UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Ingegneria

Corso di Laurea in Ingegneria Informatica

Parallel Computing

Parallel n-grams

Edoardo Sarri

7173337

February 2026

Contents

1 Introduction

The main goal of this project is to count the n-grams (the occurrences of n consecutive words) 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.

This is the repository of the project.

1.1 Features

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

- We can define the n value (of the n-gram) at compilation time passing the parameter N_GRAM_SIZE. By default, $N\_GRAM\_SIZE = 3$.
- 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

To simplify the implementation, the code works well only with English sentences. This allows us to treat a character as a single byte; if there are strange characters the code could treat them with undefined behavior.

The data sources come from a plain text collection of Hugging Face. We have a script that allows us to pass the desired size (in MB) and it handles the download in the correct directory with the correct name.

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 the following 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 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 approach to obtain the hash value is inefficient and dangerous due to 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 guarantees, 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 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 greater than the dimension of the alphabet to reduce the probability of collisions. In the computation, we analyze the corpus byte-by-byte, so if the p value is larger than 256 it will be fine. We took $p = 257$, the first prime number after 256.
- M: The choice of M determines 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 allows us to have a good fill factor (i.e., 0.75 circa) on a big corpus in order to have a good compromise between memory usage and speed.

1.3.4 Overflow

Thanks to these choices and the calculation trick for the hash function, defined in Section 1.2, we are sure that we won't have overflow errors if we store the intermediate result in a 32-bit variable.

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

1.4 Statistics

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

We have considered the below statistics:

- Text statistics
  We considered the following text statistics:
  - The K n-grams 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-grams.
- Hash table performance
  We considered the following statistics 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 corresponds with the average length of chain.
  - Fill factor, i.e. the ratio between the busy buckets and the total buckets.

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 integrated in the compiler that allows us to discover runtime errors in our code. The tool instruments the code to store metadata for each variable and this allows detecting runtime errors that the compiler can't see because it works in a static way.

The sanitizer is useful during the development phase. In the release version we compile without the sanitizer because it introduces complexity and brings a performance degradation.

The sanitizer tools that we used are the following:

- **Address (ASan)**: Is useful for the detection of memory corruption errors like buffer overflow, use after free and memory leak.
- **Undefined (UBSan)**: Is useful for the detection of undefined behavior in the code, i.e., portion 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 useful 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 steps:

1. The download of the input. We have to exec the `exec/download_input.sh` script, passing the desired dimension of the final file; examples are 500 (i.e. 500MB), 1GB, 1.5GB.
2. Finally we can execute one of the scripts in the `exec/` folder. Which script depends on our interest: see README for more details.

2 Sequential

In this chapter we present the main aspects of the sequential implementation to count the n-grams in an input, that is in the sequential GitHub folder.

2.1 Pipeline

Let's look at the pipeline of our sequential algorithm. This was implemented in the main.c file, that calls some helper functions.

1. We map the input file, which 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 sequential order, we inform the kernel about this using 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 contain $N-1$ words, we return. This helps us maintain the loop cleanly across the whole 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. To handle tokens with a very high length we use a buffer with 256 bytes allocated on the stack.
   c) Use the `add_gram` function in the 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 Section 2.3.
7. Release the memory allocated for the hash table and return.

2.2 Hash Table

The hash_table.c file models the hash table structure and its operations.

2.2.1 Implementation Choice

In Section 1.3.4 we said that we must use at least a 32-bit variable to avoid the overflow of intermediate values during the index calculation. To obtain this goal we have used the *uint_fast32_t* C type that guarantees the fastest type that has at least 32 bits.

We have analyzed the dimension of the hash table, i.e., the number of buckets. In our source code, for the sequential version, we have a table dimension of 10000019; in modern 64-bit architecture a pointer is 8 bytes; the allocation of just the hash table takes 8×10000019 bytes, which is circa 76 MB. This was taken into account when we constructed the hash table in the parallel version.

