Report Bigram-Trigram parallel computing

Veronica Sgamuffa

February 2026

1 Introduction

This project concerns the extraction of bigrams and trigrams of characters and words and the creation of histograms with them. The text examined for such extraction is the English version of Tolstoy's book "War and Peace" retrieved through the Gutenberg Project; to increase the corpus, the text was multiplied 100 times (for a total of 56,843,500 words and over 322 million characters) and 300 times (for a total of 170,530,500 words and over 968 million characters) to assess differences in the results, and saved in the text file input.txt . To run the project, written in C++ with the use of the OpenMP framework for functions working in parallel, a pc was used that has the specifications shown in the following table:

CPU	AMD Ryzen 7 7730U
Core/Threads	8/16
RAM memory	16GB

2 Implementation

Two types of n-grams were implemented in the project: n-grams of characters and n-grams of words. Although the ultimate goal is similar, namely to construct a histogram of frequencies, the two versions have differences, particularly from the point of view of the data structures used. In order to analyze the character and word bigrams/trigrams, several files were created:

- **CharacterNgrams.cpp**
- **CharacterNgrams.h**
- **CMakeLists.txt**
- **input.txt**
- **main.cpp**
- **plot_ngrams.py**
- **plot_scaling.py**
- **Utils.cpp**
- **Utils.h**
- **WordNgrams.cpp**
- **WordNgrams.h**

Those of most interest for the actual construction are main.cpp, CharacterNgrams.cpp and WordNgrams.cpp .

The code in main.cpp performs text pre-processing and performance benchmarking. In a first step, the clean_all function reads the original input file (input.txt) and produces a cleaned-up file containing only ASCII characters, turning sequences of spaces into a single space and converting all punctuation, that is, turning letters into lowercase and keeping numbers and words on a single line. Next, in the main function, the CharacterNgrams and WordNgrams objects are instantiated. The benchmark_scaling function then performs a series of benchmarks by varying the number of threads from 1 to 32: for each value it measures the average execution time of the sequential and parallel versions of the two algorithms, repeating the measurements 10 times to reduce variability. The results are finally saved in

the file scaling_results.csv, which can be used to analyze the speedup and scalability of parallel solutions.

CharacterNgrams.cpp consists of 4 functions:

- **print_character_histogram** : reconstructs and sorts n-grams from the histogram
- **compute_character_ngrams** : calculates n-grams of characters in sequential version
- **parallel_compute_character_ngrams** : calculates n-grams of characters in parallel version with OpenMP
- **getNgramLength** : returns the length of the n-grams

WordNgrams.cpp also consists of 4 functions:

- **print_histogram** : sort the histogram of the n-word histograms
- **compute_word_ngrams** : calculates n-grams of words in sequential version
- **parallel_compute_word_ngrams** : calculates n-grams of words in parallel version through OpenMP
- **getNgramLength** : returns the length of the n-grams

For both files, the sequential and parallel versions were designed to be algorithmically and structurally equivalent to ensure proper comparison of performance and speedup achieved through OpenMP parallelization.

In character n-grams, the basic unit is the single character. The n-grams are generated by scrolling windows of length n within each word.

In word n-grams, the basic unit is the whole word. The n-grams are constructed by concatenating n consecutive words of the text.

