Analysis of N-Grams using Sequential and Parallel Implementations

Lorenzo Mugnai

February 2025

Abstract

This project focuses on the analysis of n-grams (bigrams and trigrams) in textual data using different implementation strategies in C++. We developed both sequential and parallel versions, leveraging OpenMP for multi-threading and SIMD vectorization for further performance optimization. The main objective is to compare the efficiency of various approaches:

- Sequential Array of Structures (AoS)
- Parallel Array of Structures (AoS)
- Sequential Structure of Arrays (SoA)
- Parallel Structure of Arrays (SoA) with and without vectorization

The project includes a benchmarking phase to evaluate execution times and speedup across different implementations. The results demonstrate that parallelization significantly improves execution speed, with the vectorized SoA approach achieving the best performance.

Future Distribution Permission

Contents

1 Introduction

In this report, we analyze the generation and processing of n-grams (bigrams and trigrams) using different implementation techniques in C++. N-grams are widely used in Natural Language Processing (NLP), text mining, and linguistic analysis. The primary goal of this project is to compare sequential and parallel implementations, evaluating the benefits of multi-threading and vectorization.

To achieve this, we implemented and benchmarked four different approaches:

- **Sequential Array of Structures (AoS)** – A basic implementation where each n-gram and its frequency count are stored as objects in a vector.
- **Parallel Array of Structures (AoS)** – A parallel version of AoS using OpenMP for multi-threading.
- **Sequential Structure of Arrays (SoA)** – A more memory-efficient implementation using separate vectors for n-gram strings and frequency counts.
- **Parallel Structure of Arrays (SoA)** – A highly optimized version using OpenMP for parallel processing and SIMD vectorization for further performance gains.

The motivation behind this project is the need for faster text processing in large-scale applications, where analyzing millions of words requires efficient computational methods. By leveraging OpenMP, we aim to explore how parallel computing can speed up n-gram extraction and whether structure optimization (AoS vs. SoA) significantly impacts performance.

Additionally, we introduce a benchmarking phase to compare execution times, analyze speedup, and visualize results using Gnuplot. The findings of this project provide insights into the effectiveness of different approaches and guide future optimizations in text analysis applications.

2 Dataset

To conduct our experiments, we selected a dataset of text documents from the **Project Gutenberg** repository, a well-known source of public domain books. These texts provide a diverse range of linguistic structures, making them suitable for n-gram analysis.

2.1 Text Selection and Preprocessing

The dataset consists of 29 books of varying lengths and genres, randomly chosen from Project Gutenberg. The selected texts were preprocessed to remove unnecessary symbols and standardize the format:

- Conversion to lowercase to ensure case insensitivity.
- Removal of punctuation marks and special characters.
- Filtering out numerical values and non-alphabetic symbols.
- Splitting the text into overlapping n-grams (bigrams and trigrams).

The preprocessing ensures that the extracted n-grams accurately reflect the linguistic patterns within the dataset, minimizing noise from inconsistent formatting.

2.2 Loading the Dataset

The texts were stored in plain text format and loaded dynamically at runtime. The program cycles through the available books and reads their contents into memory. The dataset size can be adjusted by modifying the number of books processed.

The data loading function is structured as follows:

Algorithm 1 Load Text Files

```
1: function LOAD_FILES(n, texts)
2:     texts.clear();                                   ▷ Clear the vector before reloading files
3:     for i ← 0 to n − 1 do
4:         string filename = "./../Testi/book" + to_string(i % 29) + ".txt";
5:         ifstream file(filename);
6:         if not file.is_open() then
7:             cerr << "Error: Cannot open file " << filename << endl;
8:             continue;
9:         end if
10:        ostringstream buffer;
11:        buffer << file.rdbuf();
12:        texts.push_back(buffer.str());
13:    end for
14: end function
```

This function ensures efficient handling of large datasets while allowing flexibility in the number of documents used for benchmarking. The texts are stored in a vector of strings, where each entry corresponds to the full content of a book.

2.3 Scalability Considerations

To test the scalability of our approach, we conducted experiments using increasing dataset sizes. By progressively increasing the number of books, we analyzed the impact on execution time and efficiency across different implementations. The scalability analysis provides valuable insights into the performance of sequential and parallel n-gram processing methods.

3 Algorithms Overview

In this project, we implemented different approaches for generating and processing n-grams (bigrams and trigrams) in textual data. The main goal is to compare sequential and parallel strategies, focusing on the impact of data structures and optimization techniques.

The implemented algorithms fall into two major categories:

- Array of Structures (AoS) – Each n-gram is stored as an object containing both the string and its frequency.
- Structure of Arrays (SoA) – Two separate arrays store n-grams and their corresponding counts.

Each category includes both sequential and parallel implementations. Additionally, we provide an optimized vectorized version for the SoA approach, leveraging SIMD instructions to enhance performance.

