Parallel Computing 2024 – 2025

Bigrams / Trigrams Assignment

Chiara Polenzani
chiara.polenzani@edu.unifi.it

Abstract

This work presents the implementation of a system for extracting and analyzing bigrams and trigrams of both words and characters contained in a certain group of texts. Two generic functions were developed to compute n-grams for words and characters, respectively. These functions were integrated into a sequential and parallel version of the code using the directives given by OpenMP. The program creates four distinct histograms representing word bigrams, word trigrams, character bigrams, and character trigrams. Each histogram is then saved to a corresponding output file for further analysis.

1. Introduction

N-grams are contiguous sequences of `'n'` items, typically useful in the context of NLP. These items can be characters or words, depending on the desired granularity. The value of `'n'` determines the order of the N-gram. N-grams of texts are extensively used in text mining and natural language processing tasks. Their applications include: text classification, language modeling, plagiarism detection, machine translation, spelling correction and information retrieval.
In the case of this assignment are just considered bigrams and trigrams, which are sequences of 2 and 3 consecutive characters or words respectively, but the code developed can be easily extended to the case of n-grams of any length n. The texts considered in this assignment are all retrieved from the Gutemberg project website and were all downloaded in .txt format for a total of 10 MB. The texts are chosen in a way to have various sizes and to belong to different fields to ensure variability and provide a more robust evaluation of the n-gram extraction process.

2. Implementation

To count the occurrences of each bigram and trigram, an unordered map from the C++ Standard Library was chosen as the data structure for the histograms. In this map, the key is a string representing the bigram or trigram, while the value is an integer storing the number of times it appears in the input texts. This choice was motivated by the efficiency of an unordered map, which uses a hash table that gives a search, insertion, deletion time in the order of O(1) on average and O(n) in the worst case. Unlike a classic map, it does not maintain any order among the elements, which in this project was not necessary — sorting the results would have added unnecessary overhead.
The implementation is based on two generic functions designed to compute n-grams from either a sequence of words or a sequence of characters. These functions are flexible and reusable, allowing the generation of bigrams, trigrams, or any other n-gram length by simply passing the desired value of n. Each function takes a vector of tokens and slides a window of fixed length n to construct every possible n-gram, incrementing its count in the unordered map.These functions are called from within the two main execution modes of the program: `SequentialExecution`, which processes files one after the other, and `ParallelExecution`, which distributes the work across multiple threads. Both functions receive the path to a folder containing text files to be analyzed. Before computing the n-grams, the text from each file is preprocessed; this step involves converting all characters to lowercase, removing punctuation, and replacing non-alphabetic characters with spaces. Once preprocessed, the text is tokenized.
The implemented functions calculate both the total and unique number of n-grams. The total count sums the frequencies of all n-grams, taking into account how many times each one appears. In contrast, the unique count considers only the number of distinct n-grams, counting each one once, regardless of its frequency.

N-grams	Bigrams	Trigrams
Characters	644	8.056
Words	623.867	1.387.868

Table 1. Numbers of different n-grams of letters and words in the texts

N-grams	Bigrams	Trigrams
Characters	103.687.920	103.687.770
Words	18.201.640	18.201.490

Table 2. Total number of n-grams of letters and words in the texts

2.1. Sequential version

The creation of the histograms of n-grams of letters and words is done with the function `SequentialExecution` which takes as inputs the path to a folder containing text files and an integer value that specifies how many times the execution should be repeated for performance measurement purposes. Text files are selected sequentially using the standard filesystem iterator. It does not return a value, but it computes and prints a summary of n-gram statistics, and writes the results to output files. For each file, the function counts the number of distinct n-grams, accumulates the total frequency of all n-grams and updates global maps to track all unique n-grams across files.

