



UNIVERSITÀ  
DEGLI STUDI  
FIRENZE

Scuola di Ingegneria

Corso di Laurea in Ingegneria Informatica

Parallel Computing

# Parallel Trigrams

*Edoardo Sarri*

7173337

Febbraio 2026

# Indice

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Features . . . . .	5
1.2	Input and normalization . . . . .	5
1.3	Hash table . . . . .	5
1.3.1	Hash function . . . . .	6
1.3.2	Hash table characteristics . . . . .	6
1.3.3	Choice of hyperparameters . . . . .	6
1.3.4	Overflow . . . . .	6
1.4	Statistics . . . . .	7
1.5	Tools . . . . .	7
1.5.1	Sanitizers . . . . .	7
1.5.2	Profiler . . . . .	8
1.6	Execution . . . . .	8
<b>2</b>	<b>Sequential</b>	<b>9</b>
2.1	Pipeline . . . . .	9
2.2	Hash table . . . . .	10
2.3	Statistics . . . . .	10
2.3.1	Text statistics . . . . .	10
2.3.2	Hash table performance . . . . .	11
<b>3</b>	<b>Parallel</b>	<b>13</b>
<b>4</b>	<b>Analysis</b>	<b>14</b>

# Elenco delle figure

# Listings

# 1 Introduction

The main goal of this project is to count the  $n$ -gram (the occurrences of  $n$  consecutive word) present in an input file. This will be addressed with two approaches: with the sequential implementation in Chapter 2 and with the parallel implementation in Chapter 3. After these two implementations we will evaluate the performance in Chapter 4, with particular focus on the speed-up.

## 1.1 Features

Obviously the main goal is to count the  $n$ -gram. Other features are:

- We can define the  $n$  value (of the  $n$ -gram) at compilation time passing the parameter `N_GRAM_SIZE`. By default is `N_GRAM_SIZE = 3`.
- At the end of computation, the system prints some statistics (see Section 1.4) about the text and about the hash table performance.

## 1.2 Input and normalization

The code works, for simplifying the implementation, only with English sentences. The input file for our analysis was taken from [Wortschatz Leipzig project](#), that contains a big corpus of English text.

We want to normalize the input so that similar words will be represented by the same token, i.e., we want to create equivalence classes that contain similar words. This helps us to reduce the dimension of the word dictionary that composes the trigrams.

We addressed this in these ways:

- Bring all characters in lower case form.
- Remove all the punctuation like periods, commas and exclamation points.
- We maintain only the numbers and the letters.

## 1.3 Hash table

We have to be able to count the trigrams; to address efficiently this goal we will use the hash-table structure. We have to define the hash function to obtain the index in the hash table and the characteristics of the hash table.

### 1.3.1 Hash function

In a hash table we must define the hash function to calculate the table index of the elements. We have to calculate the index of strings, so we adopt the **Polynomial Rolling Hash** technique [1]: given the string  $c = c_1 \cdots c_n$ , its hash value is defined by

$$H(c) = \sum_{i=1}^n (c_i \cdot p^{i-1}) \bmod M.$$

This approach to obtain the hash value is inefficient and dangerous for the overflow. We can exploit the distributive property of the module operation: given  $H(c_1 \cdots c_i)$ , we can obtain  $H(c_1 \cdots c_i c_{i+1}) = (H(c_1 \cdots c_i) \cdot p + c_{i+1}) \bmod M$ . This guarantee, with the correct choice of  $p$  and  $M$  like the one below, the absence of overflow.

### 1.3.2 Hash table characteristics

Considering what was said above, the hash table has these characteristics:

- The dimension of the index vector corresponds to the  $M$  value.
- We resolve the collisions with the open chain method: every position in the table is a pointer to the data that share the same hash value.
- Each node of the open chain is composed by the string (i.e., the  $n$ -gram) as key and the counter as value.

### 1.3.3 Choice of hyperparameters

For the  $M$  and  $p$  value in generale there is a rule of thumb that says to pick both values as prime numbers with  $M$  as large as possible. Starting from this:

- $p$ : Must be grether than the dimension of the alphabet to reduce the probability of collisions. In the computing we analyse the corpus byte-by-byte, so if the  $p$  value is larger thant 256 it will be fine. We took  $p = 257$ , the first prime number after 256.
- $M$ : The choice of  $M$  determ the fill factor of the hash table, that can be defined as  $\alpha = \frac{\text{busy\_buckets}}{\text{table\_dimension}}$ . We compute it empirically as  $M = 10000019$ ; this choice allow us to have a good fill factor (i.e., 0.75 circa) in a big corpus in order to have a good compromise between memory usage and speed.