2.2.2 Memory Arena

In a basic implementation, we performed a `malloc` operation for every *k*-gram added to the hash table. The malloc, while allocating the requested portion of memory, also allocates its 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 calls to `malloc`. To avoid the overhead of memory allocation, and also to optimize the code for the future parallel implementation (`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 using it for each new *k*-gram insertion.

This approach allows us to reduce the number of `malloc` operations: we make one of these only when the current block is full. Another advantage is that in this way we reduce memory fragmentation and improve cache locality. Locality doesn't improve during the scan of the linked list: the nodes 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 functions in statistics.c. These functions are responsible for generating text statistics and hash table performance metrics; the main function is responsible for printing the results.

2.3.1 Text Statistics

For the text statistics, we explored two different approaches, and we chose 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 $O(K)$, which 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 the 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 aspects of the parallel implementation to count the 3-grams in an input. In particular, we describe the approach used to transform the sequential execution into the parallel one. OpenMP is the framework used.

The repository of the project is the same, and this is the corresponding folder for the parallel version.

3.1 Implementation Design

In this section we describe the idea that was used in the parallel implementation. We will treat the pipeline of the program in Section 3.2.

3.1.1 Map-Reduce

With this pattern we allocate a private hash table for each thread. We make n partitions of the input file (where n is the number of threads) and each thread treats its partition locally. When all threads have populated their hash table, we make a partition of the buckets, and each thread is responsible for merging its partition into a global hash table. This approach is reasonable because the hash is a deterministic function, and the hash of a given k-gram is the same for each thread.

In this way no lock is necessary and there is no contention. The program can scale up when the input dimension grows; we have only the limit of the RAM size.

3.1.2 Hash Table Dimension

In the sequential version we have a hash table with 10000019 buckets. We have evaluated also the reduction of the local hash table dimension by a factor of $\frac{1}{T}$. This is a good idea if memory is a problem, because it reduces the dimension of the table of the single thread, and this makes sense because each thread works only on a partition of the whole input file; in total we have circa $76 \times T$ MB allocated for the hash table, where T is the number of threads. The downside of this approach is that during the merge phase we have to handle concurrent writes on the global hash table; using the same dimension for the local and global hash table allows us to avoid the concurrent write on the global structure, as we can map local buckets to global buckets 1:1 without collisions between merging threads.

3.1.3 Arena Management

In the sequential version, a memory arena is used to manage node allocations efficiently. In the parallel version during the merge phase, multiple threads may need to allocate new nodes in the global hash table simultaneously. Using a single global arena requires protecting the allocation function with a lock, creating a bottleneck due to high contention.

To address this efficiently, we implemented a thread-local arena: at the beginning of the merge phase, each thread creates a temporary private arena; the allocation of memory to create a new node doesn't touch the global arena, but only the private arena and this means no synchronization. At the end of the merge phase, we utilize a connection mechanism: since our arena (both local and global) is implemented as a linked list of memory blocks, we can transfer ownership of these blocks from the fork thread to the main thread simply appending these chains. This operation is performed in a minimal critical section and involves only pointer manipulation, i.e. the time is in $O(1)$ regarding data size.

The dimension of blocks in the local arena is 1MB. This minimizes the frequency of `malloc` calls, which implies a lock in the system and so an overhead.

We can see this pattern as a nested map-reduce: the first is a data map-reduce; the second is a memory map-reduce. This avoids the bottleneck that usually is present in the classical reduce phase.

3.1.4 Statistics

The extraction of statistics is the same as the sequential version (see Section 2.3) and it is performed sequentially by the main thread after the parallel region has completed. The execution time of this step is irrelevant compared to the other parts.

3.2 Pipeline

The pipeline of the map-reduce approach implementation is composed of three steps: initialization of the variables, population (Map) of the local hash table, and merging (Reduce) the local hash table into the global one. The code of this part is mainly in the `populate_hashtable` function in the main_functions.c file.

