OpenMP implementation of an n-grams histogram extractor

Davide Pucci
Matricola: 7064811
davide.pucci@stud.unifi.it

Abstract

This work is a mid term project for the course of Parallel Computing, held by professor Marco Bertini at University of Florence. The aim is to compare the computational costs of a sequential and a parallel version of an n-gram histogram extractor on words, evaluating the speedup achieved in several test situations. The implementation of the parallel version of the algorithm makes use of OpenMP API.

1. Introduction

In computational linguistics an *n-gram* is a sequence of n contiguous items from a sample of text. We consider **words** n-grams in this work, as these are widely used in statistical natural language processing [4], information retrieval [5] and many other fields. In particular we want to produce an histogram of n-grams occurrences over large texts.

An example word n-gram of the well known line *"to be or not to be"* by Shakespeare is reported in table 1.

n	***n-grams***
1	to, be, or, not, to, be
2	to be, be or, or not, not to, to be
3	to be or, be or not, or not to, not to be
4	to be or not, be or not to, or not to be
5	to be or not to, be or not to be
6	to be or not to be

Table 1. Word n-grams of the line *"to be or not to be"*.

Our aim is to count the occurrences of the n-grams in large corpus of text. Such operation is quite time consuming, as we need to manage the input from file, to generate the n-grams, to count the occurrences and finally to produce the output. In this work we consider books available in Project Gutenberg [2] and large text corpus from Statistical Machine Translation workshop of 2015 [1] in order to test our implementation over input files in `.txt` format. In this work we compare the computational costs of a sequential *n-gram* histogram extractor and a parallel version of the same extractor, implemented with OpenMP API [3] for shared memory parallel programming in C++.

2. The Data

The text corpus used in this work comes from Project Gutenberg, a virtual library consisting of over 60,000 free eBooks, and from the Statistical Machine Translation workshop of 2015, in which is available a collection of news articles from 2007 to 2014. Five books were downloaded from the first in `.txt` format, and two collections (2010 and 2011) from the second. These large corpus were used in order to test the performances of the two implementations.

```
The Project Gutenberg EBook of Within a Budding Grove, by Marcel Proust

This eBook is for the use of anyone anywhere in the United States and most
other parts of the world at no cost and with almost no restrictions
whatsoever.  You may copy it, give it away or re-use it under the terms of
the Project Gutenberg License included with this eBook or online at
www.gutenberg.org.  If you are not located in the United States, you'll have
to check the laws of the country where you are located before using this ebook.

Title: Within a Budding Grove

Author: Marcel Proust
```

Figure 1. First lines of an input file from Project Gutenberg.

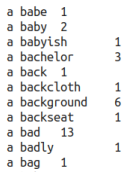

Figure 2. First lines of the output produced when $n = 2$.

As our focus is on words, punctuation was ignored, as well as numbers. Moreover uppercase characters were lowered, in order to offer a uniform, lowercase output. The output is produced in a single `.txt` file. An example of both input and output files is given in figure 1 and figure 2.

3. The Implementation

The algorithm for words n-grams extraction is made up of 3 steps:

1. Input from file
2. *n-gram* generation and count
3. Output to file

The first step consists of reading line by line the input file and memorizing the distinct words in a data structure which can handle them, so that the second step can be performed quickly. The data structure used in the implementation is the `std::vector`, as the complexity for both insertion at the end and random access is $O(1)$. Each word stored in the vector is a `std::string`. During the first step we perform as well punctuation removal, number removal and uppercase characters are converted to their lowercase version.

The second step consists in the concatenation of consecutive words in order to produce the *n-gram*, which is then added to a `std::map` in order to count the occurrences. When an n-gram is generated, it is straight forward to generate the following one: we remove the first word in the concatenated string and append the following word at the end. In such a way we can efficiently produce all the n-grams for a text corpus. Using the `std::map` allows us to quickly count the n-grams: when a new n-gram is generated we check if it is a key to the map, if so we add one to the associated value, which is used as a counter. On the other hand, when the n-gram is not a key for the map, we generate the new `std::pair` in which the n-gram is a key and the value is a counter, initialized to one.

The last step consists in writing the `std::map` to the output file. As the C++ implementation of the map is based on red-black trees, the output will be sorted by key. An example of the output file is reported in figure 2.

In order to measure the execution time of both the sequential and the parallel implementation, the function `omp_get_wtime()` from `omp.h` was used, comparing the wall-clock time at the beginning of the computation with the one at the end. This allowed to compute the speedup obtained in distinct test scenarios.