### 1.3.4 Overflow

Thanks to these choice and the calculation tric for the hash function, defined in Section 1.2, we are sure that we haven't overflow error if we store the intermediate result in 32 bits variable.

In fact, suppose that  $H_i$  can store in 32 bits we can prove that the intermediate variable used can be store in 32 bits: in the worst case we have that  $intermediate\_var\_dim = (M - 1) \cdot p + c_{max} < 2^{32}$ .

## 1.4 Statistics

Once we have populated the hash table structure we have to print the statistics of our  $n$ -gram. In some case print the distribution (i.e., the PDF) of our dataset is usefull, but in this case this means print the frequency of all encountered word: for a big corpus this is not reccomended due to the high number of unique  $n$ -gram.

We have considered the below statistics:

- Text statistics

We cosidered the follow text statistics:

- The  $K$   $n$ -gram with the highest frequency (with their frequency). The parameter  $K$  can be configured at compilation time with the parameter  $TOP\_K$  (default  $TOP\_K = 10$ ).
- The number of unique  $n$ -gram.

- Hash table performance

We considered the follow statistic to evaluate the hash table performance:

- The mean and the max length of the list.
- Load factor, i.e. the ratio between the total elements and the total buckets. This corresponde with the average langht of chain.
- Fill factor, i.e. the ratio between the busy bucket and the total bucket.

## 1.5 Tools

To validate and test our software we use two main tools: the sanitizers and the profiler.

### 1.5.1 Sanitizers

A sanitizer is a dynamic code analyzer integreted in the compiler that allow us to discover runtime errors in out code. The tool instruments the code to store metadata for each variable and this allow to detect runtime errors that the compiler can't see because it works in a static way.

The sanitezer is usefull during the developmet phase. In the release version we compile with the sanitizer because it introduce complexity and brings to a performance degradation.

The sanitizers tool that we used are the following:

- **Address (ASan):** Is usefull for the detection of memory corruction errors like buffer over flow, use after free and memory leak.
- **Undefined (UBSan):** Is usefull for the detection of undefined behavior in the code, i.e., partion of code that isn't conformant with the standard.

### 1.5.2 Profiler

Profiling is a technique for analysing the performance of our software: it is usefull to understand where the code spends most of its time and how many functions are called.

There are two techniques that profilers use: sampling, that is more efficient but less precise; instrumentation, that is very accurate but more expensive.

In our project we used the [GPerfTools](#). We install it with HomeBrew, but this doesn't install the *pprof* tool for the analysis. The new version was found [here](#), and it is developed by Google in Go.

## 1.6 Execution

To execute the project you have to execute one of the shell script in [exec folder](#). The *download\_data.sh* is an helper script that is used in the other.



## 2 Sequential

In this chapter we present the main aspect of the sequential implementation to count the  $n$ -grams in an input, that is in [sequential](#) GitHub folder.

### 2.1 Pipeline

Let's look the pipeline of our sequential algorithm. This was implemented in [main.c](#) file, that call some [helper functions](#).

1. We map the input file, that must be `data/input.txt`, into memory using the POSIX `mmap` system call. This allows us to access the file content as a byte array, letting the operating system handle paging efficiently. Because we only read the file in the sequential order, we inform the kernel about this with the `madvise` function with the `MADV_SEQUENTIAL` flag.
2. We allocate memory for the hash table.
3. We use a circular buffer of size `N_GRAM_SIZE` to maintain the current sliding window of words, i.e. the words that compose the current  $n$ -gram.
4. We fill the buffer with the initial  $N - 1$  words using the `get_next_word` function; if the input file doesn't contains  $N - 1$  words, we return. This helps us to maintain clear the loop across all the file.
5. We iterate through the rest of the file word by word:
  - a) Read the next word and write it into the circular buffer.
  - b) Construct the current  $n$ -gram string by concatenating the words in the buffer starting from the correct head.
  - c) Use the `add_gram` function in [hash\\_table.c](#) file to add the  $n$ -gram to the hash table. If the  $n$ -gram already exists, its counter is incremented; otherwise, a new node is created and inserted into the bucket's chain.
  - d) Advance the circular buffer head, effectively sliding the window one word forward.
6. Collect and print the statistics. See the Section 2.3
7. Release the memory allocated for the hash table and return.

