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

Alessandro Minervini
Matricola - 6414770
alessandro.minervini@stud.unifi.it

Francesco Lombardi
Matricola - 6344416
francesco.lombardi3@stud.unifi.it

Abstract

Compute and estimate occurrences of bigrams and trigrams in two different implementations: sequential and parallel. We study the different implementations through computational time and speed up, vary depending on number of threads in the parallel version. The chosen languages are C++ and Java for the sequential version and the thread introduction in both languages for the parallel version. The tests were performed on intel i5 dual core and the results show how the parallel version nearly halves the sequential times.

1. Bigrams and trigrams

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

- A sequence of two letters (e.g. of) is called a bigram
- A three-letter sequence (e.g. off) is called a trigram
- The general term n-gram means 'sequence of length n'

n	letters	name
1	o	unigram
2	of	bigram
3	off	trigram

Table 1. Examples of bigrams and trigrams.

1.1. Text

Bigrams and trigrams are searched on a wikipedia extract. Link to the wikipedia page: `https://it.wikipedia.org/wiki/Dungeons_%26_Dragons`

1.2. Computation

To compute bigrams and trigrams we have formulated two ways, in particular depending on the version to implement.

Sequential version:
Data: n = 2(bigrams) or 3(trigrams)
for $i = n - 1$ *to* $filestring.length$ **do**
  key = " ";
  for $j = n - 1$ *downto 0* **do**
    key = key + filestring[i-j];
  end
end

Algorithm 1: Sequential computation

Define *nThread* threads with these attributes: *id, n, begin, stop, fileString*.
Define *k = floor(text.length/nThread)* as number of text splits depending from number of threads.

Parallel version:
Data: i =idthreads, n =2(bigrams) or 3(trigrams),
  $begin$=(k * i), $stop$ = (i + 1) * k + ((n - 1) -1),
  $fileString$=txt
for $i = i = this.begin + this.n - 1$ *to* $this.stop$ **do**
  key = " ";
  for $j = this.n - 1$ *downto 0* **do**
    key = key + this.fileString[i-j];
  end
end

Algorithm 2: Parallel computation

2. Implementation

Next up are shown the sequential and parallel implementation: for the sequential, Java and C++ are the chosen languages. For the parallel 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()*.

```
public static char[] readTextFromFile() {
  Path path=Paths.get("yourPath");

  try {
    Stream<String> lines=Files.lines(path);
    char[] filestring=(lines.collect
       (Collectors.joining()))
       .replaceAll("[ '();:,.]", "")
       .toCharArray();

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

  }

  catch (IOException e) {
    System.out.println(e);
    System.exit(1);
    return null;
  }
}
```

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()*. This function gets as input parameters:

- *int n* that identifies if to look for bigrams or trigrams.
- *char[] fileString* as text to analyse.

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

```
public static HashMap<String, Integer> computeNGrams
                        (int n, char[] fileString) {

  HashMap<String, Integer> hashMap = new HashMap();

  for(int i=0; i<fileString.length-n+1; ++i){
       StringBuilder builder=new StringBuilder();
            for(int j=0; j<n; ++j) {
                 builder.append(fileString[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;
}
```

2.1.2 C++

Data preprocessing
Like in Java, before computing, is necessary read the text and replace invalid characters. The function implemented for these operations is *readTextFromFile()* witch recalls inside the body the fuction *removeChar()*.

```
char* readTextFromFile() {

   FILE * pFile;
   long size;
   char * buffer;
   size_t result;

   pFile = fopen ( "text.txt" , "r" );
   if (pFile==NULL) {
      fputs ("File not found",stderr); exit (1);
   }

   //file size:
   fseek (pFile , 0 , SEEK_END);
   size = ftell (pFile);
   rewind (pFile);

   //allocate memory:
   buffer = (char*) malloc (sizeof(char) * size);
   if (buffer == NULL) {
      fputs ("Memory error",stderr); exit (2);
   }

   //copy the file into the buffer:
   result = fread (buffer,1,size,pFile);
   if (result != size) {
      fputs ("Reading error",stderr); exit (3);
   }
```

```
    for (int i = 0; i < size; ++i){
        char c = tolower(buffer[i]);
        buffer[i] = c;
    }

    char blackList[] = 'here invalid characters';

    for (int i = 0; i < strlen(blackList); i++){
        removeChar(buffer, blackList[i]);
    }

    fclose (pFile);
    return buffer;
}

void removeChar(char* s, char charToRemove){
    char* copy = s;
    char* temp = s;

    while (*copy){
        if (*copy != charToRemove)
            *temp++ = *copy;
        copy++;
        }

    *temp = 0;
}
```