For character n-grams it was possible to use an indexed histogram, implemented as **vector<int>**. This is made possible by the fact that the allowable alphabet is limited (lowercase letters [a-z] and numbers [0-9]) and each n-gram can be encoded as a number in base 36. The advantages of this choice are direct O(1) access cost, contiguous memory, and significant reduction in overhead compared to other solutions, such as an unordered_map. This structure also lends itself very well to parallelization because it allows local histograms per thread with fixed size and simple reduction by element-by-element summation.

```
double CharacterNgrams::
    parallel_compute_character_ngrams(const
    std::string& filename) {

 ifstream file(filename, ios::binary);
 if (!file) {
   cerr << "Error opening file\n";
   return 1;
 }
 string content((istreambuf_iterator<char>(
     file)), istreambuf_iterator<char>());
 file.close();

 stringstream ss(content);
 string word;
 int count = 0;
 while (ss >> word) {
   count++;
 }
 ss.clear();
 ss.seekg(0, ios::beg);

 vector<string_view> words;
 words.reserve(count);
 while (ss >> word) {
   words.push_back(word);
 }

 int n = getNgramLength();
 auto num_slots = static_cast<size_t>(pow
     (36, n));

 vector<int> histogram(num_slots, 0);

 int threads = omp_get_max_threads();
 vector<vector<int>> local_histograms(
   threads, vector<int>(num_slots, 0)
 );

 auto start_compute = chrono::
     high_resolution_clock::now();

 #pragma omp parallel
 {
   int tid = omp_get_thread_num();
   auto& local_hist = local_histograms[tid];

   #pragma omp for schedule(static)
```

```
    for (size_t widx = 0; widx < words.size()
        ; ++widx) {
      const auto& w = words[widx];

      for (int p = 0; p <= static_cast<int>(w
          .size()) - n; p++) {
        size_t index = 0;

        for (int i = 0; i < n; i++) {
            const char c = w.at(p + i);
            int offset;

            if (c >= '0' && c <= '9') {
                offset = c - '0';
            } else if (c >= 'a' && c <= 'z')
                {
                offset = c - 'a' + 10;
            } else {
                goto skip_ngram;
            }

            index = index * 36 + offset;
        }

        local_hist[index]++;

      skip_ngram:
        ;
      }
    }
  }

  for (int t = 0; t < threads; ++t) {
    for (size_t i = 0; i < num_slots; ++i) {
      histogram[i] += local_histograms[t][i];
    }
  }
  auto end_compute = chrono::
      high_resolution_clock::now();

  auto start_print = chrono::
      high_resolution_clock::now();
  print_character_histogram(histogram.data(),
       static_cast<int>(num_slots), n);
  auto end_print = chrono::
      high_resolution_clock::now();

  double compute_time =
  chrono::duration<double>(end_compute -
      start_compute).count() + chrono::
      duration<double>(end_print - start_print
      ).count();

  return compute_time;
}
```

Listing 1: Character n-gram parallel function

In the case of word n-grams, the key space is potentially very large and not known a priori, as well as being dependent on the vocabulary of the text. Therefore, as the number of possible word n-grams grows rapidly and an indexed structure would be impractical in terms of memory, it was deemed appropriate to use a **unordered_map<string,int>**; as downsides, however, we have that this choice introduces overhead and higher contention and reduction costs in the parallel version.

```
double WordNgrams::
    parallel_compute_word_ngrams(const string
    & filename) {

 ifstream file(filename);
 if (!file) {
     cout << "Error opening input txt file"
         << endl;
     return 1;
 }

 vector<string> words;
 string word;
 while (file >> word) {
     words.push_back(word);
 }
 file.close();

 int n = this->getNgramLength();
 int threads = omp_get_max_threads();

 vector<unordered_map<string, int>>
     local_hist(threads);
 for (auto &h : local_hist)
     h.reserve(words.size() / threads);

 auto start_compute = omp_get_wtime();

#pragma omp parallel
 {
     int tid = omp_get_thread_num();
     unordered_map<string, int>& hist =
         local_hist[tid];

#pragma omp for schedule(static)
     for (int i = 0; i <= (int)words.size()
         - n; i++) {
         string ngram = words[i];
         for (int j = 1; j < n; j++) {
             ngram.push_back(' ');
             ngram += words[i + j];
         }
         hist[ngram]++;
```

```
    }
}
auto end_compute = omp_get_wtime();

auto start_reduce = chrono::
    high_resolution_clock::now();
unordered_map<string, int> global_hist;
global_hist.reserve(words.size());

for (auto &hist : local_hist) {
    for (auto &[k, v] : hist) {
        global_hist[k] += v;
    }
}
auto end_reduce = chrono::
    high_resolution_clock::now();

auto start_print = chrono::
    high_resolution_clock::now();
print_histogram(global_hist);

auto end_print = chrono::
    high_resolution_clock::now();

double compute_time = end_compute -
    start_compute;

double print_time = chrono::duration<double
    >(end_print - start_print).count();

double reduce_time = chrono::duration<
    double>(end_reduce - start_reduce).count
    ();

return compute_time + print_time +
    reduce_time;

}
```

Listing 2: Words n-gram parallel function

The implementation choices adopted for character and word n-grams derive directly from the intrinsic characteristics of the processed data. Character n-grams allow for a more compact and efficient implementation, while word n-grams require more flexible structures at the cost of higher overhead.This difference is clearly reflected in the experimental results and the observed speedup, which highlights how the structure of the problem crucially affects the effectiveness of parallelization.

