# OpenMP implementation of an n-grams 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 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.*

## Future Distribution Permission

## 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 [3], information retrieval [4] and many other fields.

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 |

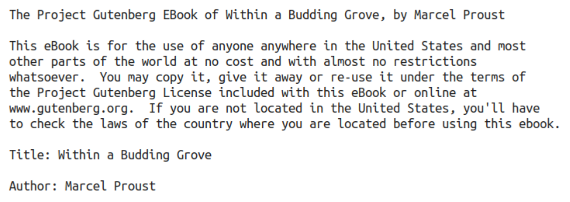
Table 1. Word n-grams of the line *"to be or not to be"*.

The task of extracting n-grams from a large corpus of text is quite time consuming, as we have to manage the input from file, to process the words and to produce the output. In this work we consider books available in Project Gutenberg [1] in order to test our implementation over input files in `.txt` format. Our aim is to compare the computational costs of a sequential *n-gram* extractor to a parallel version of the same extractor, implemented with OpenMP API [2] for shared memory parallel programming in C++.

## 2. The Data

The text corpus used in this work comes form Project Gutenberg, a virtual library consisting of over 60,000 free eBooks. Five books were downloaded in `.txt` format in order to test the performances of the two implementations.

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.



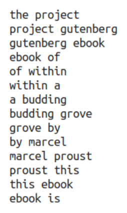
```
The Project Gutenberg EBook of Within a Budding Grove, by Marcel Proust

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.