3.2.1 Initialization

Before entering the parallel region, the main thread allocates the following resources:

- **Global Table**: The final hash table is allocated but remains empty.
- **Local Tables Array**: We allocate an array of pointers to HashTable structs of size `omp_get_max_threads()`. This function returns the number of threads that

OpenMP will allocate in a parallel region; if you use the classic function like `omp_get_num_threads()` out of a parallel region, the function returns 1.

We enforced the variable scope passing to the OpenMP clause `default(none)` on the parallel region directive. This avoids accidental race conditions or unintended sharing. More in detail we have defined as:

- `shared(local_tables, global_table)`: These structures are accessed by all threads for writing their local instance and merging into the global one.
- `firstprivate(start, end)`: The input buffer pointers are copied to each thread's stack. This provides a minor performance benefit.

3.2.2 Populate

This phase, the map phase, is executed in a parallel region, where each thread populates its own private local hash table. As discussed above, this guarantees no race conditions and allows for maximum scalability with the input file size.

The parallel region isn't created with a simple `#pragma omp parallel for` on the mapped file buffer. The reason is that the hand made chunking is more useful and powerful in this case, leaving more flexibility to us to define the n-gram boundaries.

1. **Chunking**: The input file size is divided by N to calculate the `chunk_size`, where N is the number of threads. In this way each thread is assigned to a theoretical range $[start, end)$.
2. **Alignment**: Since a mathematical division can land in the middle of a word, we apply an alignment correction. If a thread detects that the character immediately preceding its `start` pointer is alphanumeric, it means the previous word was cut. The thread advances its `start` pointer until the beginning of the next valid word. This was done for each thread but not for the main thread, because its start is already set and it can start immediately the computation.
3. **Boundaries**: To handle n-grams crossing chunk boundaries, we define the ownership of a n-gram by the start of this n-gram: the n-gram is in the thread i zone if its first word is in the thread i zone.
4. **Buffer**: Before the main loop, we fill a circular buffer with $N_{gram} - 1$ words like in the sequential version.

3.2.3 Merge

This phase, the reduce phase, begins with a barrier: after this point all threads have populated their own local hash table and they can start to merge the local hash table into the global one.

- **Bucket-Parallelism**: Instead of parallelizing over threads (which would require locking the global table), we parallelize over the bucket indices of the hash table. This means that each thread is assigned to a partition of the global table and of each local table: no lock or mutex are required because only one thread works on the cell i of each table (local and global).
- **Zero-Copy Node Relinking**: When merging a local bucket into a global one, if an n-gram is found we sum the counters, if it's not found in the global table we simply relink the existing node pointer from the local table to the global table; this avoids several calls to `malloc`. At the end of the parallel region, we use the `arena_connect` function to append the local memory arenas (containing the relinked nodes) to the global one.
- **Dynamic Scheduling**: We use a dynamic assignment of the partition to the thread due to the sparsity of the hash table. In this way if a thread finishes earlier with respect to the others, all cores remain busy.
- **False Sharing**: With the usage of `dynamic` we could have a potential performance killer with false sharing: since multiple pointers (8 bytes each) fit in a single cache line (typically 64 bytes), adjacent buckets share the same cache line; by default we have `block_size` $= 1$, so if two threads process adjacent buckets, they would continuously invalidate each other's cache lines. So we have defined the block size as $(64/\texttt{sizeof(Node*)}) \times 16$: each chunk is aligned with cache lines; the chunk is large enough (16 cache lines, 1024 bytes) to amortize the OpenMP scheduling overhead. This empirical dimension must be larger if during the profiling we see that the program spends too much time in `malloc` function.
- **Implicit Barrier**: At the end of a `#pragma omp for` there is an implicit barrier. Inserting a `nowait` clause would be a mistake, because it implies that the local table of the other thread will be freed causing a segmentation fault.

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.