3 Performance analysis

A python file (plot_ngram.py) was created to generate histograms as bar graphs. This Python script aims to graphically display the distribution of n-grams from a CSV file generated during the analysis phase. The program uses the pandas library to read the CSV file containing the n-grams and their frequencies, and matplotlib to create the graph. After reading the data, the n-grams are sorted in descending order of frequency. I chose to limit the display of the n-grams to the first 30 most frequent to make the graphs readable and meaningful because otherwise printing them all, as noted in figure reffig:allwords, would have been impossible to distinguish bars and names of individual n-grams. The result is saved as a PNG image.

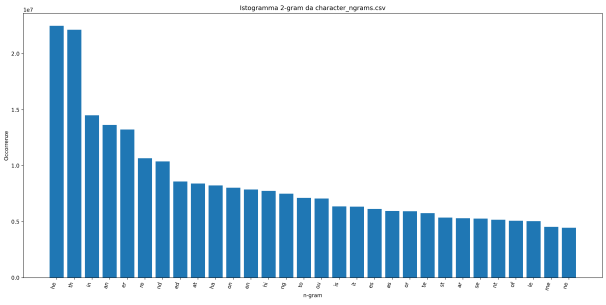

Figure 1: Histogram of top 30 characters' bigrams

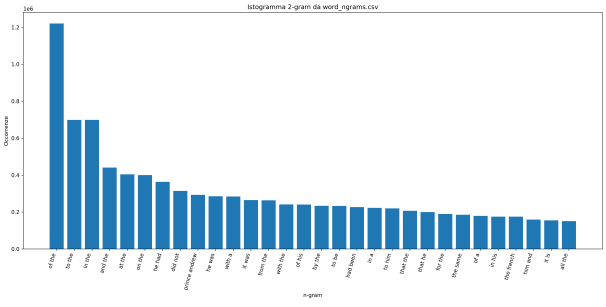

Figure 2: Histogram of top 30 words bigrams

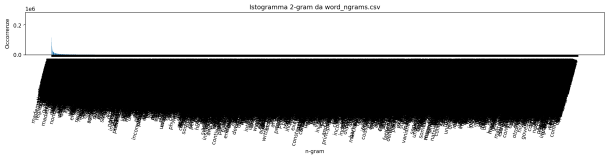

Figure 3: Histogram of all words bigrams

Again, the process was repeated with both bigrams and trigrams of characters and words (with the text input equal to 300 times the original).

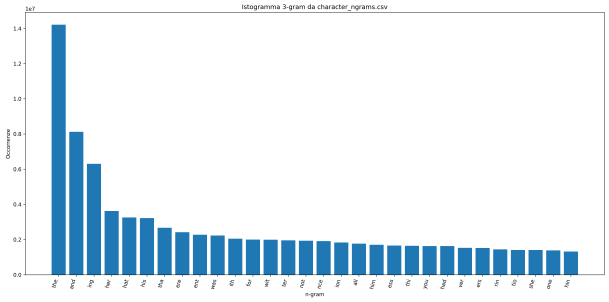

Figure 4: Histogram of top 30 characters trigrams

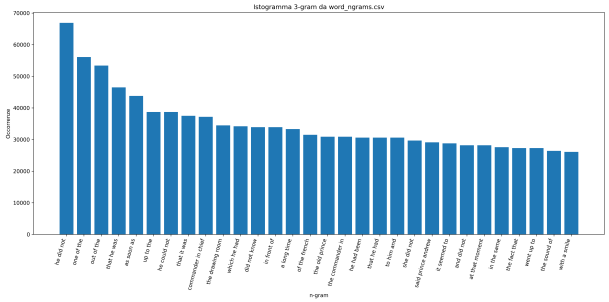

Figure 5: Histogram of top 30 words trigrams

To analyze the performance and scalability of sequential and parallel implementations of n-gram computation and generate graphs, the python file plot_scaling.py was created. Starting with a CSV file containing execution times as the number of threads varies, the program reads the data via the pandas library and calculates the speedup as the ratio of sequential to parallel time for both character n-grams and word n-grams. Then, using matplotlib, three graphs are generated: a graph of speedup as a function of the number of threads, which is useful for evaluating the efficiency of parallelization, and two graphs of execution times comparing sequential and parallel versions for each type of n-gram. Each graph is saved in PNG format.

