PC Project – Bigrams and Trigrams count

Cosimo Cinquilli
E-mail address
cosimo.cinquilli@stud.unifi.it

Gabriele Giannini
E-mail address
gabriele.giannini3@stud.unifi.it

Abstract

Our project implements three different solutions to solve the problems of evaluating bigrams and trigrams starting from a text.
The three approaches are: one sequential algorithm, one parallel algorithm written in C++ and parallelized using native C++ thread, and the last one is a parallel algorithm written in C++ and parallelized using OpenMP.
Also metrics evaluation and considerations about them are contained in this paper.

1. Introduction

Talking about bigrams and trigrams we can refer to them as **n-grams** to make the comprehension easier.

N-grams are a set of n characters. The purpose of our work is to calculate how much n-grams occur in a text. For example, in the case of bigrams, we can evaluate how much occurrences of the string “ab” are in the “Divina Commedia”.

2. The three different approaches

Afterwards the three approaches we decided to implements are briefly explained.

2.1. Sequential algorithm in C++

The main behaviour of this version is implemented in the funcion **ngrams**.
This function reads from the file using the function **compileNgram** in case of n-grams analysis or **compileNwords** in case of analysis on words (groups of characters separated by spaces).
It then counts the occurencies of all ngrams or nwords in the file and save them step-by-step in a **map** that contains the pair <string. int> where the string is the n-gram and the int is the number representing the total occurences of that particular string.

2.2. Parallel algorithm in C++ using native Threads

Here the entry point is in the function **splitFile.** In this function the input file is splitted by number of threads. Every thread executes the function **collect** that produces a map with its results and wait for the completion of a list of other threads to merge the resulting maps. This procedure will result in a **reduction** operation that in the end will produce the merge of all the maps produced from all threads.

2.3. Parallel algorithm in C++ using OpenMP

This algorithm create a parallel section containing two parallel for cicle. The first cicle calculate the ngrams and the second one execute the merge of all the results obtained by every single thread on its portion of the file.

3. Compilation and execution

Cmake has been used to manage the compilation of the projects’ executables and the CMakeList.txt file is included in the repository. During the development both MacOS and Linux systems has been used successfully to build and run the programs.
After the build has finished, three executables should have been created: ngrams-seq, ngrams-omp and ngrams, respectively the sequential version, the parallel version implemented using OpenMP and the parallel version implemented with C++ threads.
The executables expects to find .txt files to be analyzed, UTF-8 encoded, placed in a directory named “analyze” in the same working directory of the executable. All such files will be read and used by the programs once started.
All executables are expected to put output files in a directory called “output” that they will create if necessary. Output files have the same name as input files, but .csv extension is used. Every run overwrites the same output files.
The executables understand command line options: **-n** <integer> to pass in the number of characters in the ngrams (so 2 would be bigrams, 3 trigrams, etc.), **-w** to

switch the analysis from groups of characters to groups of words and, for the parallel versions, **-t** <integer or "hw"> to pass in the number of threads to use or "hw", to let the C++ runtime decide based on the current hardware available.
Even if arbitrarily large numbers can be passed to the **-n** option, insanely big ngrams calculations (as in 100 or more) would lead to extremely high memory usage and are thus to be avoided (moreover, they should make almost no sense).

4. Performances

Here we present and compare metrics obtained from the algorithm we wrote. All metrics shown have been collected on the same system, with -t hw option on a quadcore CPU.

4.1. BIGRAMS

4.1.1 Sequential algorithm (bigrams)

- **Completion time**: 275675µs
- **Throughput**: 12.0189 operations executed in 1/Service time.
- **Service time**: 0.0832025 mean time on any bigram analyzed.
- **Latency**: in a sequential algorith coincide with service time.

4.1.2 Parallel algorithm in C++ using native Threads (bigrams)

- **Completion time**: 50908µs
- **Throughput**: 64.8575 operations executed in 1/Service time.
- **Service time**: 0.0153648 mean time on any bigram analyzed.
- **Latency**: 0.0921891 time needed by a thread to process a bigram.
- **Scalability**:
  - throughput par/throughput seq = 5.3962
- **Speedup**: ts/tp = 5.4151
- **Efficiency**: speedup/numthreads = 0.9025

4.2. Parallel algorithm in C++ using OpenMP (bigrams)

- **Completion time**: 57160µs
- **Throughput**: 58.688 operations executed in 1/Service time.
- **Service time**: 0.0172518 mean time expired on one bigram analyzed.
- **Latency**: 0.103511 time needed by a thread to process a bigram.
- **Scalability**:
  - throughput par/throughput seq = 4.8829
- **Speedup**: ts/tp = 4.8228
- **Efficiency**:speedup/numthreads = 0.8038

4.3. TRIGRAMS

4.3.1 Sequential algorithm (trigrams)

- **Completion time**: 315501µs
- **Throughput**: 10.5017 operations executed in 1/Service time.
- **Service time**: 0.0952226 mean time on any bigram analyzed.
- **Latency**: in a sequential algorith coincide with service time.