We analyze the 3-grams for simplicity and we print only the top-3 frequent 3-grams. We will use the release compiling and the profiling.

We will use three dimensions for the input: a very small dataset (i.e., 100KB), in which we expect that the sequential version obtains better performance; a medium input plain text (i.e., 100MB and 500MB); a huge, but not too much for time execution, input file (i.e., 2GB).

4.1 Arena Allocator

We have evaluated the different execution times before and after the implementation of the arena allocator; before this type of allocation we used a huge number of `malloc` operations. The input file used and the old code are no longer available.

The result was: with the first implementation we had a completion time of about 18 seconds; afterwards we obtained a total execution time of about 10 seconds. This is a huge improvement in our performance.

We can also observe the benefit of this approach during the freeing of the hash table. In the first version we took around 2 seconds, while in the final version the free is irrelevant with 0.0027 seconds. We saw the same effect also using the profiler: in the first version our code spent practically the whole time in system call operations (i.e., `malloc`); with the second approach the main time is spent in the `add_gram` operation.

4.2 Text Statistics

The text statistics are the same for both sequential and parallel versions of the code.

- 100KB
  The three most frequent 3-grams are *randallsville new york*, *variety of quartz* and *in the uk*; their frequencies are respectively 24, 10 and 9.
- 100MB
  The three most frequent 3-grams are *one of the*, *as well as* and *a lot of*; their frequencies are respectively 7774, 6995 and 4264.
- 500MB

The three most frequent 3-grams are *one of the*, *as well as* and *a lot of*; their frequencies are respectively 38758, 34863 and 20851.

- 1GB
  The three most frequent 3-grams are *one of the*, *as well as* and *a lot of*; their frequencies are respectively 53934, 48790 and 29367.
- 1.5GB
  The three most frequent 3-grams are *one of the*, *as well as* and *a lot of*; their frequencies are respectively 118445, 106364 and 64775.
- 2GB
  The three most frequent 3-grams are *one of the*, *as well as* and *a lot of*; their frequencies are respectively 157788, 142530 and 86345.

4.3 Hash Table Performance

The hash table performance is the same for both sequential and parallel versions of the code.

- 100KB
  The maximum chain length is 2 nodes and the average is 1. The load factor (i.e., elements/buckets) has a value of 0.0017 and the fill factor (i.e., busy/total) of 0.0017.
- 100MB
  The maximum chain length is 11 nodes and the average is 1.69. The load factor (i.e., elements/buckets) has a value of 1.1563 and the fill factor (i.e., busy/total) of 0.6853.
- 500MB
  The maximum chain length is 20 nodes and the average is 4.76. The load factor (i.e., elements/buckets) has a value of 4.7155 and the fill factor (i.e., busy/total) of 0.991.
- 1GB
  The maximum chain length is 23 nodes and the average is 6.30. The load factor (i.e., elements/buckets) has a value of 6.2910 and the fill factor (i.e., busy/total) of 0.9981.
- 1.5GB
  The maximum chain length is 35 nodes and the average is 12.22. The load factor (i.e., elements/buckets) has a value of 12.2241 and the fill factor (i.e., busy/total) of 1.0000.

- 2GB
  The maximum chain length is 40 nodes and the average is 15.54. The load factor (i.e., elements/buckets) has a value of 15.5416 and the fill factor (i.e., busy/total) of 1.0000.

4.4 Time Statistics

The time statistics are obviously different between sequential and parallel versions of the code. We analyze the aggregates for the dimension of the input file.

4.4.1 100KB size

As we can see above, the total execution time is bigger in the parallel version. For small datasets this is due to the overhead of the fork-join operation.

It isn't useful to calculate the speed-up. This test is only useful to observe that with few data the parallel approach doesn't gain the same advantage as its execution on a big corpus.

