UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Ingegneria

Corso di Laurea in Ingegneria Informatica

Parallel Computing

Parallel Trigrams

Edoardo Sarri

7173337

Febbraio 2026

Indice

Elenco delle figure

Listings

1 Introduction

The main goal of this project is to count the n-gram (the occurencies 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 tow implementations we will evaluate the performance in Chapter 4, with particular focus on the speed-up.

1.1 Features

Obviously the main goal is two count the n-gram. Others features are:

- We can define the n value (of the n-gram) at compilation time passing the parameter N_GRAM_SIZE. By default is $N\_GRAM\_SIZE = 3$.
- There is a stride k that allow us to skip k word when we compute the next n-gram. We can provide it during the compilation with $STRIDE$ parameter. By default is $STRIDE = 1$.
- At the end of computation, the system print some statistics (see Section 1.4) abount the text and abount the hash table performance.

1.2 Pre-processing

The pipeline that we followed for the input pre-processing consists in collect the input file and normalizating it.

1.2.1 Input collection

The file that we will analyse must be in the *data/* folder and must be named as *input.txt*.

1.2.2 Normalization

We want normalizing the input so that similar words will be rappresented by the same token, i.e., we want create equivalence classes that contain similar words. This help us to reduce the dimension of the word dictionary that composes the tringrams.

We addressed this in these ways:

- Bring all characters in lower case form.
- Remove all the punctuation like periods, commas and exclamation points.

- We mantain only one type of separatore between the words, i.e. single space. This help us when we have to bound a word.

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 has table and the caratteristichs 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

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

This approch 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 $H(c_1 \cdots c_i c_{i+1}) = (H(c_1 \cdots c_i) \cdot p + c_{i+1}) mod M$. This garantee, 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 corresponde 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 load factor of the hash table, that can be defined as $\alpha = \frac{number\_of\_stored\_element}{table\_dimension}$. We can compute in the way that of can obtain $\alpha = \frac{expected\_unic\_n-gram}{1.5 \times expected\_unic\_n-gram} \approx 0.67 < 1$, that usually is a good load factor. To determine the $expected\_unic\_n - gram$ the best option is use an empiracal estimate sampling from the corpus like $unic\_sampling\_n - gram \times \frac{total\_words}{salpled\_word}$. For our goal this isn't a foundamntal factor, so we took simply the prime number after 150K, i.e. $M = 150K$.

1.3.4 Overflow

Thanks to these choice and the calculation tric for the hash function, defined in Section 1.2.2, 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 thei 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 α, i.e. the ratio between the busy bucket and the total bucket.

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.

1. We take the text input file, that must be named as *input.txt* and must be in *data* directory, and we pre-precessing it. The implementation, that is my_utils.c file: write all the characters except the space and the punctuation; between two token we write ony a single space. After this step we have a normalized corpus in *data/normalized_file.txt*.
2. We allocate memory for the hash table.
3. We iterate over all the *n*-gram in this normalized file and stop when there aren't another *n*-gram to analyse.
   a) Find the next *n*-gram with next_ngram function.
   b) Use the `add_gram` function in hash_table.c file to add the *n*-gram in out hash table. This function increase the related counter if current *n*-gram match one already found; althought we insert as new node in the chain and initialize the counter to one. At the end it return with the cursor in the initial position (relative to this function call) of the corpus.
   c) Move *STRIDE* words forward in the corpus.
4. Colects and prints the statistics.
5. Release the memory allocated for the hash table.

2.2 Hash table

The major complexity si 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 garantee 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.

2.3.1 Text statistics

For the text statistics the algorithm follows the steps below:

1. Traverse the whole hash table statucture to count the total number of unique n-grams.
2. Allocate a temporary array of `Node` pointers of size N, where N is the numbers of unique n-grams. This allocation was obtained using the malloc function: using the variable lenght array (i.e., specifing the dimension of the array) the array is allocated in the stack, that is faster but smaller; using the malloc function the array is allocated in the heap memory. When the dimension of the array is unknown or very large it's better to use the dinamic allocation to avoid stack overflow.
3. Populate the array with the nodes in the hash table.
4. Sort the 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.

2.3.2 Hash table performance

For the hash table performance the pipeline is the following:

1. TODO

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).