3.1. Sequential Implementation

The sequential implementation of the n-grams extractor consists in the 3 steps of the algorithm introduced before in this section. Using `std::ifstream` and `std::getline` we are able to gather each word in the text, scanning the file line by line. Using `std::remove_copy_if` we are able to perform punctuation removal, number removal and to lower any uppercase character, just by defining a custom processing function called by `remove_copy_if`.

At the end of the input phase, all words in the text are stored into a `std::vector`, then, using concatenation as described before, the n-grams are generated. Once each n-gram is produced we make use of `std::map` to count the occurrences of the n-grams as described before.

Once all the words in input are processed the `std::map` is written to the output file using `std::ofstream`.

3.2. Parallel Implementation

The parallel version of the n-grams extractor makes use of OpenMP API [3] to implement a Producer-Consumers pattern. The Producer thread reads the input file in the same way it is done in the sequential implementation but, when

a chunk of words of size `chunk_size` has been read, it enqueues the current `std::vector` of words onto a `JobsQueue`. The Producer keeps on reading and producing chunks of data, until the end of the input file is reached. Consumer threads wait for chunks to be pushed in the queue. Once a chunk is available, a Consumer extracts it from the data structure, produces the n-grams and updates its local n-gram counter (stored as a private `std::map`), and, when all words in the chunk have been processed, the Consumer moves to the next chunk. Once a Consumer finishes with its work, which means that no chunks are available in the `JobsQueue` and that the Producer has finished working, it needs to merge its local histogram into a `SharedHistogram`. Once all Consumers have merged their local histogram into the shared one, this `std::map` can be written to the output file.

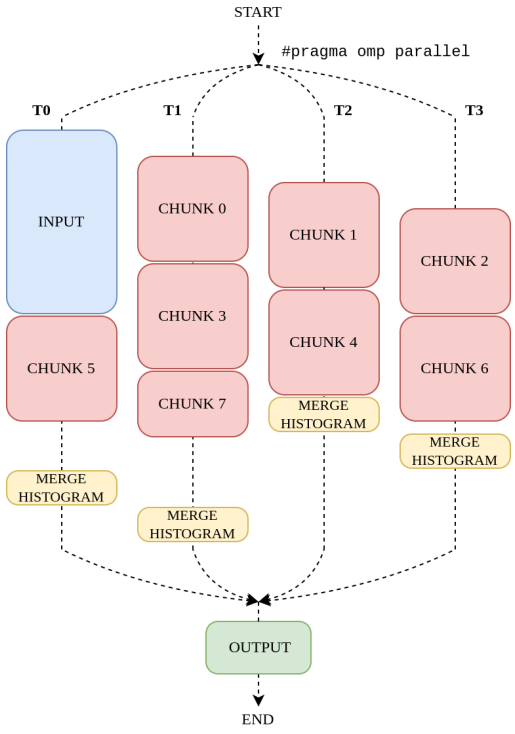

Figure 3. Organization of threads' workflow in this implementation. In blue we have the input phase executed by the Producer, in red is the n-grams generation and count performed over every chunk of data by Consumers, in yellow is the histogram merging phase into the shared one, finally in green is the output phase. Note that the last chunk of data might be smaller than the `chunk_size` and that the merging phase is performed by one thread a the time in order to avoid race conditions.

In order to implement this logic in OpenMP we make use of `#pragma omp parallel` directive to generate the threads that will work in parallel. Inside the `parallel` block `#pragma omp single nowait` is used to define the behavior of the Producer thread, whilst the Consumer threads are defined outside of the `single` block, but still inside the `parallel`. In such a way the Producer thread will become a Consumer once it has completed its task. Figure 3 reports the resulting organization of threads in this implementation.

It is important to highlight that data chunks must have an overlap of $n - 1$ words in order to generate all the n-grams in the text corpus. If we consider for example the line *"to be or not to be"* when $n = 2$; a data chunk that contains every word until *"or"* will generate as last n-gram *"be or"*. The following n-gram to be generated has to be *"or not"*, thus the next data chunk needs to include *"or"* as well. Figure 4 sums up this example.

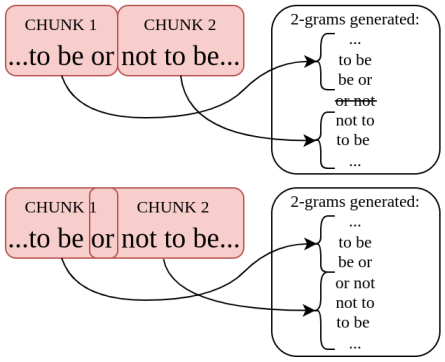

Figure 4. Word bi-grams generated with and without overlap between chunks.