```
the project
project gutenberg
gutenberg ebook
ebook of
of within
within a
a budding
budding grove
grove by
by marcel
marcel proust
proust this
this ebook
ebook is
```

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

## 3. The Implementation

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

1. Input from file
2. *n-gram* generation
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*. Once an n-gram is generated it can be written to the output file. After that, it is straight forward to generate the following n-gram: 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.

The second step and the third one are coupled together in this implementation, in order to improve efficiency.

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 of 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 available it is directly 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 [2] 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, processes the words contained and writes the produced n-grams to a private output file. Once all the words in the input file are processed, the output files produced by each Consumer are merged into a single one.

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

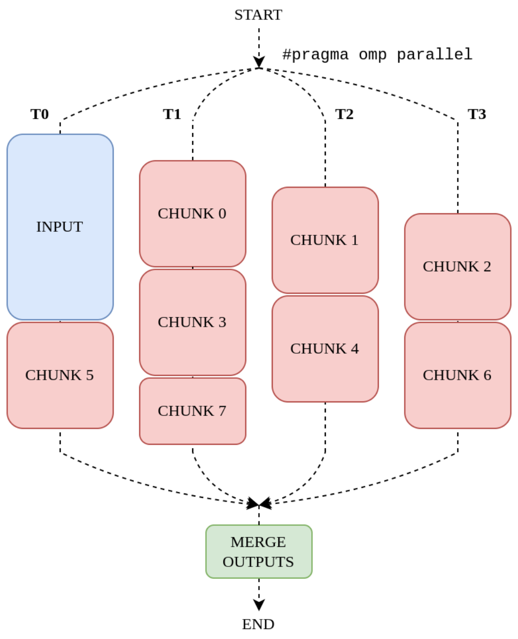


Figure 3. Organization of threads' workflow in this implementation. In blue we have the input phase executed by the Producer, in red is the n-grams generation and output performed over every chunk of data by Consumers, in green is the final merging phase. Note that the last chunk of data might be smaller than the `chunk_size`.

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.

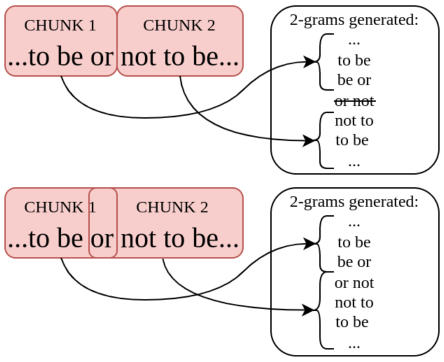


Figure 4. Word bi-grams generated with and without overlap between chunks.

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 implementation 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 logic used to merge all the partial output files relies on `std::filesystem` to iterate on the temporary directory where the partial files are stored. Then, using `std::ifstream` and `std::ofstream`, the content of each file is copied onto a single output file. Finally `std::filesystem` is used once again to remove all the partial files. The filesystem library is available since C++17, hence we need to set up the `CMakeLists.txt` file and the compiler accordingly.

## 4. Experimental Results

In order to compare the parallel implementation of the n-grams 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.

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 1 million words. The `chunk_size` is set to 10k. Results obtained are reported in Figure 5.

We can see how increasing number of threads allows to use all the computational resources available in this hardware, achieving a speedup of about 3.7 when 8 threads are used. With a larger amount of threads the performances are similar, with a slight drop 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.

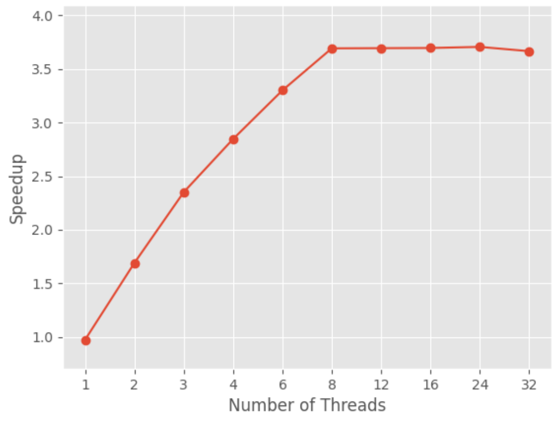


Figure 5. Speedup at the vary of the number of threads available for OpenMP. In this experiment we consider n=2, chunk_size=10k and 1 million words to be processed.

### 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_sizes of 1k, 10k and 100k. Results are reported in figure 6.

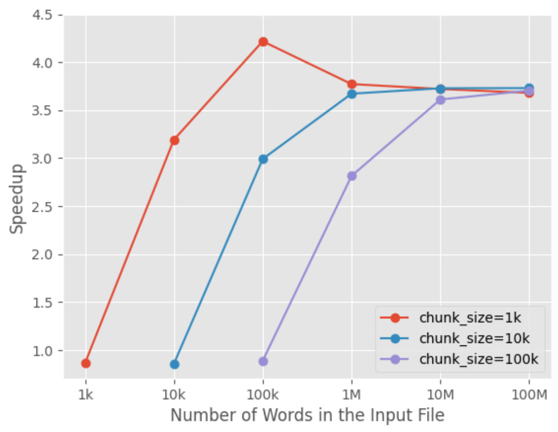


Figure 6. Speedup at the vary of the input document's length. In this experiment we consider n=2, n_threads=8 and chunk_size of 1k, 10k and 100k.

We can see 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.

A complementary experiment is reported in the following subsection.

### 4.3. Analysis at the vary of the chunk_size

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 10k, 100k and 1 million words. Results are reported in figure 7.