- Sequential
  - Total execution time: 0.0200 seconds.
  - Populating hash table time: 0.0099 seconds.
  - Freeing the hash table is irrelevant (i.e., the system prints 0.0000 with 4 decimal places) in this case.
  - Statistics extraction: 0.0101 seconds.
- Parallel
  - Total execution time: 0.0568 seconds.
  - Populating hash table time: 0.0483 seconds.
  - Freeing the hash table is irrelevant (i.e., the system prints 0.0000 with 4 decimal places) in this case.
  - Statistics extraction: 0.0084 seconds.

4.4.2 100MB size

With more data we can observe the efficiency of the fork-join approach. We have obtained a speed-up of $sup = \frac{t_{seq}}{t_{par}} = 4.97$.

- Sequential
  - Total execution time: 3.3305 seconds.
  - Populating hash table time: 3.0567 seconds.

  - Freeing the hash table is irrelevant (i.e., the system prints 0.0000 with 4 decimal places) in this case.
  - Statistics extraction: 0.2723 seconds.
- Parallel
  - Total execution time: 0.6696 seconds.
  - Populating hash table time: 0.5554 seconds.
  - Freeing the hash table is irrelevant (i.e., the system prints 0.0000 with 4 decimal places) in this case.
  - Statistics extraction: 0.1125 seconds.

4.4.3 500MB size

Before arriving at the last experiments, we make same experiments with a slow increment of input dimension.

We have obtained a speed-up of $sup = \frac{t_{seq}}{t_{par}} = 7.11$.

- Sequential
  - Total execution time: 31.3847 seconds.
  - Populating hash table time: 29.7151 seconds.
  - Freeing the hash table is: 0.0060 seconds.
  - Statistics extraction: 1.6621 seconds.
- Parallel
  - Total execution time: 4.4133 seconds.
  - Populating hash table time: 2.6503 seconds.
  - Freeing the hash table: 0.0150.
  - Statistics extraction: 1.7434 seconds.

4.4.4 1GB size

We have obtained a speed-up of $sup = \frac{t_{seq}}{t_{par}} = 8.09$.

- Sequential
  - Total execution time: 53.7854 seconds.
  - Populating hash table time: 51.0290 seconds.
  - Freeing the hash table is: 0.0118 seconds.
  - Statistics extraction: 2.7377 seconds.

- Parallel
  - Total execution time: 6.6462 seconds.
  - Populating hash table time: 3.8822 seconds.
  - Freeing the hash table: 0.0254.
  - Statistics extraction: 2.7364 seconds.

4.4.5 1.5GB size

We have obtained a speed-up of $sup = \frac{t_{seq}}{t_{par}} = 10.30$.

- Sequential
  - Total execution time: 207.6583 seconds.
  - Populating hash table time: 199.0169 seconds.
  - Freeing the hash table is: 0.0197 seconds.
  - Statistics extraction: 8.6142 seconds.
- Parallel
  - Total execution time: 20.1456 seconds.
  - Populating hash table time: 11.7647 seconds.
  - Freeing the hash table is: 0.0507 seconds.
  - Statistics extraction: 8.3235 seconds.

4.4.6 2GB size

This is absolutely the most interesting scenario because we can see that the parallel version is slower than the sequential one.

With a low dimension of the input file, it is possible to store all (or the main part) the information in the L3 cache. With a huge dataset we move from and to the main memory and this becomes a Memory-Bound task and with all cores working we have a memory bandwidth saturation.

- Sequential
  - Total execution time: 346.5373 seconds, i.e. 5 minutes and 46 seconds.
  - Populating hash table time: 333.3192 seconds, i.e. 5 minutes and 30 seconds.
  - Freeing the hash table: 0.0367 seconds.
  - Statistics extraction: 13.1746 seconds.

- Parallel
  - Total execution time: 357.9196 seconds, i.e. 5 minutes and 57 seconds.
  - Populating hash table time: 335.2379 seconds, i.e. 5 minutes and 35 seconds.
  - Freeing the hash table: 0.1226 seconds.
  - Statistics extraction: 22.5551 seconds.

Bibliography

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