The `JobsQueue` class is implemented making use of OpenMP directives and lockers, as the access to the queue needs to be thread safe. In particular the queue allows push operation at the back for the Producer and pop operation from the front for Consumers. The pop operation is secured with the usage of a locker `omp_lock_t` as otherwise we would run into a race condition, as multiple Consumers might access to the same chunk. The push operation is secured in the same way with another locker, as this implemen-

tation could work with multiple Producer threads as well. The `JobsQueue` also stores the state of the Producer thread: once the input file has been read and the last chunk of data is pushed onto the queue, the Producer needs to notify Consumers that no other chunks of words will be added. This is done tracking the number of active Producers in an attribute of the class: when a Producer finishes its work the attribute is decremented. In a scenario of multiple Producers this decrement needs to be thread safe. This can be done through `#pragma omp atomic` directive.

The queue also keeps track of the number of chunks that are enqueued and dequeued. We have two separate counters in order not to have a race condition between Producer and Consumers on the access to a single counter variable. This allows pop and push operations to be completely independent. A Consumer thread can check if its work is done looking at the number of active Producers and at the number of chunks that were enqueued and dequeued: if no Producers are active and the number of enqueued is equal to the number of dequeued the Consumer thread has completed his job. As we want to avoid reading from a not updated memory in these cases, the directive `#pragma omp flush` is used.

The `SharedHistogram` class is implemented making use of OpenMP lockers, as we need Consumers to access it without race conditions. This class owns a private `std::map` which contains the histogram that collects all the private counters of each Consumer. The class method `mergeHistogram()` allows consumer Threads to add to the shared histogram their local counter. This operation is secured with a locker, as we want to avoid race conditions.

Finally, once all Consumers have successfully merged into `SharedHistogram` their counters, the shared map is written to the output file, making use of `std::ofstream`. As the shared histogram is implemented through the usage of a `std::map`, the output will be sorted by key.

Other solutions can be proposed for this reduction step. One idea could be to gather all the partial histograms, and to join them only when producing the output. This solution is less efficient compared to the usage of a `SharedHistogram`, as the join operation is performed only once all Consumers have finished their work. Another idea could be to build a binary reducer: join two local histograms at the time, in such a way we can perform multiple of these operations in parallel, simply assigning to distinct threads the task of merging two histograms. This second solution allows a certain degree of parallelism, but at the price of a larger overhead and a larger memory usage. The tests conducted show how the solution given by the class `SharedHistogram` results the best in terms of speedup.

4. Experimental Results

In order to compare the parallel implementation of the n-grams histogram extractor with the sequential one, we measure their execution time. In such a way we're able to estimate the **speedup** achieved by the parallel implementation in several test situations. The speedup si computed as the ratio between the execution times of the sequential implementation and the parallel one.

In this analysis we evaluate the speedup at the vary of the number of threads, at the vary of the length of the input document (in terms of amount of words to be processed), at the vary of the `chunk_size` and at the vary of n. This allows us to identify the situations in which the parallel implementation is better performing than the sequential one, and, on the other hand, when the opposite behavior can happen, due to overhead for example. Moreover, in order to show the performances in a real-world scenario, we report the execution times of both sequential and parallel implementations when processing well single or collections of books.

The experiments were executed on a **Intel Core i7-8565U CPU**, which has 4 physical cores and 8 threads, running Ubuntu 18.04 LTS operating system. In order to avoid fluctuations in the results, multiple runs were executed for each test.

As aforementioned `omp_get_wtime()` from `omp.h` was used to measure the execution times.

4.1. Analysis at the vary of number of threads

Here is reported the speedup measured at the vary of the number of threads available for OpenMP. In order to set the maximum number of threads to be used, we make use of `num_threads(MAX)` clause in the directive `#pragma omp parallel`.

In this experiment we consider bi-grams (n=2) for an input file of 100 million words. The `chunk_size` is set to 10k. Results obtained are reported in Figure 5.

