Comparative Analysis of Sequential and Parallel Algorithms for N-gram Extraction using C++ and Python

Dylan Fouepe
E-mail address
dylan.fouepe@edu.unifi.it

Abstract

This project examines the performance differences between sequential and parallel algorithms for n-gram extraction and counting. Implementations are done in C++ using the OpenMP library and in python using the asyncio concurrent library. By processing a corpus of text, the study highlights the efficiency and scalability of parallel computing techniques compared to their sequential counterparts.

1. Introduction

The rapid growth of data has necessitated the development of efficient algorithms for text processing. One common task in natural language processing is the extraction and counting of n-grams, which are contiguous sequences of n items from a given text.Traditional sequential algorithms, while straightforward, often fail to scale effectively with large datasets. This project investigates the potential of parallel computing to enhance the performance of n-gram extraction algorithms. By leveraging the OpenMP library in C++ ad the asyncio library in python, we aim to compare the execution times and efficiency of sequential versus parallel implementations. This comparison will provide insights into the benefits and challenges of parallelizing text processing tasks.

1.1. Data

The data used in this study originates from books downloaded in text (txt) format from Project Gutenberg, a large repository of free and accessible e-books. To evaluate the performance of our program, we generated larger corpora by repeating and combining these texts. This approach allows us to simulate larger datasets and observe performance changes as the data size increases. Our focus is on words, so we ignored punctuation and numbers. Additionally, all uppercase characters were converted to lowercase to ensure a uniform output. The final output is generated in a single .txt file.

1.2. N-grams

N-grams are contiguous sequences of n items from a given text, where the items can be characters, words, or other linguistic units. They are widely used in natural language processing (NLP) and text mining tasks for capturing patterns and relationships within textual data. N-grams of different lengths (unigrams, bigrams, trigrams, etc.) provide insights into language structure, syntax, and semantics. For example, bigrams (two-word sequences) can help identify commonly occuring phrases or expressions, while trigrams (three-word sequences) can capture more complex linguistic patterns. In this project, the focus is on extracting and counting n-grams from a corpus using both sequential and parallel algorithms. Efficient n-gram extraction is crucial for tasks such as language modeling, sentiment analysis, and information retrieval, making it essential to eval-

uate the performance of different computational approaches, including their scalability and speed in handling large-scale datasets.

1.3. Optimizing N-gram Extraction: Harnessing Parallel Computing in C++ with OpenMP

In this project C++ with OpenMP libraries are employed to explore efficient algorithms for n-gram extraction from textual data. C++ is chosen for its high-performance capabilities and robustness in handling intensive computations. OpenMP, a widely used API for parallel programming in C++, facilitates the concurrent execution of tasks accross multiple threads, making it suitable for optimizing the performance of algorithms on multi-core processors. The parallelization offered by OpenMP allows tasks to be divided into smaller units that can be executed simultaneously, thereby enhancing overall computational efficiency. The choice allows for a comprehensive comparison of sequential versus parallel approaches in n-gram extraction. This comparison not only evaluates the performance gains achieved through parallel computing but also assesses factors such as scalability, resource utilization, and ease of implementation. Ultimately, the project aims to provide insights into the optimal strategies for leveraging parallelism in text processing tasks, thereby contributing to advancements in natural language processing and computational linguistics.

1.4. The Implementation

We implemented a general algorithm to generate and count n-grams from a corpus. The algorithm begins by tokenizing the text, splitting it into individual words while ignoring punctuation and numbers. All words are converted to lowercase to ensure consistency. Next, it generates n-grams, which are contiguous sequences of n words, from the tokenized text. The algorithm then counts the occurrences of each unique n-gram within the corpus. This implementation is designed to handle large datasets efficiently, with both sequential and parallel processing versions to optimize performance based on the size of the input data. For the next two sections 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

1.4.1 Sequential Implementation

The sequential implementation of the n-grams extractor follows the three steps outlined earlier. We use `std::ifstream` and `std::getline` to read the text file line by line, extracting each word. Punctuation and numbers are removed, and uppercase characters are converted to lowercase using `std::remove_copy_if` with a custom processing function. After processing, all words are stored in a `std::vector`. The n-grams are then generated through concatenation as previously described. Each n-gram's occurrences are counted using a `std::map`. Once all words have been processed, the `std::map` is written to an output file using `std::ofstream`.

1.4.2 Parallel Implementation

The parallel version of the n-grams extractor makes use of the OpenMP API [1] to implement a Producer-Consumers pattern. The Producer thread reads the input file in the same way as in the sequential implementation. However, 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 continues reading and producing chunks of data until the end of the input file is reached. Consumer threads wait for chunks to be pushed into 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`). When all words in the chunk have been processed, the Consumer moves to the next chunk. Once a Consumer finishes its work, meaning that no chunks

are available in the `JobsQueue` and the Producer has finished working, it merges its local histogram into a `HistogramCollector`. In order to implement this logic in OpenMP, we use the `#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, while the Consumer threads are defined outside of the single block, but still inside the parallel block. This way, the Producer thread will become a Consumer once it has completed its task.

The `JobsQueue` class is implemented using OpenMP directives and locks to ensure thread-safe access to the queue. Specifically, the queue allows the Producer to push operations at the back and Consumers to pop operations from the front. The pop operation is secured with a lock (`omp_lock_t`) to prevent race conditions, as multiple Consumers might try to access the same chunk. Similarly, the push operation is secured with another lock, as this implementation could work with multiple Producer threads as well.

The `JobsQueue` also tracks 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 more chunks will be added. This is managed by tracking the number of active Producers in an attribute of the class. When a Producer finishes its work, this attribute is decremented. In a scenario with multiple Producers, this decrement needs to be thread-safe, which can be achieved using the `#pragma omp atomic` directive.