Let us now look in detail at the results for each case study addressed. Since the parallelizable part is limited only to the main cycle of the functions because the rest (such as reading the file) is identical in both the sequential and parallel functions, to calculate the speedup we were limited to using only the time related to that part.

3.1 Bigram (input x100)

In the case of calculating bigrams with the input.txt equal to 100 times the original, with character bigrams we note that the time taken to run the loop in the sequential version is between 0.5 and 0.6 seconds, while the time taken for the parallel version decreases rapidly as the number of threads increases until it stabilizes at 0.1 from 15 threads onward (lesser time that at 16 threads, then it grows minimally).

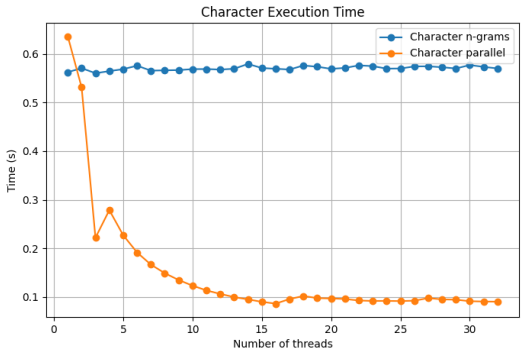

Figure 6: Execution time of characters bigrams

Regarding word bigrams in the parallel case, we see that the time gradually decreases until we reach 16 threads, after which time begins to increase again. Up to this threshold, the workload is effectively distributed over the threads relative to the available

cores, allowing a significant reduction in execution time. Beyond 16 threads, the number of threads exceeds the number of threads relative to the threads corresponding to the machine cores (oversubscription), causing an increase in context switching. In addition, contention over shared resources grows and added to these factors is the management and synchronization overhead introduced by OpenMP. Finally, the analyzed text probably does not represent a sufficiently high workload to offset these costs.

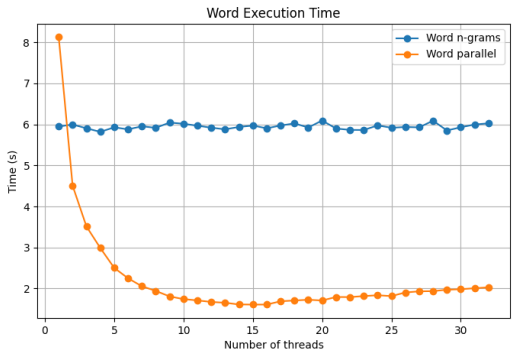

Figure 7: Execution time of words bigrams

The speedup we observe is a consequence of the considerations just made: in fact, it can be seen that we get the greatest speedup in the case of characters at 16 threads (6.6x), while in the case of words at 15 threads (3.7x).

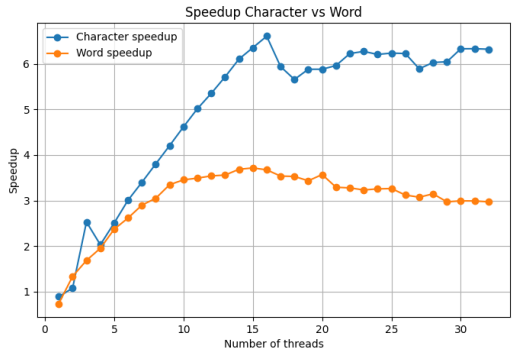

Figure 8: Speedup bigram

3.2 Trigram (input x100)

Also with the execution of trigrams we find ourselves in a situation similar to that observed in the execution of bigrams. In the case of the characters the cycle execution time of the sequential version varies by small differences of hundredths of a second, while in the parallel counterpart we again observe the execution time decreasing gradually until we reach 16 threads, beyond which it rises slightly.

