Parallel Computing 2023-2024
Midterm assignment

Matteo Trippodo
matteo.trippodo@edu.unifi.it

Abstract

This assignment has the aim to show how it's possible to create the histograms of bigrams, or trigrams, of letters and words contained in a certain group of texts. In the code this is done using a map as data structure for the histograms, searching for the n-grams in the texts and then updating them. The code that was initially written in a sequential way is then made parallel using the directives given by OpenMP.

1. Introduction

An "n-gram" is a sequence of n tokens which appear one after the other in a text, and, a token can be either just a letter or an entire word. Looking for the n-grams in a text can be useful in various way, for example, it can help to study the statistical properties of a language, which can be used in various fields, like Cryptography or even Data Mining.
In the case of this assignment are just considered bigrams and trigrams, which are trivially n-grams made of two and three tokens respectively, but the code developed can be easily extended to the case of n-grams of any length n.
The texts that are considered in this assignment are all included and retrieved from the site of Gutemberg project. Every text has been downloaded in *.txt* format directly from `https://www.gutenberg.org/` so that it can be easier to extract the contained text from the files.
The texts are chosen in a way to have various sizes and to belong to different fields and cultural backgrounds, so that there is any sort of bias on the founded n-grams. The group of texts has a total size of around 100 MB, but iterating multiple times though the folder of texts, during the experiments, it is reached up to a total size of 1 GB.

N-grams	Bigrams	Trigrams
Letters	59.148.945	41.860.743
Words	17.942.115	17.942.029

Table 1. Total number of n-grams of letters and words in the texts

N-grams	Bigrams	Trigrams
Letters	638	8.432
Words	3.278.847	9.974.663

Table 2. Numbers of different n-grams of letters and words in the texts

2. Implementation

The basic idea from which the implementations starts is that an histogram can be easily represented as a map of the standard C library, in which the keys are the strings of the n-grams and the values are integers that count the occurrences of each specific n-gram.
Actually, since our main purpose is just to count the n-grams in text, there is no need to use a classic map, but instead an unordered map can be used. Unordered maps are newer than the classic standard C library maps and in respect to the latter have some advantages regarding the access speed of the map elements and the retrieve of the respective value. In fact, since this type of map doesn't need the keys to be ordered they are implemented with an hash table that gives a search, insertion, deletion time in the order of $\mathcal{O}(1)$ on average and $\mathcal{O}(n)$ in the worst case. Instead, the classical maps are represented with a tree data structure, so every operation requires around $\mathcal{O}(log(n))$ and therefore on average the unordered map outperforms the regular map.

The implementation is based on two main functions `UpdateHistogramLetter` and `UpdateHistogramWord` which take as inputs a reference to the histogram object (a map of string and int), a reference to the text string and an integer representing the size of the n-gram to focus on. For the string and the map is used a pass by reference in order to avoid to copy every time the text string in input and the histogram data structure, which can be quite long. In this way we save time and memory in our code. Instead, the size of the n-gram is passed by copy since it will be just 2 or 3 and using a reference might cause an useless additional cost.
The first function iterates through the string of the text and founds all the letter n-grams that are composed by alpha-

betic characters, then it reduces every letter to lowercase and updates the histogram passed in input. The update is done using the operator [] which creates a new item of the map if the key doesn't already exist.

The second function deals with doing the same job but focusing on words. For doing that, fist of all, the string of the texts is converted into an object of the class `istringstream`, then, through an iterator, all words in the text are extracted and inserted in a vector. From the vector of words the n-grams of words are retrieved, cleaned from every not alphabetic characters and then the histogram is updated, following the same approach used for letters.

2.1. Sequential code

The creation of the histograms of n-grams of letters and words is done with the function `CreateHistogramSequential` which takes as inputs a reference to a vector of maps, one for the histogram of letter n-grams and one for words one, and an integer referred to the size of the n-grams. Inside this function the text files are selected sequentially using the standard file system iterator and opened. From each text the content is extracted as a string using the library `stringstream`. Then, the aforementioned functions for updating the histograms are called and, when they return, the file is closed. The function returns void since all the updates are done directly on the data structures provided in input.

