# Three algorithms for letter n-grams extraction

Stefano Vannucchi
E-mail address
stefano.vannucchi@stud.unifi.it

## Abstract

*With the introduction of modern multithreads CPUs, has proved necessary the definition of new programming approach capable to exploit this new processing power.*

*Therefore, many programming languages have been introduced (natively or by integration of external modules) new data structures which allow to manage the concurrent execution of multiple threads.*

*In this paper we will compare the implementations of three different algorithms able to solve the problem of find letter n-grams in text corpus, mined from the Gutenber project.*

*An algorithm will be presented in its sequential form, while the others will be implemented exploiting the parallelism of C++ and Java languages.*



### 1. Introduction

N-grams are contiguous sequences of n items from a given sample of text or speech. In our context we treat sequences of letters, but they could be also sequences of words. In the last one case the calculated n-grams may also be called *shingles*.

N-grams are widely used in natural language processing, for instance in information retrieval, language identification, text classification, and so on. One of the basically usage in these context is to calculate n-grams of words in a corpus and then compose a *bag of word* model.

The compute of an n-gram (of letters) is quite expensive, because it need to iterate every single letter of a sentence and extract a sequence of n characters a time to compose the global index. (We can imagine the algorithm as a sliding window of length n flowing over the sentence, letter after letter).

As we have said previously for this experiment has been used a massive extraction of text books from the Gutenberg project, a library of over 60,000 free eBooks, for enjoyment and education.

The extraction has been performed thanks to an automatic web crawling using *wget* software and Gutenberg's REST api.

In the next section we will explain sequential implementation first and then we will continue with the parallel versions. This paper will end with a comparison of the gained performance of these algorithms.

### 2. Sequential implementation using C++

The first version was made in the serial form, exploiting the C++ programming language.

The main part of the algorithm, which is at the base of the entire resolving process, consist in reading all the files in a directory (into a *std::ifstream* object) and calling the *getNgrams(...)* method on this object.

This method scans every single text row and process it separately, searching the desired letter n-grams inside.

This n-grams will be save in an *hash table*, through key-value pairs; where the key is the sin-

gle n-gram and the value will be the number of times that the n-gram appear in the text.

The algorithm is quite simple, even if inside there are several nested loops that make its computational complexity grow up. For this reason was developed a parallel version, able to solve the same problem.

Below is reported a code snippet that shows how the *getNgrams* method extracts the single text elements.

```
// Code corpus of getNgrams method
std::string s;
while (getline(infile, s)) {
  std::transform(s.begin(), s.end(),
      s.begin(), ::tolower);
  s.erase( std::remove(s.begin(), s.end(),
      '\r'), s.end() );
  s.erase( std::remove(s.begin(), s.end(),
      '\xbf'), s.end() );
  if (s.size() >= n) {
    for(int i = 0; i < s.size() - n + 1;
        ++i) {
      std::string w;
      for(int j = 0; j < n; j++) {
        w += s[i+j];
      }
      if (isNgramGood(w))
        ngrams[w]++;
    }
  }
}
```

We can see that after finding the n-gram, its goodness is evaluated into the *isNgramGood* method. This function checks if the n-gram contains blanc spaces or other characters different from letters or numbers; in these cases the n-gram is discarded, otherwise stored.

## 3. Parallelization of the algorithm

The base idea, used for the parallelization of the algorithm, consist in the implementation of the paradigm *producer/consumer* useful for separation of duties between the various running threads.

Particularly we will have a configuration as follows:

1. one (or more) thread *producer*, with the duty of flowing all the files of a defined directory and, for each file, extract every single row to insert into a data structure (a *queue*) shared among all threads.
2. more than one *consumers* (the number of consumers will change due to program configuration that we want to apply, and it will be strictly dependent to the computer's hardware on which we will execute the program), where each one with the duty of extract an element (row of text) from the *shared queue* and process it. This process consist in finding the n-grams and save them into the *shared hash table*, also mentioned previously during the sequential algorithm explanation.

The threads are independent, but their work is narrowed by the state of the shared queue: if a producer try to insert an element into a full queue, it will be forced to wait; similarly a producer will have to wait in case of extraction of an element from an empty queue.

In this algorithm's version, the threads (producer(s) and consumers) will be started all together. The producer will end its execution when it will have read all the files, and inserted the *poisonPills* (as many as the number of consumers threads) into the shared queue. In the same way each consumer will end its execution after extracting and reading its own *poisonPill* from the shared queue.

### 3.1. Java version

In the Java version is used the *java.util.concurrent* package, that includes several data structures designed for concurrent access. Particularly are used:

1. A *BlockingQueue* (implemented as a concrete *ArrayBlockingQueue* object) for the management of the shared queue. This object define

a structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.

2. A *ConcurrentMap* (implemented as a concrete *ConcurrentHashMap* object) for the management of a shared map used to track the read files (only in case of multiple producer threads).

Furthermore, for the management of the shared map, where are stored the n-grams, is used a custom data structure (called *MyMap*) that encapsulates a *ConcurrentHashMap* object and implements a thread-safe *"addOne()"* method.

