preprocessing

To read and replace invalid characters is used the same function *readTextFromFile()* of sequential version [6].

Data structure

We used Unordered Maps as in sequential version.

Computation

1. First of all we have implemented the *Thread.h* class including <pthread.h>. Then we have defined all the thread methods like *self*, *join*, *start*, *runThread* and the destructor in the *Thread.cpp* file. Now we have the definition of Thread class.

2. We create a *parellelThread.h* file that include *Thread.h*, defining the thread methods like *run* and *getMap ()*.

3. In the *parallelThread.cpp* file, we have implemented the method *int *run()*, that computes bigrams or trigrams as described in [7], and *getMap()*, to return the UnorderedMap.

4. Define an *HashMerge* function to merge the returned maps from threads (same as the JAVA version 8).

5. In the final step, we have declared an array of threads and a vector of UnorderedMaps. The threads are instantiated into the array and then started. After the *join()* the returned maps are pushed back into the vector of UnorderedMaps and then merged [10].

Algorithm 10: Thread generator and execution in C++

```
Data: nThread, fileLength
ParallelThread *thread [nThread];
vector <unorderedMap<string,int>> maps;
unorderedMap <string,int> finalMap;
k = Math.floor (fileLength/nThread);
for i = 0 to nThread do
    thread [i] = new ParallelThread("constructor
      value");
    thread [i] -> start ();
    existingvalue = finalMap.getValue
      (map.getKey());
    futuresArray.add(f);
end
for i = 0 to nThread do
    thread [i] -> join ();
    maps.pushBack (thread [i] -> getMap ());
end
finalMap = maps [0];
for i = 1 to maps.size () do
    finalMap = hashMerge (maps [i] , finalMap)
end
```

3. Test

To test the sequential versions (Java and C++), the computational time has been collected 100 times vary with the text of 50KB, 500KB, 10MB, 50MB, 100MB and 150MB and averaged. For parallel versions the number of tests is depending on the number of threads used.

- 2,4 and 8 threads for the JAVA parallel version.

- 2,4 and 8 threads for the C++ parallel version.

For every thread number the computational time has been collected 100 times and averaged. Over the maximum number of threads chosen, computational times don't improve more significantly.

4. Results

The different implementations are valutated through the computational time and speed up. Below we show the results through plots for bigrams and trigrams either in JAVA and C++ implementations. We have used a semi-logarithmic scale.