4.3.2 Parallel algorithm in C++ using native Threads (trigrams)

- **Completion time**: 69467µs
- **Throughput**: 47.4745 operations executed in 1/Service time.
- **Service time**: 0.0209664 mean time on any bigram analyzed.
- **Latency**: 0.125798 time needed by a thread to process a bigram.
- **Scalability**:
  - throughput par/throughput seq = 4.5206
- **Speedup**: ts/tp = 4.5417
- **Efficiency**: speedup/numthreads = 0.7569

4.3.3 3.6. Parallel algorithm in C++ using OpenMP (trigrams)

- **Completion time**: 84571µs
- **Throughput**: 40.2842 operations executed in 1/Service time.
- **Service time**: 0.025525 mean time expired on one bigram analyzed.
- **Latency**: 0.15315 time needed by a thread to process a bigram.
- **Scalability**:
  - throughput par/throughput seq = 3.8359
- **Speedup**: ts/tp = 3.730
- **Efficiency**:speedup/numthreads = 0.6216

5. Conclusion

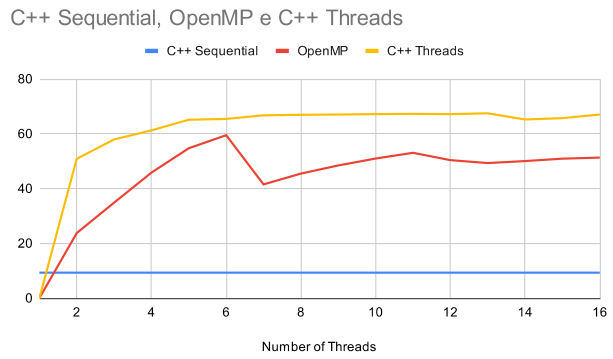

Figure 1: Bigrams throughput with the three implementations with a range of threads from 0 to 16.

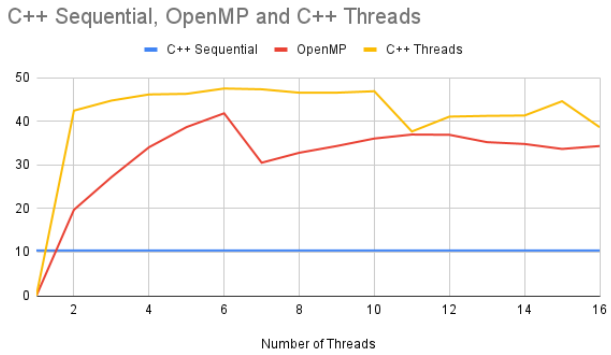

Figure 2: Trigrams throughput with the three implementations with a range of threads from 0 to 16.

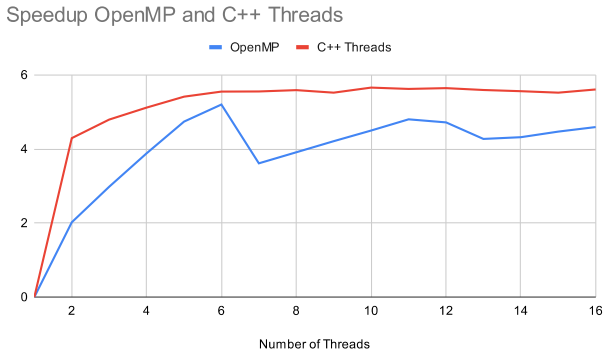

Figure 3: Bigrams Speedup of the two parallel implementations with a range of threads from 0 to 16.

In conclusion it is possible to see that the best algorithm is the on parallelized with native threads in C++.
We tried a lot to make the OpenMP version work better, trying to make the data fit better into cache or trying to move or reduce the impact of the parallel sections.
We found that some of those implementations were worse than the sequential one, and in the end we settled on something better than the first attempt in OpenMP, but not better than the one with native C++ threads.
We also found that the OpenMP version does not scale up very well on the number of threads, as it almost always works better with lower thread number than the pure C++ version.

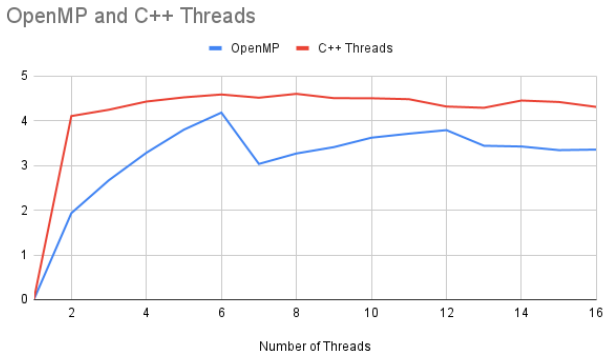

Figure 4: Trigrams Speedup of the two parallel implementations with a range of threads from 0 to 16.