2.2. Parallel code

The parallel versions are an evolution of the function just seen and they use the OpenMP directives in order to exploit multiple threads and speed up the code execution. This process is done in three slight different ways:

1. `CreateHisotgramParalleV1`: the function takes as inputs a reference to a vector of maps for the histograms of letter n-grams and the histogram of word n-grams, in addition to the size of the n-gram.

   First of all, the parallel section is created and for each thread a local version of the histograms is generated too.

   In this function the sequential scrolling of the texts is maintained, indeed it's done by only one thread at a time exploiting the directive `#pragma omp single`. In this way, only the fastest thread (the fist reaching the statement) opens the file, extracts the string of text and then append it to a vector of texts, which is shared between all threads.

   When all the files have been read the shared variable contains all the strings of texts and, therefore, using the directive `#pragma omp parallel` the work load can be distributed to the available threads. This implies that every thread calls the functions to update the histograms providing them as inputs their local version and one of the texts in the vector.

   In addition, the clauses `nowait` and `schedule(dynamic)` are used, so that the implicit barrier of the loop is bypassed and the work load is distributed in a dynamic way to the threads.

   In the end, we have the synchronization phase which aims to create a singular histogram from the various local versions of the threads. This must be done using critical sections because each local version could have elements in common with the one of another thread. Therefore, using other ways, for example, like an atomic operation (which is more lightweight and keeps the control of just the selected element and not of the entire data structure, as instead guaranteed with a critical section), will result in a race condition when multiple threads may try to update the same element of the output histogram.

   So, using critical section each thread updates the output histograms using all the elements of its local histogram and their counts. In order to reduce the overhead needed for the acquisition and release of the semaphore of the critical sections, in the implementation each thread remains inside the section until it have finished all its updates. In this way the number of critical sections is proportional to the number of the used threads.

   Moreover, with the aim to avoid possible deadlocks and the thread to block each other for different tasks, the critical sections are named differently for the letter and word histograms, so that if a thread is updating the letter histogram an other thread can work with the word one.

2. `CreateHisotgramParallelV2`: in this case the part of the access to the file is parallelized using the directive `#pragma omp parallel` and this is easily done because every text file is named with an integer index. For doing this, it's also needed an initial loop that iterates over the folder of texts and count them (this is done for allowing the code to be decoupled from the specific considered texts set).

   After that, as in the previous version, the parallel section is created, where each thread has its own version of the histograms.

   Using the directive `#pragma omp parallel` the various index of the loop, and therefore the texts, are assigned to the threads, using also the clauses `nowait` and `schedule(dynamic)`. In this way multiple files can be opened and being read in parallel, allowing also to the threads which finish early to continue the execution without need to wait for the others.

   As in the previous case, after the text file is opened each thread calls the functions for updating the his-

tograms passing them the text string of the just opened file.
After that, the synchronization phase starts, where each thread access to the two critical sections, one for the letter histogram and the other for the word histogram.

3. `CreateHisotgramParallelV3`: this function with respect to the aforementioned version differs for just a small thing. In this version the access to the text file is decoupled from the actual update of the local version of the histograms. In fact, while in the second version each thread reads the file and immediately updates its histograms, in this version every thread has a local vector of texts and in the first loop it just read the files according to the indexes given from the parallelization of the loop and then it update its histograms in a following parallelized loop. Also in this version the clauses `nowait` and `schedule(dynamic)` are included, so that the texts load is distributed dynamically to the threads and when a thread finished to read files it can immediately go to the next loop, thus speeding up the execution.

3. Profiling

Profiling consist in the recording and analysis of the behaviors that occurs during the code execution. It's a great way to understand how the code is actually running and provide insights on where we should focus our design efforts, highlighting the hotspots in the code and studying how much the threading is exploited. With this purposes the Intel VTune Profiler has been used for studying the code performances.
First of all, a general performance snapshot is performed considering the case of 10 consecutive runs of each function (the sequential one and the three parallel versions), in order to have an average estimation of the code performance. This first analysis report that the code has:

- **Logical Core Utilization**: 43.3 %.This means that on averages 3.467 cores out of 8 are used. Since on my computer threads work with hyper-threading, also the Physical Core Utilization can be evaluated and it is 55.1 % (2.205 out of 4);

- **Microarchitecture Usage**: 37.7 % of Pipeline Slots. In particular we have: Retiring: 37.7%, Front-End Bound: 24.5%, Bad Speculation: 10.7%, Back-End Bound: 27.1% (Memory Bound: 13.9%, Core Bound: 13.2%) ;

- **Memory Bound**: 13.9% of Pipeline Slots. In particular we have Cache Bound: 20.9% and DRAM Bound: 15.7% of Clockticks.