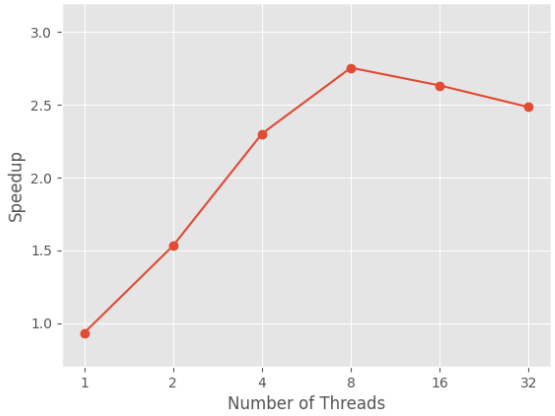

Figure 5. Speedup at the vary of the number of threads available for OpenMP. In this experiment we consider n=2, chunk_size=10k and 100 million words to be processed.

We can see how increasing number of threads allows to use all the computational resources available in this hardware, achieving a speedup of about 2.8 when 8 threads are used. With a larger amount of threads performances get worse due to increasing overhead. Below 8 threads we are not making use of all the potentialities of the hardware, resulting in smaller speedups. When only 1 thread is available the sequential implementation results faster, as we're using the parallel version in a sequential way.

4.2. Analysis at the vary of the input size

Here is reported the speedup measured varying the length of the input document, in terms of amount of words to be processed. In this experiment we consider bi-grams (n=2), with 8 threads available for OpenMP and chunk_size=10k. Results are reported in figure 6.

We can see how the document's length influences the speedup as we would expect from Gustafson's law. With small input documents the sequential implementation results the quickest, as the histogram merging phase, which is almost sequential, results the most time-consuming task of the parallel implementation. When the size of the input document increases, the hot-spot of the parallel program becomes the n-gram generation and the subsequent count, which can be executed totally independently by each thread, as a result we can achieve good performances in terms of speedup.

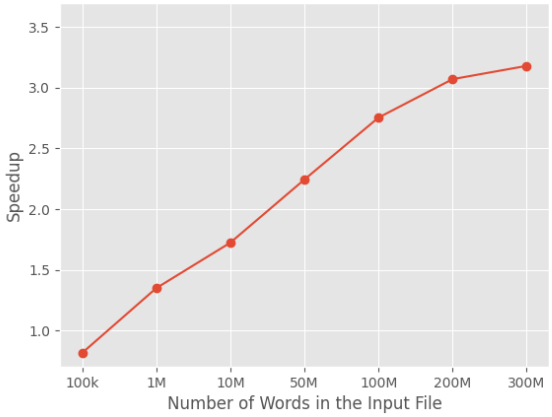

Figure 6. Speedup at the vary of the input document's length. In this experiment we consider n=2, n_threads=8 and chunk_size=10k.

With large documents both sequential and parallel implementations make use of a large amount of RAM in order to store the histograms. As a result it was not possible to measure the speedup with input documents containing more than 300 million words.

The speedup values measured in this subsection are related to chunk_size, as reported in the following subsection.

4.3. Analysis at the vary of the chunk_size

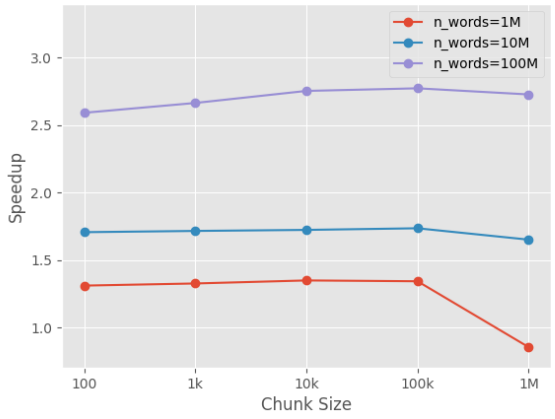

Figure 7. Speedup at the vary of the chunk_size. In this experiment we consider n=2, n_threads=8 and input of 1, 10 and 100 million words.

Here is reported the speedup measured at the vary of the `chunk_size`. In this experiment we consider bi-grams (n=2), with 8 threads for OpenMP and input documents of 1, 10 and 100 million words. Results are reported in figure 7.

The results obtained show how the document's length influences the speedup, in relation to the chunk_size. With smaller chunk sizes the Consumer threads start working before, as a result we can achieve good speedups as long as the input document is not too long: with large documents many chunks are created, thus the overhead increases. On the other hand with a large chunk size Consumer threads start working later, but there will be less overhead.

The more suitable chunk size is a trade off between the need to quickly start the computation and the need to create enough chunks to keep all the units working, still avoiding small chunks, as this causes too many accesses to the `JobsQueue`, resulting in a bottleneck.

When the `chunk_size` coincides with the size of the input document the speedup is below 1 as just one chunk of data is created. In this scenario only 1 Consumer will be effectively working, resulting in a sequential-like execution.