2.2. Parallel version

The `ParallelExecution` function receives as input the path to a folder containing text files and an integer that specifies how many times the execution should be repeated. Similar to the sequential version, the function does not return any value but computes and prints a summary of the n-gram statistics. Specifically, it calculates the total and unique counts for word bigrams, word trigrams, character bigrams, and character trigrams. The function starts by collecting the text files and stores their paths in a vector. It replicates this list a fixed number of times to simulate a larger dataset, avoiding repeated filesystem access during execution. The core logic of reading and processing each file remains the same as in the sequential function. The key difference lies in how the processing is parallelized. OpenMP is used to parallelize the loop that iterates over the files. This allows each file to be processed independently by different threads. The `#pragma omp parallel` directive marks the beginning of a parallel region in the code. Each thread creates its own local maps for storing the computed n-grams, which prevents any conflicts in shared memory. In addition, the clause `schedule(dynamic)` is used so that the files are divided among the threads, and each thread processes its assigned file(s). When a thread finishes processing its current chunk of work, it grabs the next available chunk. The dynamic scheduling helps ensure that the threads that finish early will not sit idle while others are still working. In addition, the number of iterations in the loop is dynamically divided among the threads. This means that each thread does not have a fixed amount of work; instead, once a thread finishes its assigned work, it can pick up the next available chunk of work.
This helps balance the workload and ensures that no thread remains idle while others are still processing. Once a thread finishes processing a file, it merges its results into shared global maps using a `#pragma omp critical` section because each local version could have elements in common with the one of another thread. This ensures that the updates to the shared maps are done safely, though it may introduce a slight performance bottleneck.

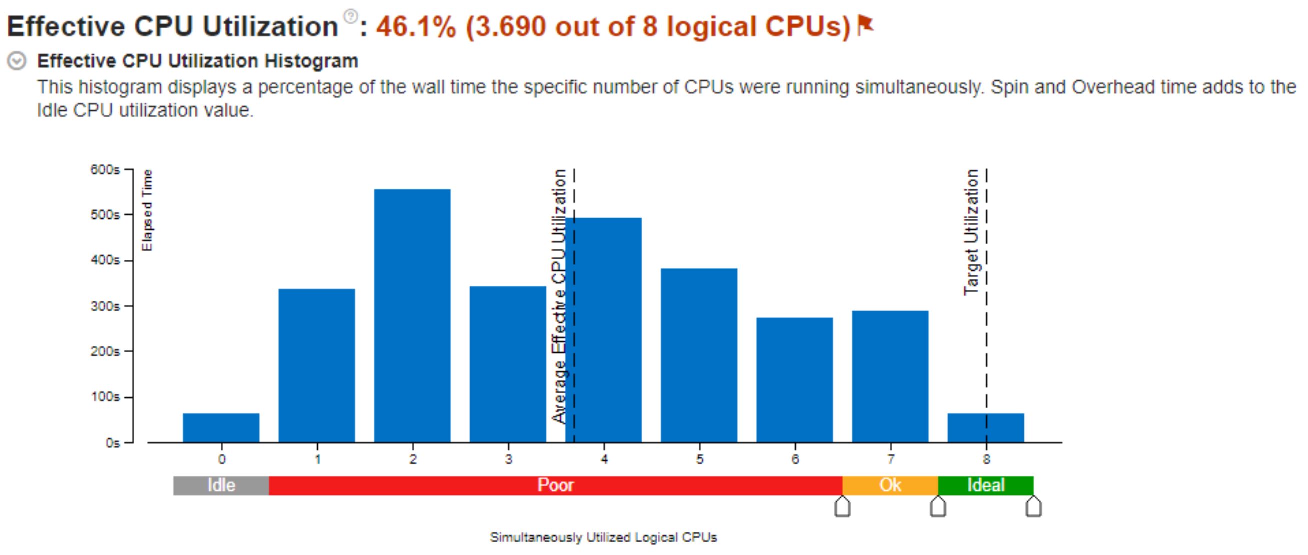
3. Profiling

Profiling consist in the recording and analysis of the behaviors that occurs during the code execution. With this purposes the Intel VTune Profiler has been used for studying the code performances. The sequential part is not included in the test.

As we can see in Figure 1, the threading level is not very high because the parallelization implemented suffers from a strong need of synchronization, which is unavoidable. The VTune graph shows that the program spends most of its time using only 2 to 5 threads concurrently. A significant portion of the wait time is wasted on inefficient synchronization. There is high contention on shared resources, such as semaphores, events, and files. Additionally, the presence of too many unnecessary threads increases overhead without improving performance. If we repeat the performance test considering only thread counts from 8 onward, we observe an increase in both logical and physical core utilization, reaching up to 53%. This happens because when using fewer than 8 threads, the simultaneous utilization of logical CPUs remains low, leading to underutilization of available cores.