- **Vectorization**: 0.1% of Packed FP Operations. In fact we have x87 FLOPs: 0.2% of μOps and Non-FP: 99.8% of μOps.

From this brief report we can understand better how the code works and know what to expect in the experiments. As we can see, the threading level is not very high both because also the sequential part is included in the test, but even because the parallelization implemented suffers from a strong need of synchronization, which is unavoidable though.
Moreover, the quite high values of Front-End Bound and Core Bound suggest that the implementation can be improved taking particular cares on how the work loads are distributed on the cores and increasing the predictability of branches in the code, in order to help the compiler to optimize the code in a better way.
Furthermore, given the nature of the problem, the implementation cannot benefit from the use of vectorization since most of operations are not FLOPS and SIMD operations are not used.
In addition, for the sake of completeness, if we repeat the performance test considering only the parallel functions then we have an increment of logical and physical core utilization that goes to 70% (5.603 out of 8) and 80.6% (3.223 out of 4) respectively, while the values of the other metrics remain quite the same.

After that, an evaluation of the hotspots of the code has been performed with the respective tool of VTune. During multiple tests are popped up some particular statements and part of code that contribute more in percentage to the execution time. Some of them helped in changing the implementation in order to slightly improve the performance, but there are always some hotspots that occurs when the execution deals with the histogram data structures.
In fact, the most percentage of execution in related to the use, access, managements and update of the maps used for the histograms. After some tries and various versions, the final version of the code is the one which better reduced the costs.

Finally, it's possible to test the threading level of each parallel versions of the functions that creates the histograms. For each function are performed 20 iterations, considering the case of bigrams, and then the average results are evaluated. As we can see from the graphs in Figures 1-3 all the three versions have an effective CPU utilization around 50% and the last one seems to be the slightly better version. In all the case the most of wait time occurs when the code deals with semaphores and auto-reset events, needed for the synchronization of the threads, in order to create the two general histograms from the local versions of each thread.