## 2.2 Hash table

The major complexity is in the `hash_table.c` file, that modeling the hash table structure and its operation.

Now we see the main characteristics of this implementation:

- In the Section 1.3.4 we said that we must using almost a 32bits variable to avoid the overflow of intermediate value during the index calculation. To obtain this goal we have used the `uint_fast32_t` C type that guarantee the faster type that have almost 32bits.

## 2.3 Statistics

To extract the statistics described in Section 1.4, we implemented two function in `statistics.c`. These functions are responsible for the generation of text statistics and hash table performance; the main is responsible for the printing of the results.

### 2.3.1 Text statistics

For the text statistics, we explored two different approaches, and we choose the second one.

#### Two steps approach

The algorithm followed these steps:

1. Traverse the whole hash table structure to count the total number of unique  $n$ -grams ( $N$ ).
2. Allocate a temporary array of Node pointers of size  $N$ . This allocation was obtained using the `malloc` function. Using a Variable Length Array (VLA) allocates memory on the stack, which is faster but limited in size. Using `malloc` allocates memory on the heap, which is necessary when  $N$  is unknown or large to avoid stack overflow.
3. Populate the array with the nodes from the hash table.
4. Sort the entire array using the standard C library function `qsort`, with a custom comparator that orders nodes by frequency in descending order.
5. Print the top  $K$  elements from the sorted array and the count of unique  $n$ -grams.
6. Free the allocated array.

#### Top-K approach

This is the algorithm that we implemented in our code.

1. Allocate a small array *top\_ngrams* of pointers of size *TOP\_K*, i.e., the number of top elements requested.
2. Iterate through the whole hash table only once. For each unique *n*-gram:
  - a) Increment the counter of unique *n*-grams.
  - b) If the *top\_ngrams* array is not yet full, add the current *n*-gram to it.
  - c) If the array is full, find the element within *top\_ngrams* that has the minimum frequency.
  - d) Compare the current *n*-gram's frequency with this minimum. If the current *n*-gram is more frequent, it replaces the minimum element in the array.
3. Finally, sort the *top\_ngrams* array in descending order and print the results.

### Complexity Analysis

Let  $N$  be the number of unique *n*-grams and  $K$  be the number of top elements to find.

- **Time Complexity**

In the first approach the bottleneck is the sorting step, which takes  $O(N \log N)$ . In our implementation we pass through the hash table only once, so we have  $O(N)$ ; for each *n*-gram we pass over  $K$  elements, so in total we have  $O(N \cdot K)$ . Since  $K$  is usually very small, the total time complexity is  $O(N)$ .

- **Space Complexity**

The first approach requires  $O(N)$  extra space for the temporary array. The second approach requires only  $O(K)$  that is irrelevant for small  $K$ .

### 2.3.2 Hash table performance

For the hash table performance the pipeline is the following:

1. Initialize counters, *total\_elements*, *busy\_buckets*, and *max\_chain\_len*, to 0.
2. Iterate through the whole *buckets* array of the hash table.
3. For each bucket:
  - a) If the bucket is not empty increment *busy\_buckets*.
  - b) Traverse the open chain of bucket to count the nodes.
  - c) Update *max\_chain\_len* if the current chain length is greater than the current maximum.
  - d) Add the chain length to *total\_elements*.
4. Print the results.

## Complexity Analysis

The time complexity is  $O(M + N)$ , where  $M$  is the number of buckets and  $N$  is the total number of unique  $n$ -grams: we visit every bucket and every node exactly once.

The space complexity is  $O(1)$ : we only use a few variables for counting.

## 3 Parallel

In this chapter we present the main aspect of the parallel implementation to count the 3-grams in an input.

## 4 Analysis

# Bibliografia

- [1] Jakub PACHOCKI e Jakub RADOSZEWSKI. «Where to Use and How not to Use Polynomial String Hashing.» In: *Olympiads in Informatics 7* (2013).