PC-2023/24 Final-Term Project

Giacomo Magistrato
E-mail address
giacomo.magistrato@edu.unifi.

Abstract

The objective of this report is to analyze and evaluate the advantages of implementing parallelism through OpenMp, in relation to the Bigrams and Trigrams histograms. The report aims at highlighting how parallel programming can enhance the efficiency of the algorithm, and under which circumstances, with the eventually encountered limitations. More specifically, the benefits of parallelism have been outlined by implementing the algorithm in C++ and creating a parallel version of the program, in which part of the code is executed using OpenMp. The source code will be released publicly.

1. Introduction

Bigram and trigram histograms are widely used tools in natural language analysis and many other applications that require text processing. A bigram is a sequence of two consecutive words or characters, while a trigram is a sequence of three consecutive words or characters. The histograms of bigrams and trigrams graphically represent the frequency of occurrence of these sequences within a given text corpus.
This practice is widely used in fields like language recognition, text classifications, predictive models. The proposed algorithm analyzes various text files, cleans them of numbers, special characters and images and creates a new file without all this which will be used as input for the algorithms. The language used to implement the algorithm is the C++.

2. The Algorithm

2.1. Ngrams of Words

The algorithm uses istringstream to read the text file and than saves all words in a string vector. Than the vector is iterated and the words are saved in a variable(currentNgram). Once currentNgram reach the lenght n, is added to the histogram and resetted for the next n-gram.
The histogram is implemented using an unorderedMap. This uses words n-grams as the Keys and a counter as Value for each key.

2.2. Ngrams of Characters

As for the n-grams of words, all the words of the file are stored in a string vector. This time the algorithm iterate words and each characters belonging to the word. Characters are temporary saved in a variable (currentN-gram) and added to the histogram once reached the correct length.
The histogram is implemented as for n-grams of words implementation. This time Keys are characters n-grams.

2.3. Printing Phase

This part belong to both Characters and Words N-grams implementation. Once the iteration of the words is finished, printHistogram() function is called. Each element of the unorderedMap is pushed into a priorityQueue which will automatically order all the n-grams in descending order by Value(the counter). Then using pop() function, most common n-grams are popped out and printed. The unorderedMap (the histogram) store each n-gram as Key and the counter for each n-

gram as Value

3. Parallel Version

The parallel version of the algorithm uses Open Mp to implement parallelism. This is done dividing the vector of words into equal parts and assigning each part to a specific thread. Each thread will iterate the part of the vector belonging to it. It will count all the n-grams and update a localHistogram. Once it finished to iterate, the thread update the globalHistogram with the values of the localHistogram that it created. This operation is done by each thread using a omp critical directive in order to avoid concurrency on globalHistogram. Both Words and Characters N-grams counters follows this strategy to implement parallelism

3.1. Implementation

Below is presented part of the implementation for the analysis of the ngrams of words:

```
#pragma omp parallel default(none) shared(histogram, start_idx, end_idx)
    {
        int numThreads = omp_get_num_threads() ;
        int threadNum = omp_get_thread_num();

        int itemsPerThread = (end_idx - start_idx + 1) / numThreads;
        std::unordered_map<std::string, int> localHistogram;

        int threadStartIdx = start_idx + threadNum * itemsPerThread;
        int threadEndIdx = threadStartIdx + itemsPerThread - 1;

        if (threadNum == numThreads - 1) {
            // The last thread may have extra items.
            threadEndIdx = end_idx -1;
        }

        for (int i = threadStartIdx; i <= threadEndIdx; i++) {
            int countW = 0;
            std::string currentNgram;

            for (int j = i; j < i + n; j++) {
                if(words[j] != "") {
                    currentNgram += words[j] + " ";
                    countW++;
                }
            }
            if(countW == n) { //aggiungo solo se n-gramma di lunghezza corretta
                localHistogram[currentNgram]++;
            }
        }

#pragma omp critical(updateHistogram)
            for (auto [ngram, count]: localHistogram) {

                histogram[ngram] += count;
            }
#pragma omp barrier
    }

    printHistogram(histogram);
}
```