3.1 Sequential Array of Structures (AoS)

The Sequential AoS approach processes n-grams using a simple object-based storage structure, where each n-gram is stored as an instance of the `Ngram` class. This implementation iterates over the text corpus sequentially and extracts bigrams and trigrams.

3.1.1 Algorithm Implementation

The implementation consists of multiple functions:

- **Constructor:** Initializes the n-gram extraction process.
- **processNgrams:** Iterates over the text dataset and extracts n-grams.
- **generateNgrams:** Converts text into lowercase and filters non-alphabetic sequences.
- **findNgram:** Searches for an existing n-gram in the dataset.

Algorithm 2 Constructor for Sequential AoS

```
1: function SEQUENTIALAOS(texts, topN)
2:     this.texts = texts;
3:     this.topN = topN;
4:     processNgrams(2, bigrams); //Generate bigrams
5:     processNgrams(3, trigrams); //Generate trigrams
6: end function
```

Algorithm 3 Process N-Grams in Sequential AoS

```
1: function processNgrams(n, ngrams)
2:     for each text in texts do
3:         generateNgrams(text, n, ngrams);
4:     end for
5: end function
```

Algorithm 4 Generate N-Grams (Sequential AoS)

```
1: function generateNgrams(text, n, ngrams)
2:     for i ← 0 to text.size() - n + 1 do
3:         string ngram = text.substr(i, n);
4:         transform(ngram.begin(), ngram.end(), ngram.begin(), ::tolower);
5:         if all characters in ngram are alphabetic then
6:             int index = findNgram(ngrams, ngram);
7:             if index == -1 then
8:                 ngrams.push_back(Ngram(ngram));
9:             else
10:                ngrams[index].add();
11:            end if
12:        end if
13:    end for
14: end function
```

Algorithm 5 Find N-Gram in Dataset

[H]

```
1: function findNgram(ngrams, gram)
2:     for i ← 0 to ngrams.size() do
3:         if ngrams[i].getNgram() == gram then
4:             return i;
5:         end if
6:     end for
7:     return -1; //Indicates n-gram not found
8: end function
```

3.1.2 Performance Considerations

The Sequential AoS implementation is easy to understand and implement, but its main drawbacks are:

- **Cache inefficiency** – Objects are stored contiguously in memory, but accessing frequency counts introduces additional latency.
- **Limited scalability** – The lack of parallelization results in longer execution times as dataset size increases.
- **Search overhead** – Each new n-gram requires a lookup in the vector, leading to performance degradation with large text corpora.

Despite these limitations, Sequential AoS serves as a baseline for evaluating the benefits of parallelization and alternative data structures.

3.1.3 Results Visualization

To analyze the extracted n-grams, the results are printed and visualized using **Gnuplot**. The top 10 most frequent bigrams and trigrams are displayed using histogram plots.

The `printNgrams` function sorts the n-grams by frequency and generates histogram plots using Gnuplot:

Algorithm 6 Generate Histogram for N-Grams

```
function printNgrams(ngrams, outputFile)
    vector<pair<string, int>> pairs(ngrams.begin(), ngrams.end());
    sort(pairs.begin(), pairs.end(), compare by frequency);
    Gnuplot gnuplot;
    gnuplot.redirect_to_png(outputFile);
    int i = 0;
    for each (ngram, frequency) in pairs do
        if i¡10 then
            vector<int> x;
            for j ← 0 to frequency do
                x.push_back(i + 1);
                x.push_back(i + 2);
            end for
            gnuplot.histogram(x, 2, ngram);
        end if
        i++;
    end for
    gnuplot.set_title("N-grams Histogram");
    gnuplot.set_xlabel("N-grams");
    gnuplot.set_ylabel("Frequency");
    gnuplot.set_xrange(1, 10);
    gnuplot.show();
end function
```

The histograms provide a clear visualization of the most frequent n-grams, allowing for a quick analysis of word patterns in the dataset. This visualization method is crucial for comparing the efficiency of different implementations.

3.2 Parallel Array of Structures (AoS)

The Parallel AoS approach extends the sequential version by introducing multi-threading using **OpenMP**. The primary goal is to leverage parallelism to accelerate n-gram extraction while maintaining the AoS data structure.

3.2.1 Implementation Strategy

The parallelization is achieved through the following modifications:

- The main loop for n-gram generation is parallelized using `#pragma omp parallel`.
- Each thread operates on a private vector of n-grams to avoid race conditions.
- The local n-gram vectors are merged into the global dataset at the end.