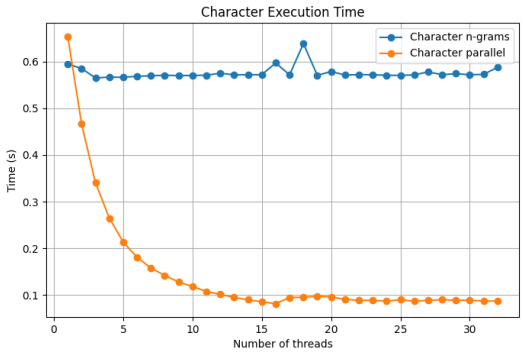

Figure 9: Execution time of character trigram

Even in the case of words trigrams we have a similar situation (sequential constant time, parallel time decreases up to 14 threads and then rises again).

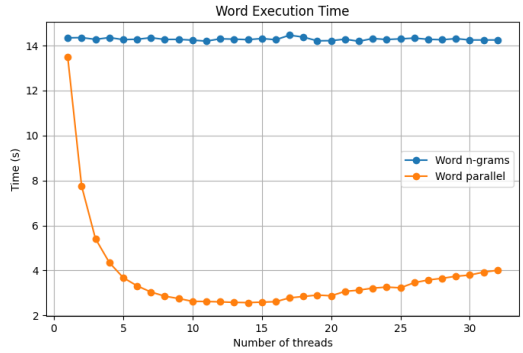

Figure 10: Execution time of words trigram

Analyzing the speedup results we see that in the

case of words trigrams we have the best speedup near 14 threads (5.6x), while in the case of characters we have it on 16 threads (7.3x). Comparing these results with previous results for bigrams, we note that the speedup is better in both cases, mainly due to ratio of useful work to overhead.

Figure 11: Speedup trigram

3.3 Bigram (input x300)

Analyzing the execution times in the case of the input multiplied 300 times for bigram analysis, the line representing the parallel function time drops to the minimum at the 16 threads.

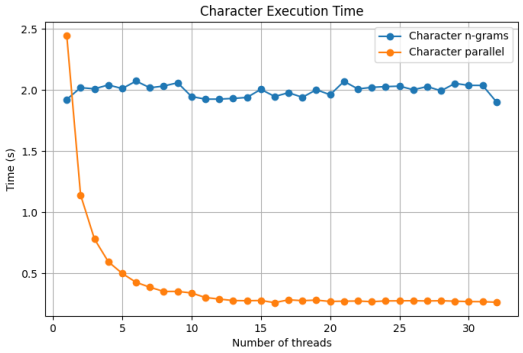

Figure 12: Execution time of character bigram

In the case of words bigrams, the average execution time of the sequential version takes about 17-18 seconds, while the parallel version decreases as might be expected until it reaches a minimum of 3 seconds near 16 threads.

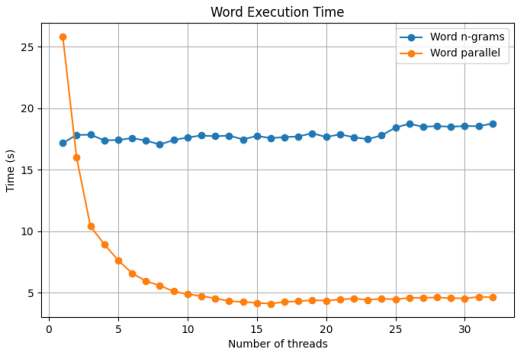

Figure 13: Execution time of words bigram

Regarding speedups, in both the case of characters (7.4x) and words (4.3x) we get the maximum speedup on the 16 threads. Comparing the results obtained with those returned by running bigrams with x100 input, we notice that in both cases there is a slight improvement in performance.

