UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Ingegneria

Corso di Laurea in Ingegneria Informatica

Parallel Computing

Parallel Trigrams

Edoardo Sarri

7173337

Dicembre 2025

Indice

1 Introduction **5**
1.1 Pre-processing 5
1.1.1 Input collection 5
1.1.2 Normalization 5
1.2 Hash table 5
1.2.1 Hash function 5
1.2.2 Hash table characteristics 6

2 Sequential **7**
2.1 Pipeline 7
2.2 Hash table 7

3 Parallel **9**

4 Analysis **10**

Elenco delle figure

Listings

1 Introduction

The main goal of this project is to count the trigrams (the occurencies of three 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 Pre-processing

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

1.1.1 Input collection

TODO

1.1.2 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.

1.2 Hash table

We have to be able to count the trigrams; to address efficiently this goal we will use the hash-table.

1.2.1 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 overflow absence of overflow.

This algorithm can be directly use for the hash value of a word, but it is useful also for calculate the hash value of our 3-gram: for a single string we can define c_i like the ASCII code of the i-th character; for the 3-gram we can define c_i like the hash value of the i-th word. With this consideration we must define four value: *p_char*, *p_grams*, *M_char* and *M_grams*.

1.2.2 Hash table characteristics

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

- The dimension of the index vector is of *M_grams* position.
- 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 3-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 3-grams in an input.

2.1 Pipeline

We suppose that we have already did the normalizzation of the input. The pipeline is the above:

- We iterate throught the input and consider the current 3-gram.
- We calculate the hash value of this 3-gram.
- We scrool throught the whole list related to the index calculate in the previous step:
  - If current 3-gram match one already found, we increase the related counter.
  - Althought we insert as new node in the chain and initialize the counter to one.
- We pass to the next 3-gram.

2.2 Hash table

In Section 1.2 we have defined our approch to the hash function. Now we must implent teh 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_char*: Must be grether than the dimension of the alphabet. Thanks to normalizzation, described in Section 1.1.2, our dictionary dimention is circa 26 (plus the space); so we took $p\_char = 37$.
- *p_grams*: Must be grether or equal to *p_char* to reduce the number of collisions. We took $p\_grams = 101$.
- *M_char*: We took $M\_char = 16777259$, i.e. the first prime number after 2^{24}.

- M_grams: We took $M\_gram = 150001$, i.e. the first prime number after $150000 = 1.5 \cdot expected\_unic\_trigram$, if we suppose that the unic 3-grams are 150K.

Thanks to these choice and the calculation tric fot the hash function, defined in Section 1.1.2, we are sure that we haven't overflow error if we store the intermediate result in 32 bits.

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 $intermediate\_var = M\_char \cdot p\_char + c_{max} \approx 6.2 \cdot 10^8 < 2^{32}$. The same evaluating can be do for the p_grams value.

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).