In this version the consumers will store their results directly on the map shared among all running threads.

```
while(canWork) {
  if(! lines.isEmpty()) {
    String s = null;
    try {
      s = lines.take();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    if(s.equals(Application.poisonPill)) {
      this.canWork = false;
    } else {
      processString(s);
    }
  }
}
```

As we can see the consumer works until the variable *canWork* is set as true. This variable changes its value only when the *poisonPill* is read.

```
String path = "path-to-files";
File folder = new File(".../" + path);
for (final File fileEntry :
    Objects.requireNonNull(folder.listFiles()))
    {
  try {
    stream = Files.lines(Paths.get(path +
        fileEntry.getName()));
    Object[] line = stream.toArray();
    for (Object token : line) {
      lines.put(token.toString());
    }
  } catch (Exception e) {
      e.printStackTrace(); }
  stream.close();
}
for(int i = 0; i < consumerNo; i++) {
  try {
    lines.put(Application.poisonPill);
  } catch (Exception e ){
      e.printStackTrace(); }
}
```

Here is reported the code of the producer where the files under a specific directory are read, and each lines of these text files are put into the shared queue.

In this application, the threads and the shared objects will be instantiated into the main method. All the threads (producer(s) and consumers) are executed and managed through the use of a *ThreadPool*.

### 3.2. C++ version

In the C++ version are implemented and used two custom data structures:

1. One for the map, called *MyMap*, which integrates (as private attribute) an *std::unordered_map* object and a *mutex*. The first one is used to save the n-grams while the second is used to guarantee mutually exclusive access.

2. One for the queue, called *MyQueue*, which exploit an *std::queue* object protect by the usage of a *mutex* together with a *condition variable*.

In this version the consumers will manage both a private map and the shared one: during their execution they will save only in their private map, while at the end of their work (after read the *poisonPill*) they will save their local results on the shared map.

```
void merge(std::unordered_map<std::string,
    int> mapToMerge) {
  std::unordered_map<std::string,
      int>::iterator it;
```

```
  for ( it = mapToMerge.begin(); it !=
      mapToMerge.end(); it++ ) {
    std::unique_lock<std::mutex> lk(mut);
    auto global_it = map.find(it->first);
    if(global_it != map.end()) {
      map[global_it->first] =
          global_it->second + it->second;
    } else {
      map.insert(std::pair<std::string,
          int>(it->first, it->second));
    }
  lk.unlock();
  }
}
```

Here we can see the *merge()* method of the MyMap class used to finalize the work of a consumer: all the local results are copied and merged with the global results.

```
void wait_and_pop(std::string& value) {
  std::unique_lock<std::mutex> lk(mut);
  data_cond.wait(lk, [this]{return
      !data_queue.empty();});
  value = data_queue.front();
  data_queue.pop();
}
```

This last one code snippet show the *wait_and_pop()* method used by the consumer to take an element from the queue and then process it.

In this application both producer and consumer are *std::thread* objects that are manually started and *joined* after their execution.

## 4. Results and comparison between sequential and parallels versions

Now we show some obtained results comparing the three algorithm versions implemented.

The attempts have been made on a computer with an i5 and 6 cores. The results show a speedup obtained by comparison of sequential and parallels executions of the program. For simplicity are reported only test made searching trigrams, with the following three configuration:

1. 1 producer, 6 consumers
2. 1 producer, 12 consumers
3. 2 producers, 6 consumers

| No of rows | speedup Java | speedup C++ |
|---|---|---|
| 10,000 | 1.16 | 3.15 |
| 100,000 | 2.72 | 3.48 |
| 1,000,000 | 3.18 | 4.43 |
| more than 6,000,000 | 3.25 | 4.60 |

Table 1. tri-grams calculation with 1 producer and 6 consumers

| No of rows | speedup Java | speedup C++ |
|---|---|---|
| 10,000 | 0.82 | 2.09 |
| 100,000 | 2.28 | 2.42 |
| 1,000,000 | 3.48 | 2.86 |
| more than 6,000,000 | 3.28 | 3.01 |

Table 2. tri-grams calculation with 1 producer and 12 consumers

In these tables are reported few of several carried out tests. We can see a growing speedup with the increase of the working data.

Despite the growth, the speedup keeps a sublinear trend.

| No of rows | speedup Java | speedup C++ |
|---|---|---|
| 10,000 | 0.93 | 3.19 |
| 100,000 | 2.37 | 3.35 |
| 1,000,000 | 3.02 | 4.75 |
| more than 6,000,000 | 3.07 | 4.92 |

Table 3. tri-grams calculation with 2 producers and 6 consumers

We can see differences between Java and C++ speedups, probably caused by the different management of the n-grams shared map. In the Java version the threads write only on the shared map unlike the C++ version (where each consumer threads work locally with a private map). This involves an increasing contention and a consequent drop in performance.

## 5. References


1. Marco Berini - Parallel Programming course slides
2. C++ programming language
3. Java programming language
4. Project Gutenberg