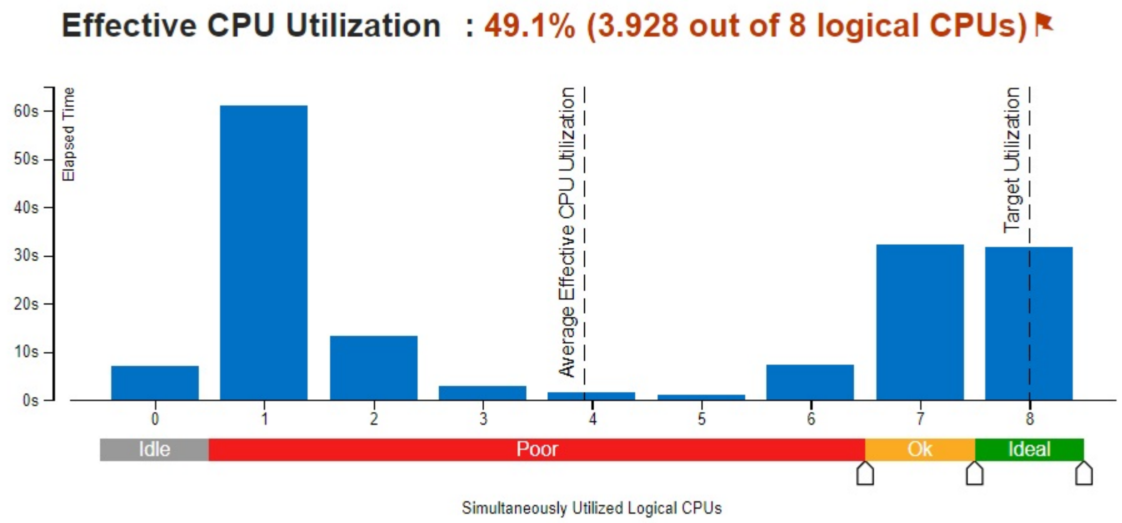

Figure 1. Threading profile for `CreateHistrogramParallelV1`

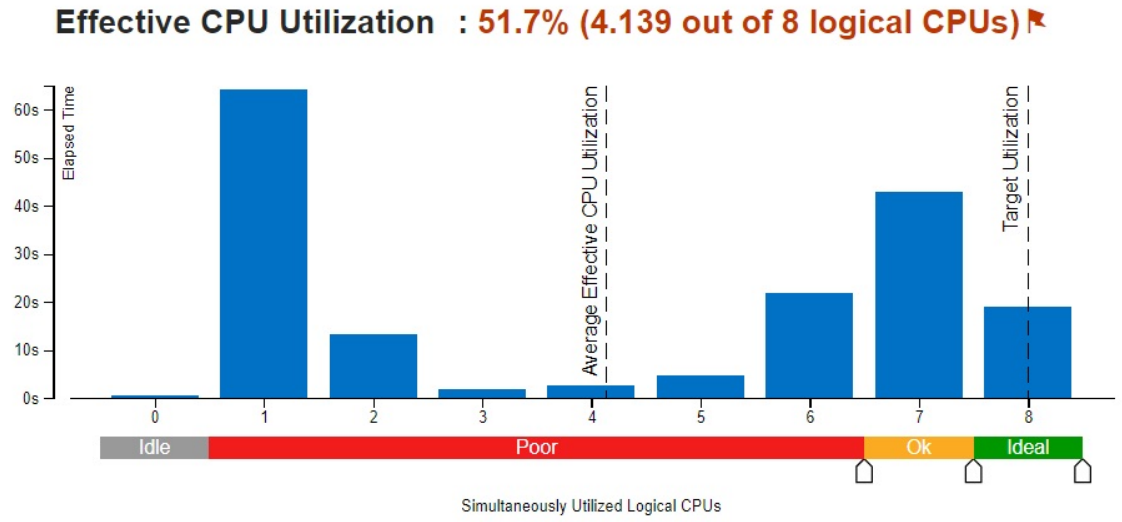

Figure 2. Threading profile for `CreateHistrogramParallelV2`

4. Experiments and Results

In this assignment experience two main experiments have been executed with the purpose to evaluate the speedups of the various versions seen above. 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, multiple iterations of the various functions are performed, in order to better evaluate the performances and avoiding to be biased by the singular specific executions. Moreover, for each study case are noted down the different behaviours that occurs when we look for bigrams or trigrams in the texts.

So, the first test that has been performed was to evaluate the speedup for bigrams varying the number of threads. For the test, as numbers of threads I've considered the values 1 and from 2 to 30 with a step size of two, in a way to have a fine-grained understanding of the performances.
The value of 1 threads is used just to verify in that case every parallel version is worse than the sequential one, because beyond using only one thread, they require also some