While below there is the implementation for the n-grams of character:

```
#pragma omp parallel default(none) shared(start_idx, end_idx, n, text, histogram, cout)
    {
        int numThreads = omp_get_num_threads() ;
        //std::cout<<numThreads<<std::endl;
        int threadNum = omp_get_thread_num();
        int itemsPerThread = (end_idx - start_idx + 1) / numThreads;
        int threadStartIdx = start_idx + threadNum * itemsPerThread;
        int threadEndIdx = threadStartIdx + itemsPerThread - 1;

        std::unordered_map<std::string, int> localHistogram;
        if (threadNum == numThreads - 1) {
            // The last thread may have extra items.
            threadEndIdx = end_idx - 1;
        }

        for (size_t i = threadStartIdx; i <= threadEndIdx  ; i++) {
            bool valid = true;

            std::string currentNgram = text.substr(i, n);
            for (char c: currentNgram) {
                if (!std::isalnum(c)) {
                    valid = false;
                    break;
                }
            }
            if (valid) {
                localHistogram[currentNgram]++;
            }
        }

#pragma omp critical(updateHistogram)
        {
            for (const auto &pair: localHistogram) {
                histogram[pair.first] += pair.second;
            }
        }
    }
    printCharacterHistogram(histogram);
}
```

In both cases for each thread is defined a startIndex, a endIndex which will determine the amount of elements belong to him.

4. Performance Analysis

The testing phase has been carried out by executing both the sequential and the parallel algorithm implementations multiple times. Both versions share the same dataset(the txt file). The performed tests consist of multiple executions with different combinations of datasets and number of threads used.

4.1. Speedup

In order to evaluate the benefits in terms of performances between the two versions of the program, the speedup metric has been calculated. The speedup is the Sequential Execution Time divided by the Parallel Execution Time

4.2. Considerations

By fixing the number of data points and looking at the elapsed times of the two versions, it is possible to notice that the time required by the Sequential version grows much faster than the time required by the parallel one. So the input file size has a big impact on the computation. Than Speedup is expected to be lower to 1 when the

text file size is small (e.g. ¡= 5MB) because of the overhead introduced by the parallelization for the allocation and synchronization of the threads, that makes sequential version faster. While speedup could be grater than 1 as soon as text file size grows.

4.3. Results

In order to evaluate the overall performances, each one of the following combinations of clusters and data points sizes has been tested 10 times. The tests have been structured with all the combinations in between the following sets:
File Sizes (in MB) = 1,3, 19, 50, 128, 450
Number of Threads = 2, 3 , 4
Speedups are show below:

Trigrams of Characters

	2	3	4
1.3MB	1.50	1.55	1.61
19MB	1.86	1.96	2.10
50MB	1.88	2.08	2.25
128MB	1.90	2.06	2.13
450MB	1.93	2.10	2.21

Trigrams of Words

	2	3	4
1.3MB	0.79	0.75	0.66
19MB	1.22	1.30	1.40
50MB	1.60	1.66	1.70
128MB	1.75	1.95	2.10
450MB	1.86	2.13	2.47

Bigrams of Words

	2	3	4
1.3MB	0.78	0.84	0.81
19MB	1.45	1.43	1.52
50MB	1.68	1.74	1.77
128MB	1.79	2.08	2.11
450MB	1.87	2.15	2.20

Trigrams of Words

	2	3	4
1.3MB	0.79	0.75	0.66
19MB	1.22	1.30	1.40
50MB	1.60	1.66	1.70
128MB	1.75	1.95	2.10
450MB	1.86	2.13	2.47

5. Conclusion

Execution times for the analysis of character n-grams are much higher than for word ngrams. This is because the algorithm has to analyze many more elements. At the same times speedups grow up. As I expected the speedups for small files are below 1, as the size of the files increases they tend to grow. This means that for small files size, parallelism doesn't bring benefits. The worst speedups (0.66) occurs in the analysis of words trigrams in the case where the file size is the smallest and it is consistent because it is the case in which there are fewer operations to perform and the overhead of the parallelism is relevant.
The source code of the project and its related reporting are publicly available on GitHub:
https://github.com/Mayo98/BigramsTrigramsParallel