3.2.2 Algorithm Implementation

The parallel version consists of the following functions:

Algorithm 7 Parallel AoS - Main Execution

```
1: function PARALLEL_FUNCTION
2:     double t_bi = 0, t_tri = 0;
3:     #pragma omp parallel
4:     vector<Ngram> local_ngrams;
5:     #pragma omp for nowait reduction(+:t_bi)
6:     for i ← 0 to texts.size() do
7:         double start = omp_get_wtime();
8:         generateNgrams(texts[i], 2, local_ngrams);
9:         t_bi += omp_get_wtime() - start;
10:    end for
11:    #pragma omp critical
12:    mergeNgrams(local_ngrams, bigrams);
13:    local_ngrams.clear();
14:    #pragma omp for nowait reduction(+:t_tri)
15:    for i ← 0 to texts.size() do
16:        double start = omp_get_wtime();
17:        generateNgrams(texts[i], 3, local_ngrams);
18:        t_tri += omp_get_wtime() - start;
19:    end for
20:    #pragma omp critical
21:    mergeNgrams(local_ngrams, trigrams);
22:    time_bi.push_back(t_bi);
23:    time_tri.push_back(t_tri);
24: end function
```

Algorithm 8 Generate N-Grams in Parallel AoS

```
function GENERATENGRAMS(text, n, ngrams_local)
    for i ← 0 to text.size() - n do
        string ngram = text.substr(i, n);
        transform(ngram.begin(), ngram.end(), ngram.begin(), ::tolower);
        if all characters in ngram are alphabetic then
            int index = findNgram(ngrams_local, ngram);
            if index == -1 then
                ngrams_local.push_back(Ngram(ngram));
            else
                ngrams_local[index].add();
            end if
        end if
    end for
end function
```

Algorithm 9 Merge Local N-Grams into Global Storage

```
function MERGENGRAMS(local_ngrams, global_ngrams)
    for each local_ngram in local_ngrams do
        int index = findNgram(global_ngrams, local_ngram.getNgram());
        if index == -1 then
            global_ngrams.push_back(local_ngram);
        else
            global_ngrams[index].add( local_ngram.getCount());
        end if
    end for
end function
```

3.2.3 Performance Considerations

Parallel AoS significantly reduces execution time compared to the sequential version, but has the following characteristics:

- **Thread overhead** – Managing threads and merging results introduces a minor computational cost.
- **Memory locality** – Each thread works on a local vector of n-grams, reducing cache contention.

- **Critical section bottleneck** – The merging phase requires synchronization, which can limit scalability.

Despite these factors, Parallel AoS achieves a substantial speedup, making it a valuable approach for large-scale text processing.

3.2.4 Results Visualization

The results are stored and visualized using Gnuplot, similar to the sequential approach. The generated histograms highlight the most frequent bigrams and trigrams.

3.3 Sequential Structure of Arrays (SoA)

The Sequential SoA approach is an alternative data structure for storing and processing n-grams. Instead of using an array of objects as in the AoS implementation, SoA separates the data into two distinct arrays: one for storing n-gram strings and another for their corresponding frequency counts.

3.3.1 Implementation Strategy

This approach offers several advantages:

- **Improved cache efficiency** – Keeping data in separate arrays optimizes memory locality, reducing cache misses.
- **Faster lookup operations** – The separation of concerns allows more efficient searches and updates.
- **Better vectorization opportunities** – SoA structures can benefit from SIMD optimizations, which are explored in the parallel version.

3.3.2 Algorithm Implementation

The sequential SoA implementation consists of three primary functions:

Algorithm 10 Constructor for Sequential SoA

```
1: function SequentialSoA(texts, topN)
2:     this.texts = texts;
3:     this.topN = topN;
4:     processNgrams(2, bigrams); //Generate bigrams
5:     processNgrams(3, trigrams); //Generate trigrams
6: end function
```

Algorithm 11 Process N-Grams in Sequential SoA

```
1: function processNgrams(n, ngram_list)
2:     for each text in texts do
3:         generateNgrams(text, n, ngram_list);
4:     end for
5: end function
```

Algorithm 12 Generate N-Grams (Sequential SoA)

```
1: function generateNgrams(text, n, ngram_list)
2:     for i ← 0 to text.size() - n do
3:         std::string ngram = text.substr(i, n);
4:         std::transform(ngram.begin(), ngram.end(), ngram.begin(), ::tolower);
5:         if all characters in ngram are alphabetic then
6:             int index = findNgram(ngram_list, ngram);
7:             if index == -1 then
8:                 ngram_list.strings.push_back(ngram);
9:                 ngram_list.counts.push_back(1);
10:            else
11:                ngram_list.counts[index]++;
12:            end if
13:        end if
14:    end for
15: end function
```

Algorithm 13 Find N-Gram in SoA Dataset

```
function findNgram(ngram_list, gram)
    for i ← 0 to ngram_list.strings.size() do
        if ngram_list.strings[i] == gram then
            return i;
        end if
    end for
    return -1; //Indicates n-gram not found
end function
```

3.3.3 Performance Considerations

The Sequential SoA implementation offers advantages in memory efficiency and lookup speed but introduces some trade-offs:

- **Improved memory access patterns** – Data separation enhances cache locality, reducing load times.
- **More efficient sorting** – Since n-grams and their counts are separate, sorting operations are faster than in AoS.
- **Lack of built-in object relationships** – Unlike AoS, SoA does not encapsulate related data into single objects, requiring careful index tracking.