4.1. JAVA Results

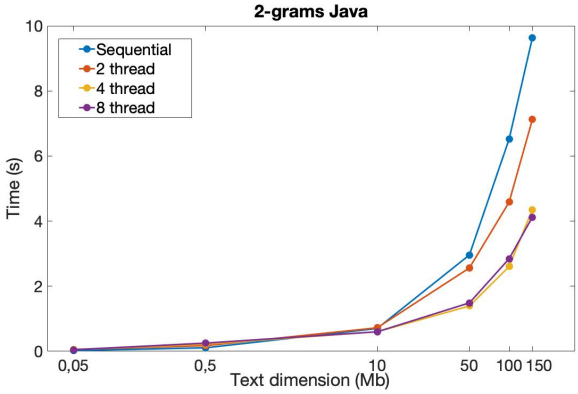

Figure 1. 2-grams result, JAVA

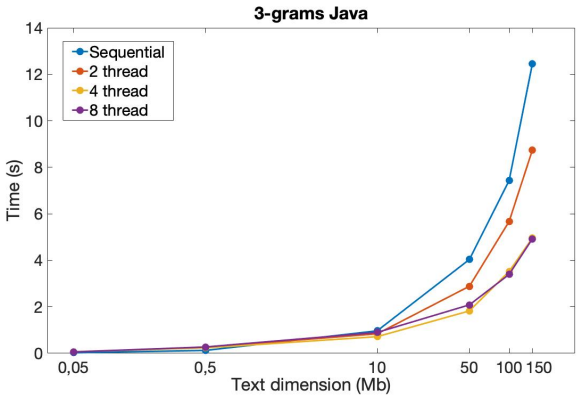

Figure 2. 3-grams result, JAVA

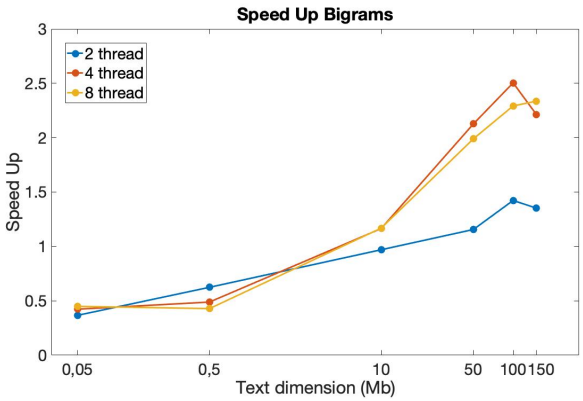

Figure 3. 2-grams speed up result, JAVA

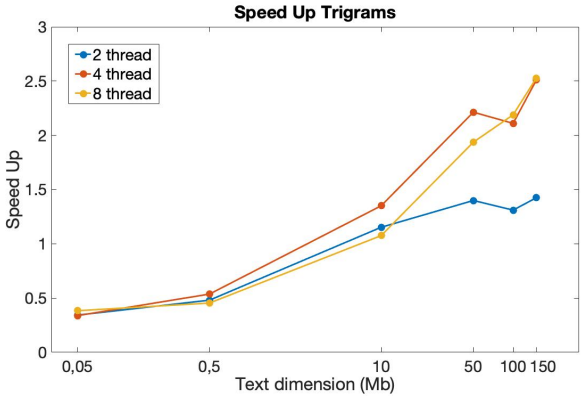

Figure 4. 3-grams speed up result, JAVA

4.2. C++ Results

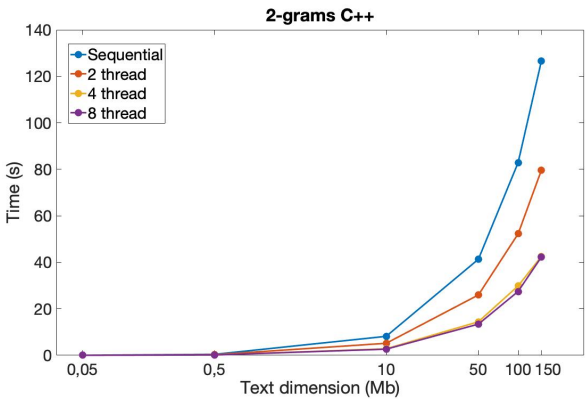

Figure 5. 2-grams result, C++

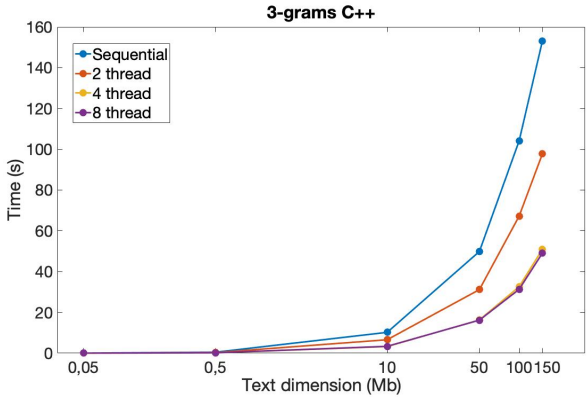

Figure 6. 3-grams result, C++

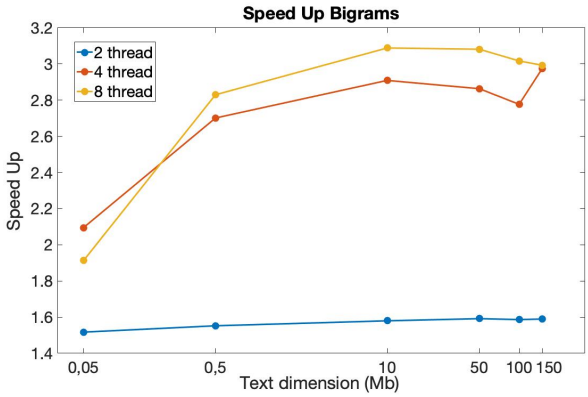

Figure 7. 2-grams speed up result, C++

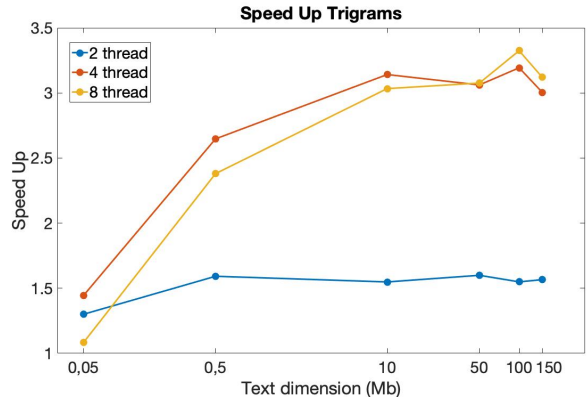

Figure 8. 3-grams speed up result, C++

5. Overview: JAVA Hashmaps VS C++ UnorderedMaps

To store the bigrams and trigrams we have choose two data structure: JAVA HashMaps and C++ UnorderedMaps. Their goal is to store elements in key-value pair and provide member functions to efficiently insert, search and delete key-value pairs.
The C++ execution time, as shown in results (4), is greater than JAVA.
So, during our tests, we have measured the time of three part of the code:

- *readTextFromFile ()*: read and write the txt file in memory.
- *computeNGrams ()*: generate the map of pair elements.
- *HashMerge ()*: merge all the maps obtained from the threads (parallel version).

The measurements we did helped us to understand that the function *computeNGrams ()* takes up most of the execution time, so the main difference in time between JAVA and C++ are the maps used. UnorderedMaps are three times slower than HashMaps in searching, reason for which this kind of implementation is much better with JAVA HashMaps [3].

References

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

[2] Wikipedia - MacOs - https://it.wikipedia.org/wiki/MacOS

[3] Chandler Carruth - CppCon Talk 2014 - https://www.youtube.com/watch?v=fHNmRkzxHWs