Data structure
Unordered Maps are the chosen structure to replace the Java HashMaps. 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()* is the implemented function, but in C++ language:

```
unordered_map<string, int> computeNgrams
                        (int n, char* fileString){

    unordered_map<string, int> map;

    string fileStr = fileString;

    fileStr.erase(remove(fileStr.begin(),
              fileStr.end(), '\n'), fileStr.end());


    for (int i = n-1; i < fileStr.length(); i++){
        string key = "";

        for(int j=n-1; j>=0; j--){
            key = key + fileStr[i-j];
        }

        if(map.count(key) == 0){

            pair<string,int> pair (key,1);
            map.insert(pair);
        }
        else{
            if(map.count(key) >= 1){
                map[key] += 1;
            }
        }
    }
    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 threading model, i n 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.

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

- First step is declare a thread 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 1.1 (parallel version) that allows to compute bigrams or trigrams.
- 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()*.

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.

Data structure
Are used Unordered Maps as in sequential version.

Computation

- First of all we have implemented the Thread class in a file .h including <*pthread.h*>. Now we have the definition of Thread class.
- We extend the Thread class(including <*Thread.h*>) defining parallelThread.
- Implement a parallelThread method, *int *run()*, that computes bigrams or trigrams as described in 1.1(parallel version). To return the UnorderedMap we have implemented a getter method *getMap()*.
- Define a HashMerge function to merge the returned maps from threads.
- Finally, 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.

3. Test

To test the sequential versions (Java and C++), the computational time has been collected 100 times vary with the text (from 50Kb to 32Mb) and averaged. For parallel versions the number of tests is depending on the number of threads used.

- 2 and 4 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.

Java results

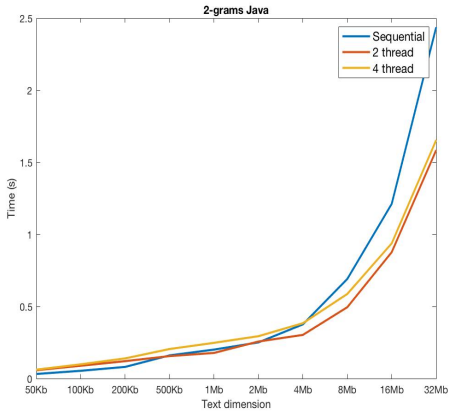

Figure 1. Bigrams results , Java

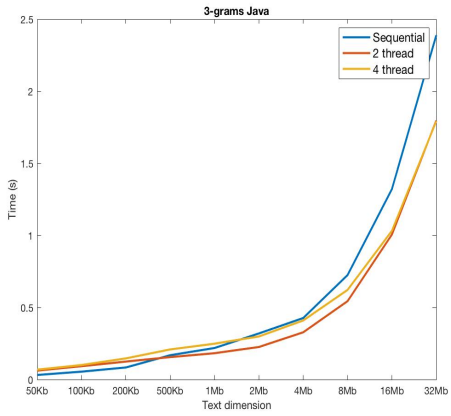

Figure 2. Trigrams results, Java

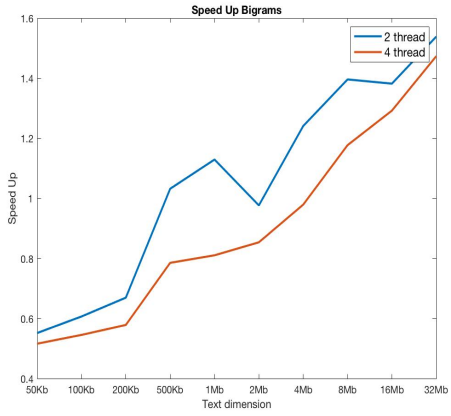

Figure 3. Bigrams speed up, Java

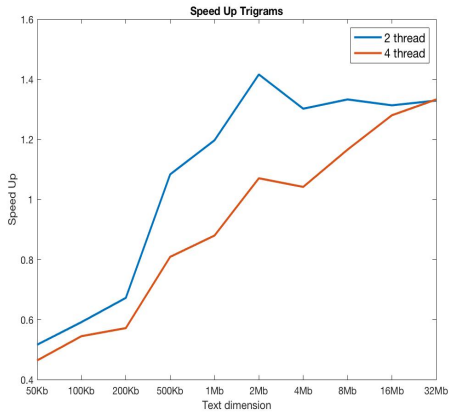

Figure 4. Trigrams speed up, Java

C++ results

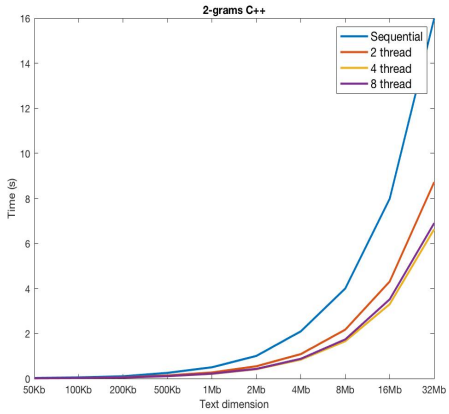

Figure 5. Bigrams results, C++

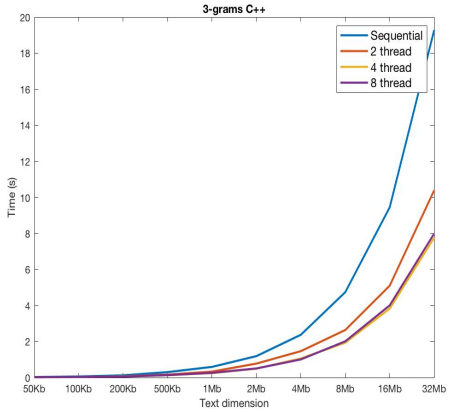

Figure 6. Trigrams results, C++

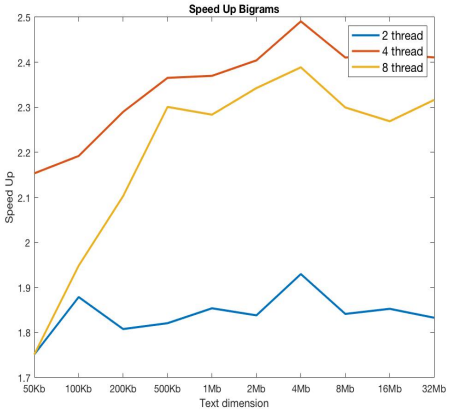

Figure 7. Bigrams speed up, C++

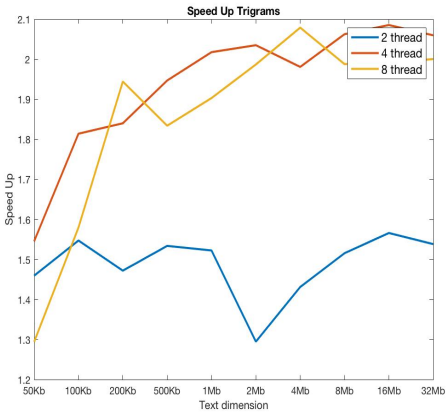

Figure 8. Trigrams speed up, C++