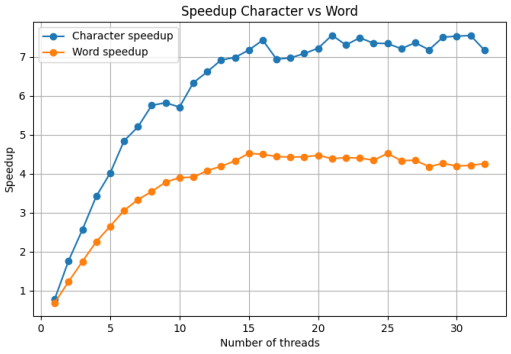

Figure 14: Speedup bigram

3.4 Trigram (input x300)

Again, the line representing the execution time of character trigrams finds the minimum at 16 threads, with a value of 0.25 seconds.

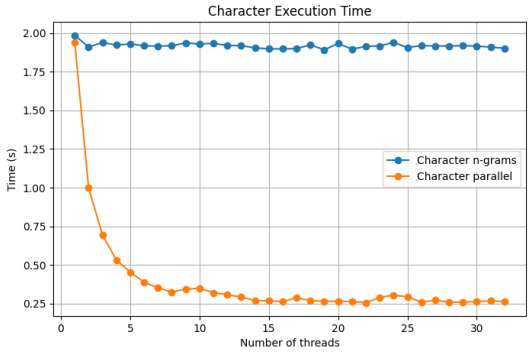

Figure 15: Execution time of character trigram

In the execution of word trigrams, the minimum regarding the execution time of the parallel function cycle is found in the proximities of 15 threads, after which the time slowly and progressively worsens.

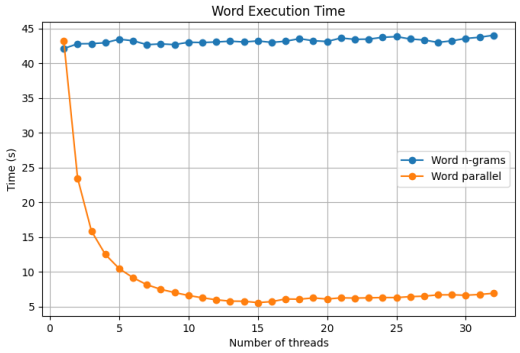

Figure 16: Execution time of words trigram

In the case of the word trigrams, the best speedup, which is 7.8x, is obtained on the 15 threads, while in the case of the character trigrams on the 22 threads (7.5x), but we understand that it is due to small fluctuations in the runtime capture since after the 16 threads (speedup 7.2x) we see a plateau in the curve. Comparing the results with those obtained in the case of the trigrams with x100 input, we notice a slight improvement in the case of the speedup of character trigrams, while a noticeable improvement in the case of the speedup of word trigrams.

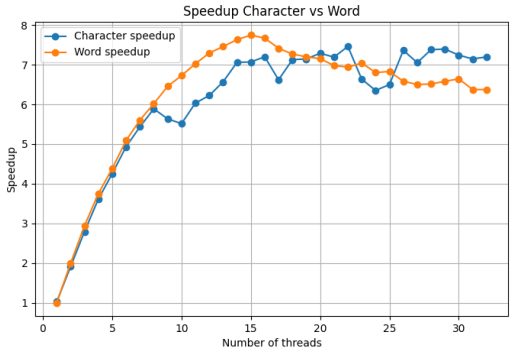

Figure 17: Speedup trigram

4 Conclusions

In this paper, the problem of bigram and trigram extraction of characters and words was analyzed by comparing sequential and parallel implementations based on OpenMP. The experimental results clearly show that the effectiveness of parallelization depends strongly on both the structure of the problem and the workload. Character n-grams, due to the use of an indexed histogram and more regular memory access, are found to be systematically more efficient and scalable than word n-grams, which require more complex data structures such as unordered_maps and introduce more overhead. In all cases analyzed, increasing the number of threads leads to a significant reduction in execution time to a value close to the number of threads of available physical cores, beyond which the benefits diminish. In addition, it was observed that trigrams show better speedups than bigrams, since the higher computational load per unit of data allows for a better amortization of the parallelization overhead.

	Bigram x100	Bigram x300	Trigram x100	Trigram x300
Character	6.6	7.4	7.3	7.5
Words	3.7	4.5	5.6	7.8

Table 1: Confronto speedup

Overall, the results confirm that careful design of data structures and adequate problem size are crucial

factors in achieving real benefits from parallel computing techniques.