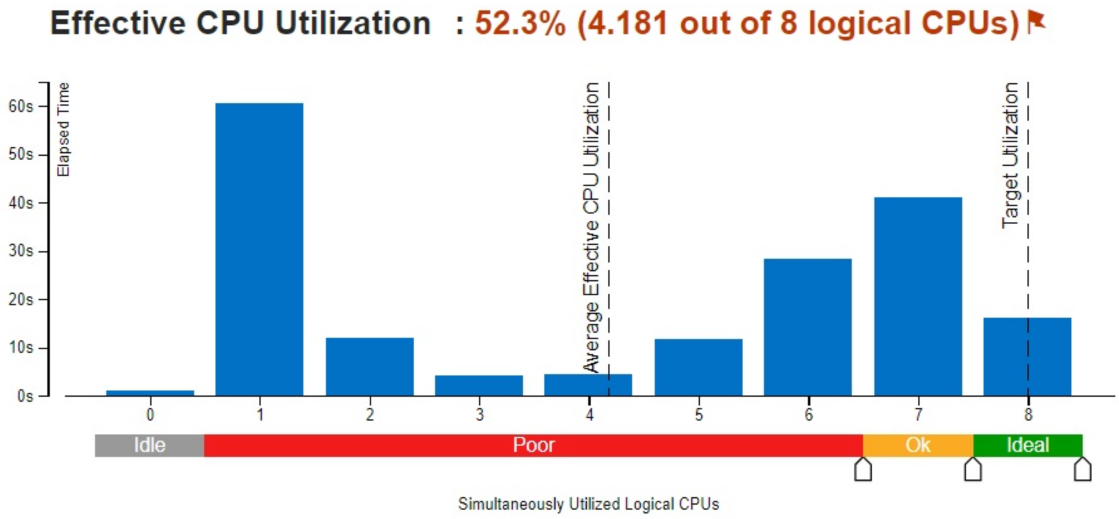

Figure 3. Threading profile for `CreateHistrogramParallelV3`

overhead for the creation and the management of the parallel section (fork-join).

For the case of bigrams I've considered 20 iterations of every function, and then, for each version the average execution times are measured, later used for computing the speedup with the respect to the sequential version.

The typical trend of the speedup in the case of bigram is shown in Figure 4. As we can see, the best performances belong to the third parallel version, but for all of them the speedup seems to remain quite stable in an interval between 1.8 and 2 without varying that much with the number of the evolved threads.

The trend correctly follows the Amdahl's law that states that the performances are limited by the sequential part of the code. So, for a fixed set of data, even if we increase the amount of threads it's not possible to reach a speedup over a certain threshold, which is related (inversely proportional) the percentage of the sequential code, which in this case is not that small because we need a strong synchronization of threads in order to create the final histograms.

Therefore, what we can experience is that there is an upper bound for the speedup around the value of 2 and increasing the number of threads we just have a loss of efficiency, since more threads don't result in more speedup.

Then, it's evaluated the speedup for the trigrams, but this time, since looking for trigrams requires more time, in this case just 10 iterations have been considered, resulting in a bit more oscillating results.

The trend for trigrams is visible in Figure 5 and, as we can see, we have the same curves shapes as in the previous test, but in this case the performances of the various versions are slightly closer to each each and they take on lower values than for the bigrams. In fact, varying the number of threads the speedup always remains in a range from 1.4 to 1.6 for each version.

The reason of this behavior is hidden in the number of word trigrams shown in Table 2, since for any possible n, the magnitude of words n-grams always overwhelms the amount of letter n-grams, and therefore, the performances are dominated by the update of that histogram.

In fact, what can been assessed is that the number of different word trigrams, and so the number of items in the histogram, is around 10 millions and about 3 times more than the bigrams. This implies that there is more time needed for updating the histograms and that there are critical sections which last more, resulting in an unavoidable decrease of the speedup.

So, what we can understand from this experiment is that increasing the size of the wanted n-grams brings to a reduction of the performance trend and this is due to strong need of synchronization which implies a sequential execution of the threads.

Overall, both for bigrams and trigrams using the parallelization does not reduce significantly the execution times, but at its best it just halves it, like in the case of bigrams.

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. For both bigrams and trigrams are considered sizes from 100 MB (one iteration of the folder) to 1 GB (ten iterations of the folder) and for evaluating the average speedup 10 executions of each parallel versions have been