4. Experiments and Results

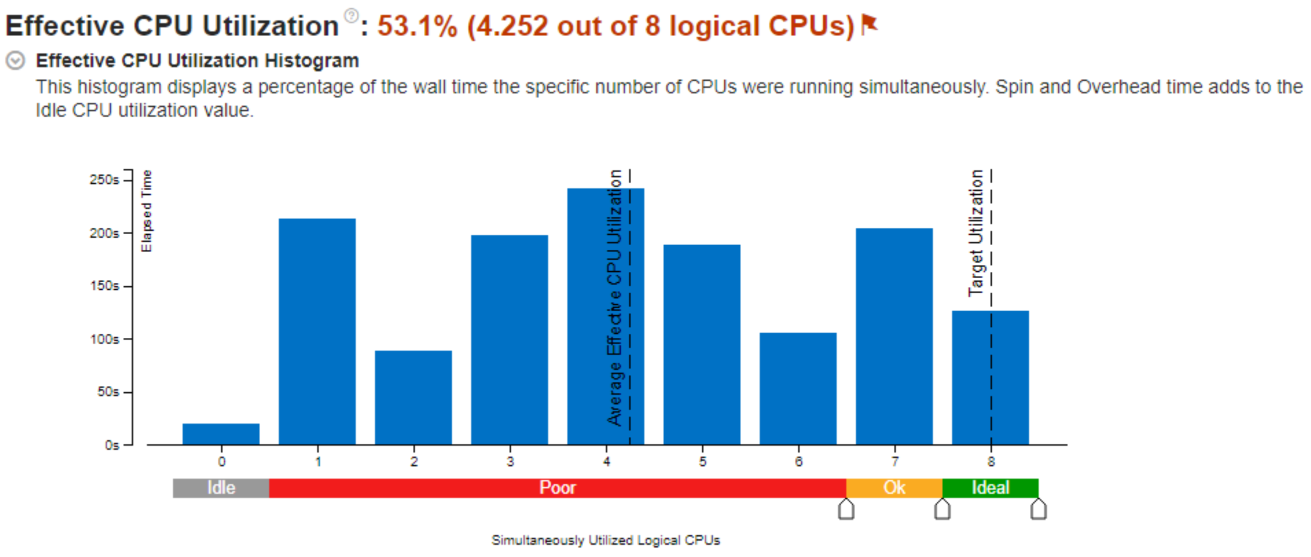
In this assignment two experiments have been executed with the purpose to evaluate the speedups between sequential and parallel version. This is done, at first, setting the size of the dataset at 100 MB and varying the number of threads involved in the execution, and then, the second experiment is to keep steady the number of threads but increasing artificially the total dimension of the texts set. For each of the two cases, 3 iterations of the two main functions are performed, in order to better evaluate the performances and avoiding to be biased by the singular specific executions. For the first test, the average execution times are measured, later used for computing the speedup with the respect to the sequential version.
To accurately evaluate the impact of the number of threads on parallel execution, tests were conducted one at a time. An inner for loop to iterate over different thread counts in a single execution was not used, as it produced unreliable timing results. This behavior is known in OpenMP: setting the number of threads and interactions with the operating system can introduce non-deterministic side effects. Therefore, to avoid interference between tests and obtain reliable


Effective CPU Utilization: 46.1% (3.690 out of 8 logical CPUs)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

Figure 1. Threading profile for `ParallelExecution` with all threads


Effective CPU Utilization: 53.1% (4.252 out of 8 logical CPUs)

Effective CPU Utilization Histogram

This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.

Figure 2. Threading profile for `ParallelExecution`

measurements, each test configuration was executed separately in a controlled manner. This approach ensures that the observed behavior more accurately reflects the actual performance for each thread count, avoiding spurious effects related to hardware or dynamic system resource allocation. The typical trend of the speedup is shown in Figure 3.
As we can see, initially, the speedup increases rapidly as the number of threads increases, showing a clear benefit of parallel execution. However, after reaching a peak around 8 threads, the speed improvement tends to stabilize.
This behavior can be interpreted using Amdahl's Law, which states that the maximum achievable speedup is limited by the fraction of the code that cannot be parallelized. In this context, operations like result merging, which are executed in critical sections, contribute to the sequential part of the code and limit scalability.