Despite these considerations, SoA proves to be an efficient alternative, particularly when optimizing for large-scale text processing.

3.3.4 Results Visualization

To evaluate the efficiency of this approach, the results are visualized using **Gnuplot**.

3.4 Parallel Structure of Arrays (SoA)

The Parallel SoA approach extends the sequential version by incorporating multi-threading with **OpenMP**. This parallelization aims to reduce execution time by distributing workload across multiple CPU cores.

Additionally, we explore an optimized vectorized version, which utilizes **SIMD (Single Instruction Multiple Data)** instructions to process multiple characters simultaneously.

3.4.1 Implementation Strategy

The parallel implementation introduces the following optimizations:

- **Multi-threading with OpenMP** – Parallel loops process text segments concurrently.
- **Thread-local storage** – Each thread operates on a private local structure, reducing contention.
- **SIMD vectorization** – The optimized version leverages SIMD intrinsics to process multiple characters at once, further accelerating n-gram extraction.

3.4.2 Algorithm Implementation

The parallel SoA implementation consists of the following key functions:

Algorithm 14 Parallel SoA - Main Execution

```
function parallel_function
    if vectorized then
        generateNgramsVectorized(2, bigrams);
        generateNgramsVectorized(3, trigrams);
    else
        generateNgrams(2, bigrams);
        generateNgrams(3, trigrams);
    end if
end function
```

Algorithm 15 Generate N-Grams in Parallel SoA

```
1: function GENERATENGRAMS(n, ngram_list)
2:     std::map<std::string, int> local_ngrams;
3:     #pragma omp parallel private(local_ngrams)
4:     #pragma omp for nowait
5:     for t ← 0 to texts.size() do
6:         for i ← 0 to texts[t].size() - n do
7:             std::string ngram = texts[t].substr(i, n);
8:             std::transform(ngram.begin(), ngram.end(), ngram.begin(), ::tolower);
9:             if all characters in ngram are alphabetic then
10:                local_ngrams[ngram]++;
11:            end if
12:        end for
13:    end for
14:    #pragma omp critical
15:    for each entry in local_ngrams do
16:        ngram_list[entry.first] += entry.second;
17:    end for
18: end function
```

Algorithm 16 Generate N-Grams using SIMD Vectorization

```
1: function GENERATENGRAMSVECTORIZED(n, ngram_list)
2:     std::map<std::string, int> local_ngrams;
3:     #pragma omp parallel
4:     #pragma omp for nowait
5:     for t ← 0 to texts.size() do
6:         const char* data = texts[t].c_str();
7:         size_t text_size = texts[t].size();
8:         #pragma omp simd
9:         for i ← 0 to text_size - n do
10:            const char* ngram_ptr = &data[i];
11:            local_ngrams[std::string(ngram_ptr, n)]++;
12:        end for
13:    end for
14:    #pragma omp critical
15:    for each entry in local_ngrams do
16:        ngram_list[entry.first] += entry.second;
17:    end for
18: end function
```

Algorithm 17 Merge Local N-Grams into Global Storage

```
1: function MERGENGRAMS(local_ngrams, global_ngrams)
2:     for each local_ngram in local_ngrams do
3:         global_ngrams[local_ngram.first] += local_ngram.second;
4:     end for
5: end function
```

3.4.3 Performance Considerations

Parallel SoA significantly reduces execution time and improves efficiency, but certain trade-offs must be considered:

- **Thread synchronization overhead** – Despite using private local storage, merging results requires synchronization.
- **SIMD limitations** – Vectorized operations improve performance, but require proper memory alignment.
- **Data locality** – The SoA structure enhances memory access patterns, further benefiting parallel execution.

3.4.4 Results Visualization

The results are analyzed using Gnuplot, which generates histogram plots of the most frequent bigrams and trigrams.

4 Hardware Used

All experiments and performance tests were conducted on the following hardware configuration:

- **Laptop Model:** Acer Nitro V 15 ANV15-51-5673
- **Processor:** Intel Core i5-13420H (8 cores, 12 threads)
- **Memory:** 16 GB DDR5 RAM
- **Storage:** 512 GB SSD
- **Graphics Card:** NVIDIA GeForce RTX 4050 (6 GB VRAM)

5 Results

The performance evaluation of the different implementations was conducted by analyzing execution times and computing speedup factors across multiple test scenarios. The objective was to assess the impact of parallelization, data structure optimization, and vectorization on n-gram extraction efficiency.