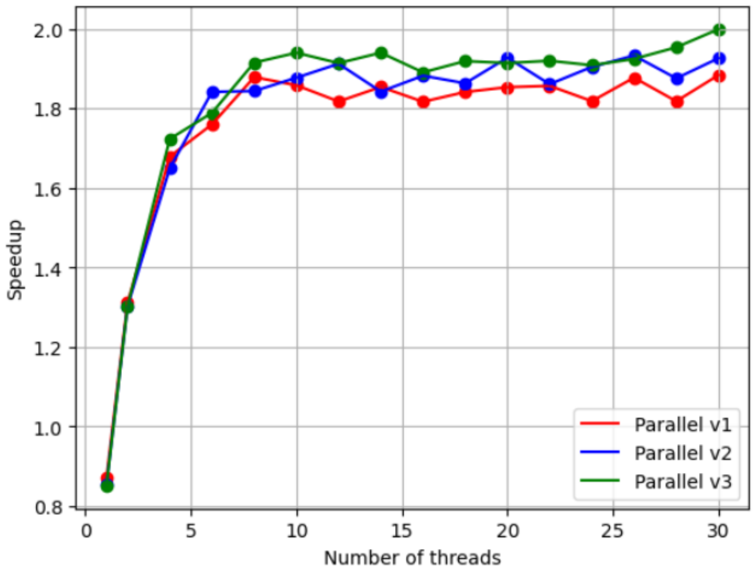

Figure 4. Speedup trend for bigrams varying the number of threads

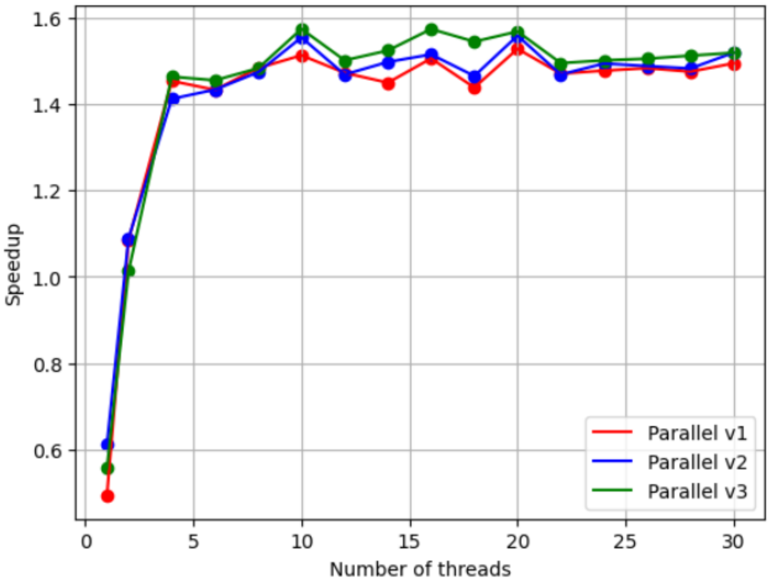

Figure 5. Speedup trend for trigrams varying the number of threads

done.

In the case of the bigrams we obtain a quite expected behaviour: the sequential execution times grow up in a linear way with respect to the size, while the execution times of the parallel versions increases, but much more slowly.
This results in a sort of linear behavior for the speedups of every parallel versions, as we can see in Figure 6.
The curve of the third parallel version is still the best one and it reaches almost a speedup of 2.9 for the maximum size, followed by the second version that reaches 2.8 and a bit more lower the fist version with a speed up around 2.7. So, in this case increasing the size of the texts dataset allows the parallel versions to outperform the sequential version of the code.

Instead, in the case of trigrams we have a bit strange behaviour for the speedup trend. In fact, as we can see from Figure 7 in this case the performances become worse and worse with the increment of the dataset size and the first

