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 $N\_GRAM\_SIZE = 3$.
- 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 Input and normalization

The code working well, for semplifing the implementation, only with english sentences. This allow us to threat a characters as a single byte.

We use a test simple and small plain text file to understand the correctness of the program. This file came from the Wortschatz Leipzig project, that contains a big corpus of english text; in specific this link is the direct download of the file. For the final test and for the analysis we take a plain text from Hugging Face; we download the file untill its dimension reach 1GB.

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 maintain only the numbers and the letterns.

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

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

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 fill factor of the hash table, that can be defined as $\alpha = \frac{busy\_buckets}{table\_dimension}$. 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.
- **Memory (MSan)**: Detects the use of uninitialized memory. In MacOS it isn't supported, but it is present anyway.

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

The execution of the project requires two step: the download of test input or real imput (the one used for the analysis), simply executing `download_test.sh` script or `download_input.sh` script in `exec/` folder; the second spet is executing the script named `executing.sh`, always in the `exec/` folder. This last script execution execute the program on the file previusly downloaded intput text with all performance flag on. In the same folder there are also others script for the execution with profiling (different type) or with the sanitizers or in debug mode.

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 intput file doesn't contains $N-1$ words, we return. This helps us te 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 an 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. To handle tokens with a very high length we use a buffer with 256 bytes allocated in the stack.
   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 hash_table.c file, modeling the hash table structure and its operation.

2.2.1 Implementing choice

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.

2.2.2 Memory Arena

In a base implementation we made a `malloc` operation every *k*-gram added in the hash table. The malloc, while allocate the request portion of memory, allocate also its part for metadata (e.g., dimension of the allocation and the state of the block); this is a big memory overhead if we use frequent and small call to `malloc`. To avoid the overhead of the memory allocation, also in the way to optimize the code for the furure parallel implemention (`malloc` implies a unique system lock), we implemented a Memory Arena allocator. The arena works by pre-allocating a block of contiguous memory (16MB by default) and use it for each new *k*-gram insertion.

This approach allow us to reduce the numbers of `malloc` operation: we make one of this only when the current block is all busy. Another advantage is that in this way we reduce the fragmentation of the mempory and improve chache localy. The localy doen't improve during the scansion of the linked list: the node in a list aren't added in a sequential manner.

2.3 Statistics

To extract the statistics described in Section 1.4, we implemented two function in statistics.c. These functions are responsible fot 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 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 seond approach requires only $O(K)$ that is irrilevant 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

In this Chapter we consider the time performance of our project. We will focus on the difference and the speed-up between the sequential version, described in Chapter 2, and the parallel version, described in Chapter 3.

4.1 Sequential

Here we evaluate the performance of the sequential implementation of the program.

4.1.1 Arena allocator

In this test input file we obtain a completion time about 18 second. After the implemention that allow us to allocation block of contigous memory, we obtain a total execution time about 10 second. This is a huge improvment in out performance.

We see also the benefit of this approch during the freeing of the hash table. In the first version we took abound 2 second, while in the final version the free is irrilevan with 0.0027 seconds.

We saw the same effect also using the profiler. In the first version our code spend the approx whole time in system call operation (i.e., `malloc`); with the second approach the main time is spend in add gram operation.

4.1.2 Time analysis

Now we see the performance on the final test case.

4.2 Parallel

Bibliografia

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