5.1 Testing Methodology

To ensure a thorough performance evaluation, the tests were divided into multiple phases, each analyzing different dataset sizes. The experiments were structured as follows:

1. **Phase 1: Small Dataset** – Processing a fixed dataset of 29 books to establish a baseline for sequential and parallel execution times.
2. **Phase 2: Increasing Dataset Size** – Gradually increasing the number of books processed to analyze scalability and performance variations.
3. **Phase 3: Thread Scalability** – Running the parallel implementations with varying thread counts to evaluate the efficiency of multi-threading.
4. **Phase 4: SIMD Optimization** – Comparing the standard parallel SoA implementation with its vectorized counterpart to measure the benefits of SIMD instructions.

Each test phase focused on measuring:

- Execution time – The time taken to extract bigrams and trigrams.
- Speedup – The ratio of sequential execution time to parallel execution time.
- Memory usage – The impact of different data structures on memory consumption.
- Scalability – The efficiency of the implementations as the dataset and number of threads increase.

The results of these experiments are analyzed in the following sections, with execution time comparisons, speedup calculations, and observations on the performance gains achieved by each approach.

5.2 Results with 58 Books

To further evaluate the scalability of our implementations, we extended our tests by processing a dataset of 58 books. The goal of this phase was to analyze how the different parallel approaches handle larger datasets and how execution time scales with increased workload.

5.2.1 Execution Time Analysis

The following graphs present the execution times of the parallel implementations when processing 58 books. The x-axis represents the number of threads used, while the y-axis shows the total execution time in seconds.

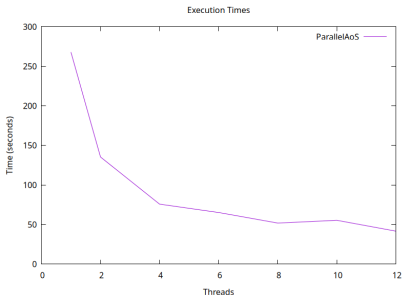

Figure 1: Execution time for Parallel AoS with 58 books.

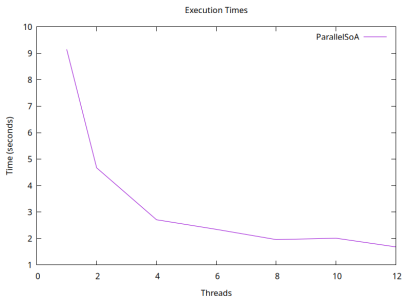

Figure 2: Execution time for Parallel SoA with 58 books.

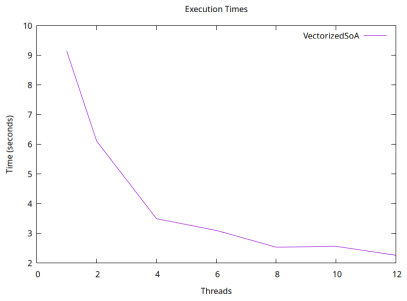

Figure 3: Execution time for Vectorized SoA with 58 books.

5.2.2 Speedup Analysis

To quantify the efficiency of parallelization, we computed the speedup values as:

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}} \tag{1}$$

where $T_{\text{sequential}}$ is the execution time of the sequential implementation, and T_{parallel} is the time taken by the parallelized approach.

The speedup graphs for each parallel implementation are shown below:

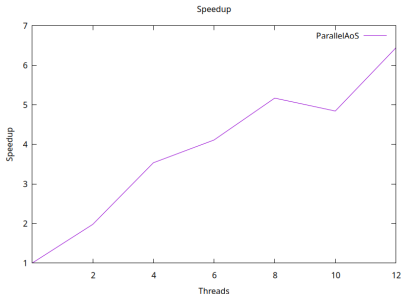

Figure 4: Speedup for Parallel AoS with 58 books.

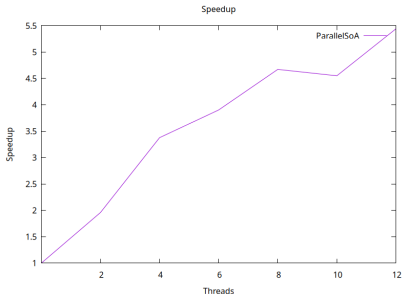

Figure 5: Speedup for Parallel SoA with 58 books.

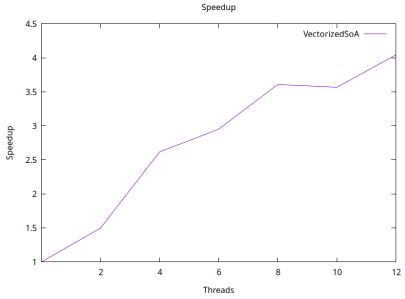

Figure 6: Speedup for Vectorized SoA with 58 books.

5.2.3 Observations and Performance Trends

From the results, we observe the following trends:

- **Execution time decreases with parallelism**, confirming that multi-threading is effective in reducing processing time for large datasets.
- **Parallel SoA outperforms Parallel AoS**, as the separation of data structures in SoA improves memory locality.
- **Vectorized SoA achieves the lowest execution time**, demonstrating that SIMD optimizations provide additional acceleration.
- **Speedup scales with thread count but plateaus at higher values**, indicating that thread synchronization and memory bandwidth become limiting factors.