After that, the performances varying the total size of texts have been studied, keeping the number of threads at 8 (equal to the logical cores of the computer). This has been done just iterating multiple times into the folder containing the texts files. We considered data sizes ranging from 100 MB to 600 MB. To evaluate the average speed, each test was executed

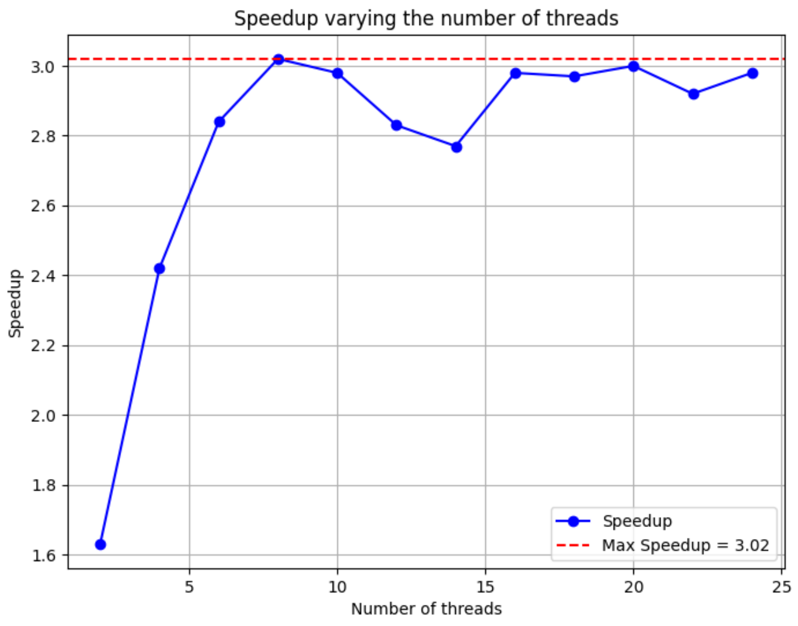


Figure 3. Speedup trend varying the number of threads

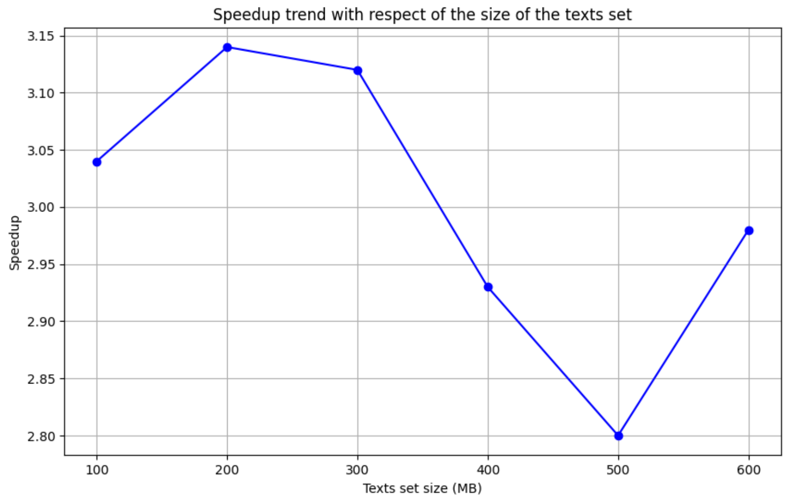


Figure 4. Speedup trend varying the size of the texts set

3 times. We have a bit strange behaviour for the speedup trend. In fact, as we can see from Figure 4 the speedup is not always increasing; initially, it is relatively high, thanks to a small number of files being duplicated many times, threads are well utilized without too much contention, with efficient cache usage and minimal overhead.
However, when the data set grows, the speed decreases significantly. This drop is mainly due to the increase in the number of files. As these files are repeated multiple times, the file I/O overhead increases, and the critical section used to merge local results likely becomes a performance bottleneck, limiting the benefits of parallelism. When the dataset size grows to 600 MB, the speed improves. Each thread handles a larger amount of work, and the relative impact of

synchronization and scheduling costs decreases. With larger workloads, parallel execution becomes more efficient overall as threads spend less time competing for shared resources and more time performing useful computations.

5. Conclusions

In this assignment we have seen that it is possible to use OpenMP directives to speed up the performance of a code that computes the histogram of characters and words (bigrams and trigrams). The results obtained show an improvement in the parallel version of the code. Performance as the number of threads varies follows Amdahl's law, so there is an initial increase in speedup that then stabilizes even for higher numbers of threads, with speedup values ranging from 1.6 to 3.0. When increasing the number of texts initially the speedup increases, then performance deteriorates until it increases again with sizes above 600 MB. In the type of parallelization used each file is assigned to a single thread, but one thread can process multiple files if there are many. This approach works but there may be better workload balancing. In this code, each thread enters the critical section multiple times per file processed. A possible modified version could reduce this to one entry per thread, helping to reduce the performance overhead associated with critical sections. In summary, parallelization has greatly improved the performance of this problem, but the results are highly variable and highly dependent on factors such as hardware, system resources, and other environmental conditions.