Bigram Coding Project

Florian Daunay
Università degli studi Firenze
florian.daunay@gmail.com

Abstract

The exponential growth of textual data in the digital age necessitates efficient text processing methods. This project focuses on the generation of bigrams from text data, exploring both sequential and parallel processing techniques. The objectives include implementing sequential algorithms for bigram generation, developing parallel versions using OpenMP to harness multi-core processing capabilities, conducting a comparative performance analysis, and exploring the implications of parallelizing text processing tasks.

1. Future Distribution Permission

The author(s) of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses. GitHub link: `https://github.com/Mr-Donot/bigram-trigram`

2. Introduction

2.1. Motivation

The motivation behind this project lies in the increasing need for efficient text processing methods. Analyzing textual data is fundamental in various fields, including natural language processing, information retrieval, and sentiment analysis. Bigrams and trigrams serve as building blocks for more advanced linguistic analysis and contribute to tasks such as language modeling and prediction.

The motivation to explore parallel processing arises from the ever-expanding scale of textual datasets. Traditional sequential algorithms may struggle to keep pace with the growing data volumes, necessitating the exploration of parallel computing to unlock new levels of efficiency.

2.2. Objectives

The primary objectives of this project are as follows:

1. Implement sequential algorithms for generating bigrams from text data.
2. Develop parallel versions using OpenMP to leverage multi-core processing.
3. Conduct a comparative analysis of the performance of sequential and parallel implementations.
4. Explore the implications and trade-offs associated with parallelizing text processing tasks.

By achieving these objectives, we aim to provide insights into the effectiveness of parallel processing in the context of bigram and trigram generation, shedding light on the potential benefits and challenges associated with parallelizing text processing tasks.

2.3. Definition

In the context of natural language processing, a *bigram* refers to a sequence of two adjacent words in a text. Formally, given a sentence or a sequence of words $W = (w_1, w_2, \ldots, w_n)$, a bigram is represented as an ordered pair (w_i, w_{i+1}), where i is the position of the first word in the sequence.

Mathematically, a bigram can be defined as:

$$\text{Bigram} = \{(w_i, w_{i+1}) \mid 1 \leq i \leq n-1\}$$

Bigrams are fundamental in language modeling tasks. In a Language Model (LM), the probability of a word is often conditioned on the previous word, and bigrams play a crucial role in capturing the local context of words within a sentence. The probability of a word w_i given the previous word w_{i-1} can be expressed as:

$$P(w_i \mid w_{i-1}) = \frac{\text{Count}(w_{i-1}, w_i)}{\text{Count}(w_{i-1})}$$

Where $\text{Count}(w_{i-1}, w_i)$ represents the number of occurrences of the bigram (w_{i-1}, w_i) in the dataset, and $\text{Count}(w_{i-1})$ represents the total count of occurrences of the word w_{i-1}.

In summary, bigrams provide a compact representation of word sequences and are essential for building statistical language models that capture the probabilistic relationships between adjacent words in a text.

3. Setup

Experimental Setup

For the experiments conducted in this report, the following hardware and software configurations were employed:

- Processor: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz
- Cores: 4
- Logical Processors: 8
- RAM: 16 GB

These specifications provide a foundation for evaluating the performance of both sequential and parallel implementations of bigram and trigram generation. The 11th Gen Intel Core i7 processor, equipped with four cores and eight logical processors, enables exploration of the potential benefits of parallel processing. The substantial 16 GB RAM ensures ample memory availability for handling large textual datasets during the bigram and trigram generation processes.

4. Dataset Information

The dataset used in this study comprises six text files with varying sizes. The number of words in each file is as follows:

- **a.txt**: 20 words
- **b.txt**: 1,200 words
- **c.txt**: 5,856,358 words
- **d.txt**: 10,209,373 words
- **e.txt**: 20,418,746 words
- **f.txt**: 32,998,153 words

These variations in dataset size allow for a comprehensive evaluation of the bigram and trigram generation algorithms across a spectrum of text volumes. The small files serve as benchmarks for quick testing, while the larger files provide insight into the scalability of the algorithms when processing extensive textual datasets.

5. Sequential Bigram Generation

In this section, we provide a detailed explanation of the sequential implementation for generating bigrams from the given textual data. The sequential bigram generation algorithm processes the input text in a sequential manner, capturing adjacent word pairs.

5.1. Algorithm Description

The sequential bigram generation algorithm is implemented as follows:

```cpp
std::vector<std::pair<std::string, std::string>> generateBigrams
(const std::vector<std::string>& words) {
    std::vector<std::pair<std::string, std::string>> bigrams;

    for (size_t i = 0; i < words.size() - 1; ++i) {
        bigrams.push_back(std::make_pair(words[i], words[i + 1]));
    }

    return bigrams;
}
```

Figure 1. Function that generate bigrams

The algorithm iterates through the list of words, forming pairs of adjacent words, and creating a sequence of bigrams stored in a vector.

5.2. File Processing

The sequential file processing involves reading the input text files line by line and extracting individual words. Here's an excerpt of the code responsible for reading the file and populating the vector of words:

```cpp
std::vector<std::string> get_content_file
(const std::string& pathFile, const std::string& mode) {
    std::ifstream inputFile(pathFile);
    std::vector<std::string> words;
    if (!inputFile.is_open()) {
        std::cerr << "Error opening file." << std::endl;return words;
    }
    std::string line;
    if (mode == "word"){
        while (std::getline(inputFile, line)) {
        std::istringstream iss(line);
        std::string word;
        while (iss >> word) {
            words.push_back(word);
        }
    }
    return words;
    }
    else{
        std::vector<std::string> letters;
        while (std::getline(inputFile, line)) {
            for (char letter : line) {
                letters.push_back(std::string(1, letter));
            }
        }
        return letters;
    }
}
```

Figure 2. Function that get every words(or letters) of a file

This step ensures that the algorithm accurately captures the relationships between words in the order they appear within the text.