As we can see from figure 7 a chunk_size of 10-100k is suitable for every situation.

4.4. Analysis at the vary of n

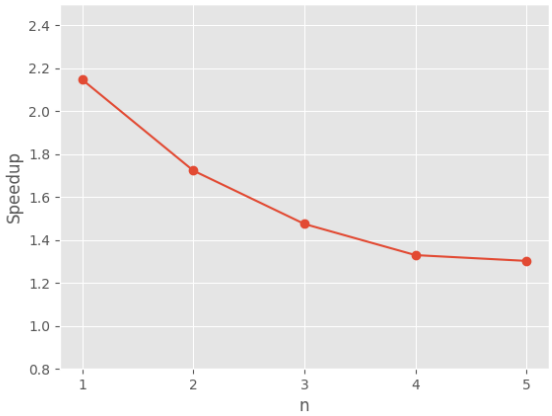

Figure 8. Speedup at the vary of n. In this experiment we consider n_threads=8, chunk_size=10k and input of 10 million words.

Here is reported the speedup measured at the vary of n. In this experiment we consider from uni-grams up to 5-grams when 8 threads are available for OpenMP, the input document contains 10 million words and the chunk_size is 10k. Results are available in figure 8.

With increasing values of n the speedup gets lower. This is caused by the larger histograms that are generated when n increases, as a result the merging phase required in the parallel implementation will cause a delay in the production of the output.

Moreover, as highlighted in figure 4, when n is larger chunks become more overlapped. As a result the computation of the n-grams will be partially repeated in each thread. Still, as in this experiment we consider chunks of 10k words and n below 5, this phenomena is less relevant.

4.5. Execution Time on Books

Finally, here is reported an analysis of the execution times in a real-world scenario, of both sequential and parallel implementations. We consider bi-grams extraction from 3 books available in Project Gutenberg and from 1 collection of news articles of 2010 available in SMT workshop website.

For the parallel implementation we consider n_threads=8 and chunk_size=10k. The results, av-

	Approx. Number of Words	Sequential Runtime	Parallel Runtime	Speedup
Roman Stoicism (E. Vernon Arnold)	196k	$0.4043s$	$0.3414s$	1.1851
Within a Budding Grove (Marcel Proust)	239k	$0.4687s$	$0.3739s$	1.2537
The Bible (King James version)	824k	$0.9870s$	$0.7030s$	1.4043
News Crawl (Articles from 2010)	136M	$305.23s$	$107.03s$	2.8517

Table 2. Execution time of the two implementations over books from Project Gutenberg. In the parallel implementation we consider n_threads=8 and chunk_size=10k.

eraged over multiple executions, are reported in table 2. We can see how we can achieve a good speedup for every text corpus considered. Possibly using a larger chunk_size would allow better performances over large text corpus such as the news collection.

5. Conclusions and Further Developments

In this work is presented a detailed comparison between a sequential implementation of a word n-grams histogram extractor and its parallel equivalent. The results obtained prove the efficiency of the parallel version in real-world scenarios.

We have seen the details of both implementations, considering as well distinct solutions for the parallel implementation. We have tested the most diverse situations, in order to understand the properties of both implementations. We've also considered some corner-cases in which the parallel implementation doesn't work properly, such as when chunks are too large to allow an equal distribution of the workload or when the input document is too small to take advantage of parallelism.

Thanks to OpenMP this implementation can easily run on every hardware, taking advantage of all the computing units offered by the CPU, and, taking care not to end-up in a pathological case, can always achieve good performances in terms of speedup.

The implementation could be easily adapted to make use of multiple Producer threads: in a scenario in which two distinct files need to be processed, we could think to assign the input of each file to a distinct Producer, while keeping the rest as it is.

References

[1] E. 2015. Tenth workshop on statistical machine translation, september 2015.

[2] P. Gutenberg. https://www.gutenberg.org/.

[3] OpenMP Architecture Review Board. OpenMP application program interface version 5.0, May 2019.

[4] G. Sidorov, F. Velasquez, E. Stamatatos, A. Gelbukh, and L. Chanona-Hernández. Syntactic n-grams as machine learning features for natural language processing. *Expert Systems with Applications*, 41(3):853–860, 2014.

[5] X. Wang, A. McCallum, and X. Wei. Topical n-grams: Phrase and topic discovery, with an application to information retrieval. In *Seventh IEEE International Conference on Data Mining (ICDM 2007)*, pages 697–702, 2007.