UNIVERSITÀ DEGLI STUDI FIRENZE

Scuola di Ingegneria

Corso di Laurea in Ingegneria Informatica

Parallel Computing

Parallel Trigrams

Edoardo Sarri

7173337

Dicembre 2025

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 pass the n value (of the n-gram) at compilation time.
- There is a stride k that allow us to skip k word when we compute the n-gram.

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

We have to collect some text to analyze. In our experiment we use *.txt* file.

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.

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** [1]: given the string $c_1 \cdots c_n$, its hash value is defined by

$$H(c_1 \cdots c_n) = \sum_{i=1}^{n} (c_i \cdot p^{i-1}) mod M.$$

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 have that $H(c_1 \cdots c_i c_{i+1}) = (H(c_1 \cdots c_i) \cdot p + c_{i+1}) mod M$. 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.

2 Sequential

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

2.1 Pipeline

Let's look the pipeline of our sequential algorithm. This was implemented in main.c file.

- 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 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 file in *data/normalized_file.txt*.
- We iterate over all the n-gram in this normalized file:
  - Find the next n-gram.
  - 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.
- Stop when there aren't another n-gram.
- TODOOOOO: visualizzazione.
- Release the memory allocated for the hash table.

2.2 Hash table

In Section 1.3 we have defined our approch to the hash function. Now we must implent the hash table with its functionalities.

Let's start from the key value of the function, p and M. In general 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. Thanks to normalizzation, described in Section 1.2.2, our dictionary dimention isn't to much big. We took $p = 101$.

- 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. $M = 150K$.

2.2.1 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 $intermediate\_var = (M-1) \cdot p + c_{max} < 2^{32}$.

2.2.2 Implementation

The implementation of the hash table is in hash_table.c file.

We have to observe two things:

- The n-gram dimension (i.e., the n value) is define in the *CMakeLists.txt* file. This give us the possibility to pass this iperparaeter from terminal when we execute the prorgam.

- Due to the osservation at the starting of the Section 2.2, we use to memorize the hash value the type `uint_fast32_t`. This garentee the maximum speedy type in the architecture that has almost 32 bits.

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).