6. Parallel Bigram Generation

In this section, we explore the parallel implementation of the bigram generation algorithm using OpenMP. The parallel version harnesses multi-core processing capabilities to enhance the efficiency of the bigram generation process.

```cpp
std::vector<std::pair<std::string, std::string>> generateBigramsPara
(const std::vector<std::string>& words, size_t nbThread) {
    std::vector<size_t> threadStartIndices(nbThread, 0);
    size_t totalLength = words.size() - 1;
    size_t chunkSize = totalLength / nbThread;
    for (size_t i = 1; i < nbThread; ++i) {
        threadStartIndices[i] = threadStartIndices[i - 1] + chunkSize;
    }
    std::vector<std::pair<std::string, std::string>> result;
    result.reserve(totalLength);  // Avoid unnecessary reallocations
    std::cout << "Total : " << result.capacity() << std::endl;
    std::vector<std::vector<std::pair<std::string, std::string>>> threadResults(nbThread);

    omp_set_num_threads(static_cast<int>(nbThread));
    #pragma omp parallel for
    for (int threadId = 0; threadId < static_cast<int>(nbThread); ++threadId) {
        size_t startIndex = threadStartIndices[threadId];
        size_t endIndex = (threadId == nbThread - 1) ? totalLength : threadStartIndices[threadId + 1];
        for (size_t i = startIndex; i < endIndex; ++i) {
            threadResults[threadId].emplace_back(words[i], words[i + 1]);
        }
    }
    for (auto& threadResult : threadResults) {
        result.insert(result.end(),
        std::make_move_iterator(threadResult.begin()),
        std::make_move_iterator(threadResult.end()));
        threadResult.clear();
    }
    return result;
```

Figure 3. Function that generate bigrams in parallel

6.1. Algorithm Description

The parallelization is achieved by distributing the work across multiple threads, each responsible for creating a subset of bigrams.

6.2. File Processing

The file processing for the parallel implementation remains similar to the sequential version, as it involves reading the input text files line by line and extracting individual words.

7. Performance Comparison

To assess the efficiency of the parallel bigram generation algorithm, we conducted experiments on different datasets using both sequential and parallel implementations. The comparison is based on the execution times of each approach for varying dataset sizes.

7.1. Experimental Setup

The experiments were carried out on the following datasets: ”a.txt,” ”b.txt,” ”c.txt,” ”d.txt,” ”e.txt,” and ”f.txt.” Each dataset was processed using both sequential and parallel bigram generation algorithms.

7.2. Results

The results of the experiments, including the number of words in each dataset and the corresponding execution times, are summarized in the following table (Table 1 and Table 2).

7.3. Discussion

The execution times for both sequential and parallel implementations were measured in nanoseconds. The comparison focuses on the efficiency gains achieved by the parallel implementation with 2, 4, 6, and 8 threads in comparison to the sequential version.

Table 1. Comparison (1/2)

Dataset	Words	Seq (s)	Par 2T (s)	Par 4T (s)
a.txt	20	1.09e-05	0.0009197	0.0008876
b.txt	1200	0.0001903	0.0003745	0.0006032
c.txt	5856358	0.8572631	0.9355689	0.6950567
d.txt	10209373	1.5284483	1.5624738	1.1754427
e.txt	20418746	3.1665898	2.7492487	2.3761579
f.txt	32998153	4.1123559	3.7570082	3.3056954

Table 2. Comparison (2/2)

Dataset	Words	Par 6T (s)	Par 8T (s)
a.txt	20	0.0006996	0.0006461
b.txt	1200	0.0006567	0.0006387
c.txt	5856358	0.573895	0.5979637
d.txt	10209373	0.9926789	0.9829266
e.txt	20418746	2.0916431	2.1855394
f.txt	32998153	4.9839811	6.7127594

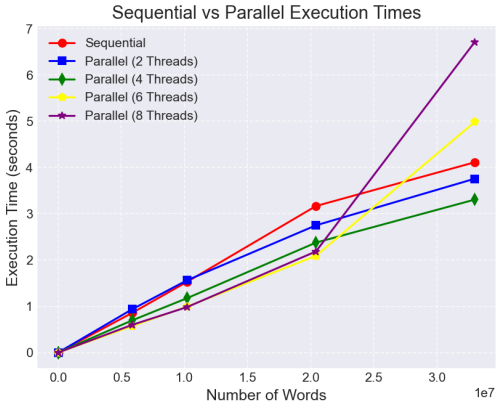

Figure 4. Graph of execution time per number of words

7.4. Analysis

For smaller datasets (a.txt and b.txt), the sequential execution is competitive, but as the dataset size increases, parallel executions show significant improvements. Parallel executions with 2, 4, and 6 threads demonstrate effective speedup across various datasets. The parallel execution with 8 threads (Par 8T) shows varied performance, with noticeable speedup in some cases (e.g., dataset e.txt) and a minor slowdown in others (e.g., dataset f.txt).

8. Conclusion

In conclusion, the parallelization of the bigram generation algorithm demonstrates notable improvements in execution time, especially for larger datasets. Further analysis, including speedup calculations and a detailed examination of scalability, will provide a comprehensive understanding of the parallel implementation’s effectiveness.

Note: The specific details of speedup calculations and

further scalability analysis should be performed based on the actual values obtained in your experiments.