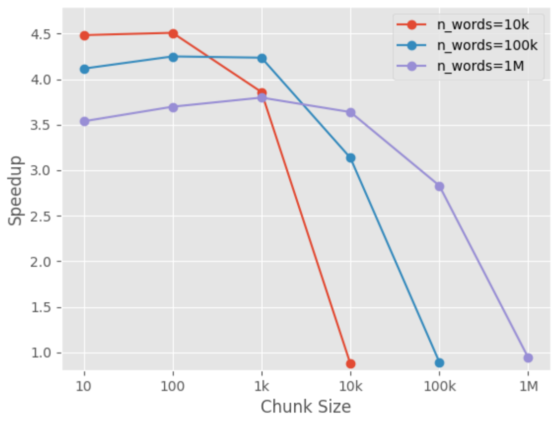


Figure 7. Speedup at the vary of the chunk_size. In this experiment we consider n=2, n_threads=8 and input of 10k, 100k and 1 million words.

The results obtained in this experiment remark the same points highlighted in the previous subsection: when documents are smaller the best choice is to use small chunk_sizes in order to start the computation of the n-grams as soon as possible. With larger documents instead we need to enlarge the chunk_size in order to have the best possible trade-off between size and number of chunks.

We notice as well that when the number of words in the input document is below 100k we can achieve a speedup grater than 4: over the number of CPU's physical cores. This is achieved thanks to Hyper-Threading, which allows quick context-switches between threads. As Consumer tasks include mostly output to files, a lot of time is spent into wait state, thus, thanks to fast context-switches, other threads can be executed in the meantime, saving time.

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.

### 4.4. Analysis at the vary of n

Here is reported the speedup measured at the vary of $n$. In this experiment we consider from bi-grams up to 6-grams when 8 threads are available for OpenMP, the input document contains 1 million words and the chunk_size is 10k. Results are available in figure 8.

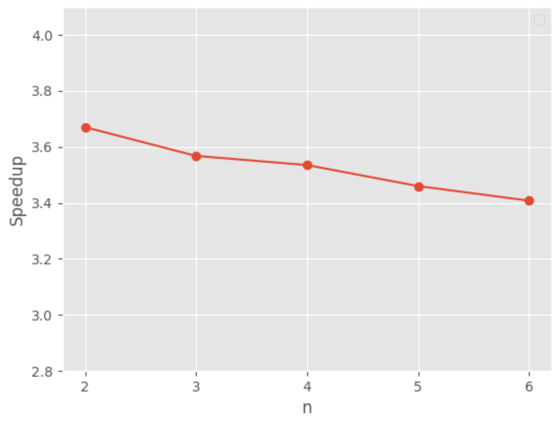


Figure 8. Speedup at the vary of n. In this experiment we consider n_threads=8, chunk_size=10k and input 1 million words.

|  | **Approx. Number of Words** | **Sequential Runtime** | **Parallel Runtime** | **Speedup** |
|---|---|---|---|---|
| A Study In Scarlet (Arthur Conan Doyle) | 46k | $0.1241s$ | $0.0301s$ | 4.170 |
| The Path of Honor (Burton Egbert Stevenson) | 77k | $0.2015s$ | $0.0496s$ | 4.098 |
| Roman Stoicism (E. Vernon Arnold) | 196k | $0.4675s$ | $0.1148s$ | 4.104 |
| Within a Budding Grove (Marcel Proust) | 239k | $0.6024s$ | $0.1583s$ | 3.862 |
| The Bible (King James version) | 824k | $1.9972s$ | $0.5449s$ | 3.667 |

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=1000.

With increasing values of $n$ the speedup gets lower. This might be caused by the final merging phase which joins all the partial output files generated by each thread: when $n$ is larger the n-grams will occupy more space, as each word is repeated $n$ times; as a result the merging phase will have more impact in the overall execution time.

Moreover, as highlighted in figure 4, when $n$ is larger chunks become more overlapped. As a result the computation will be partially repeated in each thread. Still, as in this experiment we consider chunks of 10k words and n below 6, this phenomena is negligible.

### 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 of 5 books from Project Gutenberg.

For the parallel implementation we consider n_threads=8 and chunk_size=1000. The results, averaged over 50 executions, are reported in table 2. We can see how we can achieve a good speedup for every book considered. Possibly using a larger chunk_size could allow better performances over large books such as The Bible.

## 5. Conclusions and Further Developments

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

We have seen the close correlation between `chunk_size` and input size, showing that a tuning of this parameter allows better performances. 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.

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 due to a wrong choice of `chunk_size`, 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] P. Gutenberg. https://www.gutenberg.org/.

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

[3] 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.

[4] 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.