A Consumer thread can determine if its work is complete by checking the number of active Producers and the number of chunks enqueued and dequeued. If no Producers are active and the number of enqueued chunks equals the number of dequeued chunks, the Consumer thread has finished its job. To avoid reading from outdated memory in these cases, the `#pragma omp flush` directive is used.

Finally, for the reduction step, the idea is to gather all the partial histograms and join them only when producing the output.

1.5. Benchmarks Results

To compare the parallel implementation of the n-grams histogram extractor with the sequential version, we measured their execution times. This allows us to estimate the speedup achieved by the parallel implementation in various test scenarios. The speedup is computed as the ratio between the execution times of the sequential implementation and the parallel implementation.

In this analysis, we evaluate the speedup by varying the number of threads, the length of the input document (in terms of the number of words to be processed), the chunk size, and the value of n. This helps us identify situations where the parallel implementation outperforms the sequential one and, conversely, where the sequential implementation might be more efficient due to overheads.

To demonstrate performance in real-world scenarios, we report the execution times for both the sequential and parallel implementations when processing single books or collections of books. The experiments were conducted on an Intel Core i7-1165G7 CPU, which has 4 physical cores and 8 threads, running the Ubuntu 20.04 operating system with the gcc compiler on CLion.

To ensure reliable results, each test was executed five times, and the average execution time was recorded. We used `high_resolution_clock` in the `measure_time` function to measure the execution times.

1.5.1 Benchmark number of threads

In this experiment, we analyze bi-grams ($n = 2$) from an input file containing 10 million words. The chunk size is set to 1000 words. The obtained results are reported.

1. NumThreads, Speedup
2. 1 0.978602
3. 2 1.63143

4. 4 2.1129
5. 8 1.42295
6. 16 1.07434

We observe that increasing the number of threads maximizes the utilization of computational resources available on this hardware, achieving a speedup of approximately 2.1129 with 4 threads. However, performance deteriorates with more than 4 threads due to increased overhead. With fewer than 4 threads, the hardware's full potential is not utilized, resulting in smaller speedups. When only 1 thread is used, the sequential implementation is faster because the parallel version is executed sequentially.

1.5.2 Benchmark chunk size

The speedup measured with varying chunk sizes is reported here. In this experiment, we consider bi-grams ($n = 2$) using 4 threads with OpenMP, and input documents containing 1 and 10 million words. The results are presented.

1. NumWords, chunk_size, Speedup
2. 1M 10 1.28529
3. 1M 100 1.43695
4. 1M 1000 1.55903
5. 1M 10000 1.61338
6. 10M 10 1.96158
7. 10M 100 2.29966
8. 10M 1000 2.38369
9. 10M 10000 2.48099

The obtained results demonstrate the influence of document length on speedup in relation to chunk size. When the chunk size matches the size of the input document, the speedup is below 1 because only one chunk of data is created. In this case, only one consumer is effectively working, leading to sequential-like execution. A chunk size ranging from 1000 to 10000 is suitable for all situations, as evidenced by the results.

1.5.3 Benchmark corpus size

The speedup measured by varying the length of the input document, in terms of the number of words to be processed, is reported here. In this experiment, we consider bi-grams ($n = 2$) with 4 threads available for OpenMP and a chunk size of 1000. The results are presented.

1. size Speedup
2. 1M 1.32002
3. 5M 1.59175
4. 10M 2.17017
5. 50M 2.86278
6. 100M 2.82635
7. 200M 2.76608

The influence of document length on speedup is evident. For small input documents, the sequential implementation is the quickest because the histogram merging phase, which is almost sequential, becomes the most time-consuming task in the parallel implementation. As the size of the input document increases, the n-gram generation and subsequent counting, which can be executed independently by each thread, become the hotspot of the parallel program. This results in improved performance and notable speedup.

1.5.4 Benchmark n-grams

The speedup measured for varying n is reported here. In this experiment, we consider n-grams ranging from uni-grams to 5-grams, with 4 threads available for OpenMP. The input document contains 10 million words, and the chunk size is set to 10,000 words.

1. n-grams, Speedup
2. 1 2.43676
3. 2 2.19444
4. 3 1.55526

5. 4 1.38305
6. 5 1.33114

As n increases, the speedup decreases. This reduction in speedup is due to the larger histograms generated for higher values of n. Consequently, the merging phase in the parallel implementation becomes more time-consuming, leading to delays in output production.

1.5.5 Books Speedup

Finally, an analysis of execution times for both sequential and parallel implementations in a real-world scenario is reported. This analysis involves bi-grams extraction from 5 books available in Project Gutenberg.

1. Book, Length, Speedup
2. The Illustrated War News 13098 2.17844
3. My further disillusionment 39496 2.14223
4. anna karenine 194145 2.11632
5. bible.txt 1029954 2.45048

We observe that a good speedup is achieved for each text corpus considered. Utilizing a larger chunk size may further improve performance for large text corpora, such as the Bible.

1.6. Conclusion

This work presents a detailed comparison between a sequential implementation and its parallel counterpart for a word n-grams histogram extractor. The results demonstrate the efficiency of the parallel version in real-world scenarios. Thanks to OpenMP, this implementation can run on any hardware, leveraging all the computing units provided by the CPU. By avoiding pathological cases, it consistently achieves good performance in terms of speedup.

The implementation can be easily adapted to utilize multiple Producer threads. For example, if two distinct files need to be processed, each file could be assigned to a separate Producer, while maintaining the existing setup for the rest of the process.

References

[1] OpenMP Architecture Review Board. (2023). OpenMP Application Programming Interface. Retrieved from `https://www.openmp.org/specifications/`