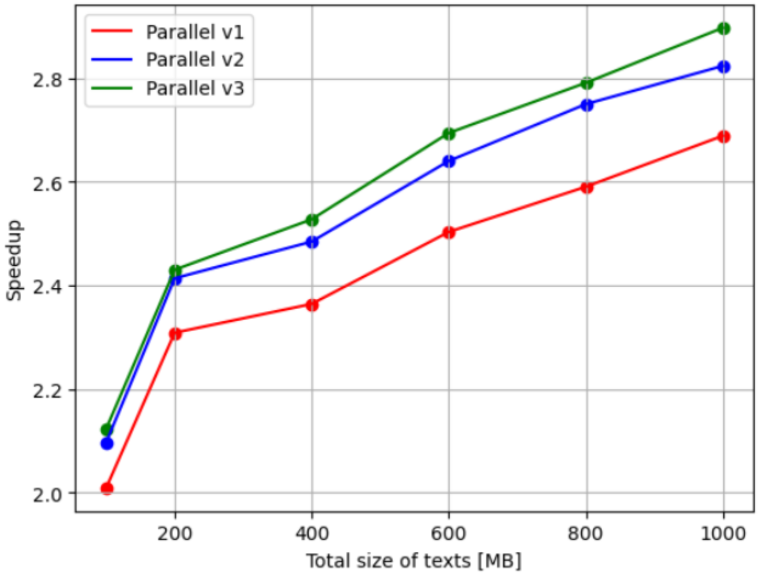

Figure 6. Speedup trend for bigrams with respect of the size of the texts set

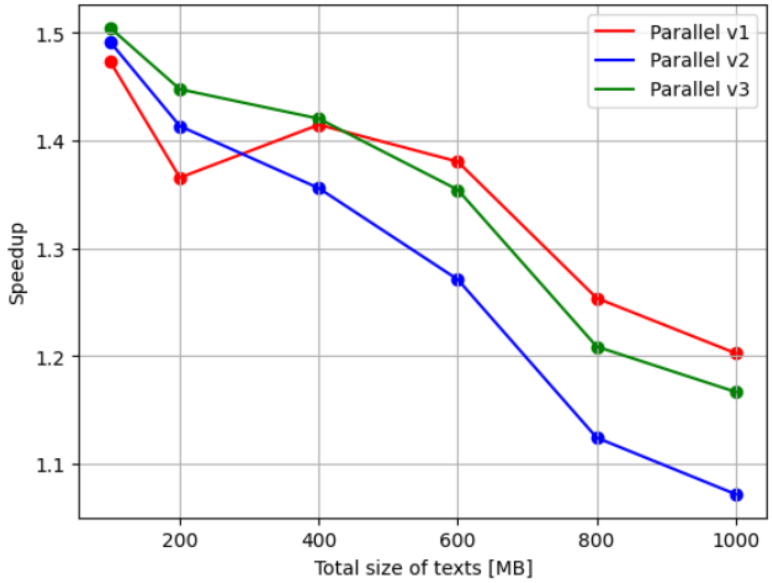

Figure 7. Speedup trend for trigrams with respect of the size of the texts set

parallel versions, which it used to be the worst one, now it seems to perform a bit better than the others when the total size overcome the 400 MB.

As it's happened in the case of varying the number of threads, the trigram trend shows lower performances than the bigram one and the reasons are probably because in the case of trigram, as already said, there are much more elements in the histograms and, moreover, when we increase the size of the texts more time for the synchronization is required, resulting in a deep lowering of the speedup.

In fact, with the increase of the size and the use of dynamic scheduling of the texts to the threads, more texts imply more elements in each local version of the histograms. This results in a greater time needed for the synchronization, because each thread keeps the semaphore of the critical section for more time.

As mentioned above, it's curious that in this case the first version performs better from a certain point onwards (around 400 MB) and the reason of that it's probably because in the first version just a `single` directive has been used and, therefore, every time the fastest thread that reaches the statement access and read the text, instead, in the other two functions it's used a `omp for` directive that deals with the distributing the loop instances to the threads and with their management. So, these two versions require more overhead for manage the various threads and that overhead increase linearly with the size of the considered texts (and that it's due to the fact that a fictitious increment of the size has been made iterating multiple times across the directory containg the texts) resulting in lower performances.

So, from this second experiment we can understand that the considered implementation is connected with the size of n-grams in a deeper way that we may expect, since varying the size of the dataset, if we consider bigrams, we have a increase of the performances, but if we consider the trigrams we have a strong worsening of the speedup values.

5. Conclusions

We have seen that it's possible using the OpenMP directives to speedup the performance of a code that compute the histogram of n-grams of letters and words (in the particular case of bigrams and trigrams) but without achieving very high values, due to the particular problem issues which needs a strong synchronizations of the threads.

The performances varying the number of threads follow the Amdahl's law and therefore we have an initial increment of the speedup and then it remains similar even for higher numbers of threads, with sightly better values of speedup for the bigrams than for the trigrams, since in the first case the histograms have less elements. Instead, when we increase the number of the texts in the case of bigrams the speedup grows up, while in the case of trigrams the performance get worse, again, due the great amount of time needed for the synchronization.

In conclusion, we can understand that for this particular problem the parallelization is not very worthy, since it's very difficult reach high values of speedup, or at least we can have decent speedup only in the case of bigrams and considering a great amount of texts.