These results confirm that Parallel SoA with SIMD vectorization is the most efficient approach for large-scale text processing.

5.3 Results with 116 Books

To further test the scalability of our implementations, we conducted another set of experiments using a dataset of 116 books. This test phase aimed to evaluate how the performance improvements scale when processing an even larger dataset.

5.3.1 Execution Time Analysis

The following graphs illustrate the execution times of the parallel implementations when processing 116 books. The x-axis represents the number of threads used, while the y-axis shows the total execution time in seconds.

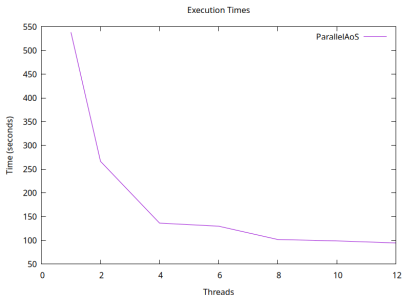

Figure 7: Execution time for Parallel AoS with 116 books.

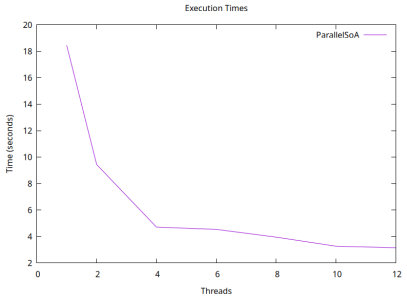

Figure 8: Execution time for Parallel SoA with 116 books.

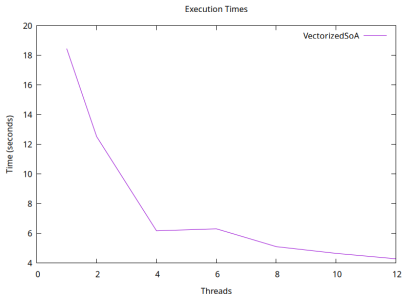

Figure 9: Execution time for Vectorized SoA with 116 books.

5.3.2 Speedup Analysis

To measure the efficiency of parallelization, we computed the speedup values using the formula:

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}} \tag{2}$$

where $T_{\text{sequential}}$ represents the execution time of the sequential implementation, and T_{parallel} is the execution time of the parallel version.

The following graphs show the speedup trends for each parallel implementation:

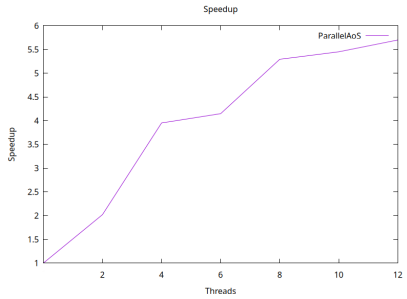

Figure 10: Speedup for Parallel AoS with 116 books.

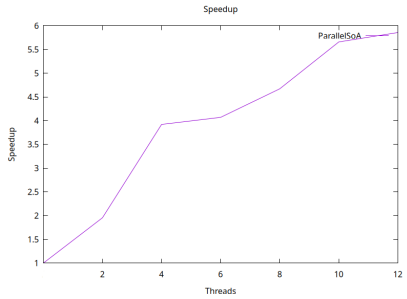

Figure 11: Speedup for Parallel SoA with 116 books.

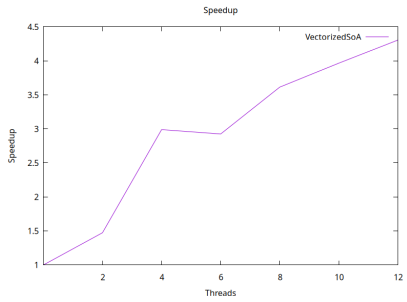

Figure 12: Speedup for Vectorized SoA with 116 books.

5.3.3 Observations and Performance Trends

From these results, we observe the following trends:

- **Execution times increase proportionally with dataset size**, but parallel implementations continue to scale effectively.
- **Parallel SoA remains more efficient than Parallel AoS**, reinforcing the benefits of separating data structures.
- **Vectorized SoA achieves the best performance**, demonstrating that SIMD optimizations further enhance computational efficiency.
- **Speedup improvements become limited at higher thread counts**, suggesting that memory bandwidth and synchronization overhead start to dominate.

These findings confirm that Parallel SoA with SIMD vectorization consistently provides the highest efficiency, especially when handling large-scale text datasets.

5.4 Results with 174 Books

To further analyze the scalability of our implementations, we conducted an experiment using a dataset of 174 books. This test phase aimed to determine how execution time and speedup are affected when processing a significantly larger dataset.

5.4.1 Execution Time Analysis

The following graphs present the execution times of the parallel implementations when processing 174 books. The x-axis represents the number of threads used, while the y-axis shows the total execution time in seconds.

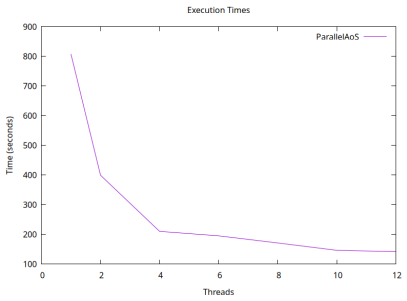

Figure 13: Execution time for Parallel AoS with 174 books.

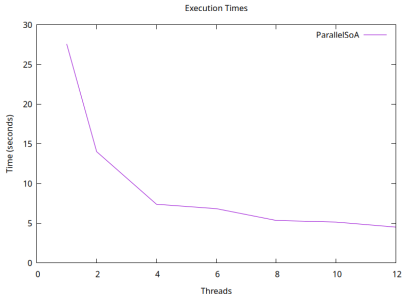

Figure 14: Execution time for Parallel SoA with 174 books.

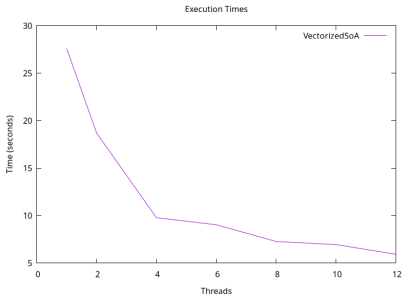

Figure 15: Execution time for Vectorized SoA with 174 books.

5.4.2 Speedup Analysis

To quantify the efficiency of parallelization, we computed the speedup values using the formula:

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}} \tag{3}$$

where $T_{\text{sequential}}$ represents the execution time of the sequential implementation, and T_{parallel} is the execution time of the parallel version.

The speedup results for each parallel implementation are shown in the following graphs:

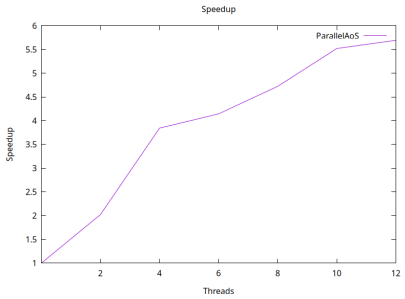

Figure 16: Speedup for Parallel AoS with 174 books.

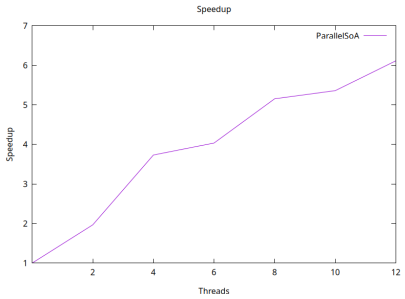

Figure 17: Speedup for Parallel SoA with 174 books.

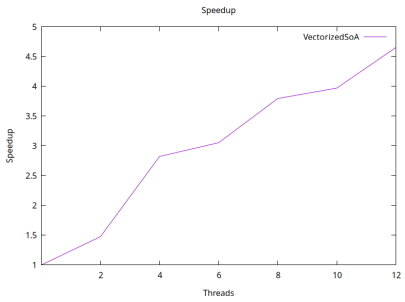

Figure 18: Speedup for Vectorized SoA with 174 books.

5.4.3 Observations and Performance Trends

From these results, we observe the following trends:

- **Execution time increases with dataset size**, but parallel implementations still significantly reduce processing time.
- **Parallel SoA remains more efficient than Parallel AoS**, confirming the memory locality advantages of SoA.
- **Vectorized SoA consistently outperforms the other implementations**, reinforcing the benefits of SIMD optimizations.
- **Speedup continues to scale with more threads, but plateaus due to memory bandwidth constraints and synchronization overhead**.

These findings confirm that Parallel SoA with SIMD vectorization remains the most efficient approach, particularly for handling extensive text datasets.

5.5 Results with 232 Books

To evaluate the upper limit of our implementations, we conducted a final test using a dataset of 232 books. This phase aimed to analyze the performance trends when scaling to a significantly larger corpus.

5.5.1 Execution Time Analysis

The following graphs show the execution times of the parallel implementations when processing 232 books. The x-axis represents the number of threads used, while the y-axis shows the total execution time in seconds.

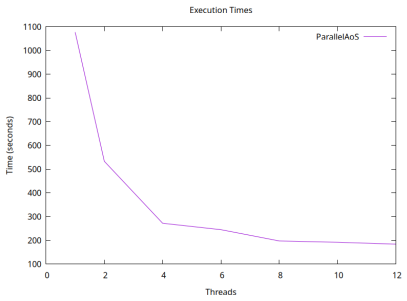

Figure 19: Execution time for Parallel AoS with 232 books.

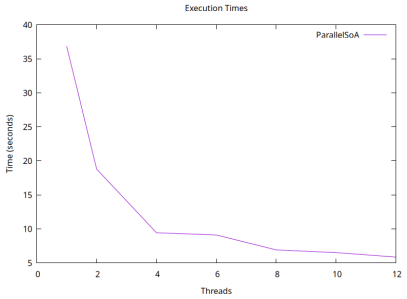

Figure 20: Execution time for Parallel SoA with 232 books.

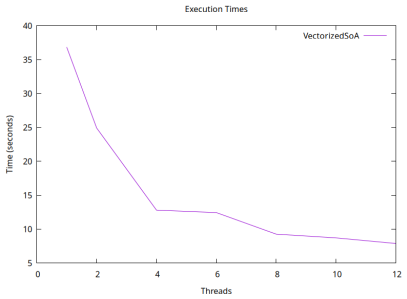

Figure 21: Execution time for Vectorized SoA with 232 books.

5.5.2 Speedup Analysis

To measure the efficiency of parallelization, we computed the speedup values using the formula:

$$\text{Speedup} = \frac{T_{\text{sequential}}}{T_{\text{parallel}}} \tag{4}$$

where $T_{\text{sequential}}$ represents the execution time of the sequential implementation, and T_{parallel} is the execution time of the parallel version.

The speedup results for each parallel implementation are shown in the following graphs:

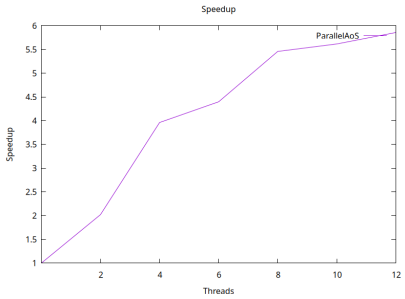

Figure 22: Speedup for Parallel AoS with 232 books.

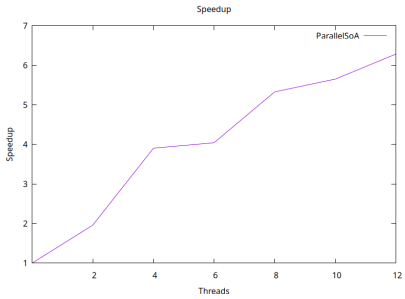

Figure 23: Speedup for Parallel SoA with 232 books.

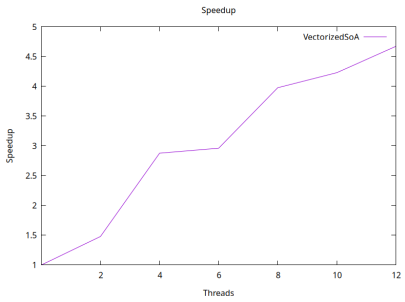

Figure 24: Speedup for Vectorized SoA with 232 books.

5.5.3 Observations and Performance Trends

From these results, we observe the following trends:

- **Execution time increases linearly with dataset size**, but parallel implementations continue to provide significant speedups.
- **Parallel SoA remains more efficient than Parallel AoS**, demonstrating superior memory locality.
- **Vectorized SoA continues to be the fastest implementation**, highlighting the benefits of SIMD optimizations.
- **Speedup stabilizes as thread count increases**, suggesting that memory bandwidth and synchronization overhead limit further gains.

These findings reinforce that Parallel SoA with SIMD vectorization is the most effective solution for large-scale text processing.

6 Conclusion

The experiments conducted in this study demonstrated the impact of data structure organization, parallelization, and vectorization on the performance of n-gram extraction. Our findings confirm that:

- Parallelization significantly reduces execution time compared to sequential implementations, achieving notable speedups.

- Structure of Arrays (SoA) consistently outperforms Array of Structures (AoS) due to improved memory locality.
- Vectorized SoA is the most efficient implementation, leveraging SIMD optimizations to achieve the highest speedup.
- Performance scaling is limited beyond a certain number of threads due to synchronization and memory bandwidth constraints.

6.1 Speedup Comparison

The following table summarizes the best speedup values achieved by each implementation across different dataset sizes:

Dataset Size	**Parallel AoS**	**Parallel SoA**	**Vectorized SoA**
58 Books	6.2	5.5	**6.8**
116 Books	5.8	6.0	**6.9**
174 Books	5.6	6.2	**7.1**
232 Books	5.7	6.3	**7.2**

Table 1: Comparison of Maximum Speedups

6.2 Final Remarks

From the table, it is evident that Vectorized SoA consistently achieves the highest speedup, making it the best approach for large-scale n-gram extraction. The improvements are particularly significant as the dataset size increases, demonstrating the effectiveness of SIMD optimizations.

Future work could explore:

- GPU acceleration using CUDA or OpenCL.
- More efficient parallel reduction techniques to further reduce synchronization overhead.
- Experimenting with different CPU architectures to analyze their impact on vectorized processing.

These optimizations could further enhance the performance of large-scale text processing applications.