PC-2017/18 Bigrams And Trigrams

Mansueto Pierluigi
E-mail address
pierluigi.mansueto@stud.unifi.it

Giuntini Andrea
E-mail address
andrea.giuntini@stud.unifi.it

1. Introduction

Bigrams and Trigrams are used in one of the most successful language models for speech recognition. They are a special case of N-grams. A N-gram is a sequence of N adjacent elements from a string of tokens, which are typically letters, syllables, or words. These methods are much useful in several cryptographic and linguistic fields, such as spelling correction and text summarization. For example, Bigram frequency can be used in cryptography to solve cryptograms or can be one approach to statistical language identification.

In our projects we have analyzed the frequency of letters related to Bigrams and Trigrams using both serial and parallel approaches. Finally, we have compared these two applications observing the execution time by varying the length of the text. For the parallelization of code, we have used `OPENMP`, an `API` for shared-memory parallel programming. This `API` is not just a library: it's a set of compiler directives (must be supported by the compiler), library routines, and environment variables.

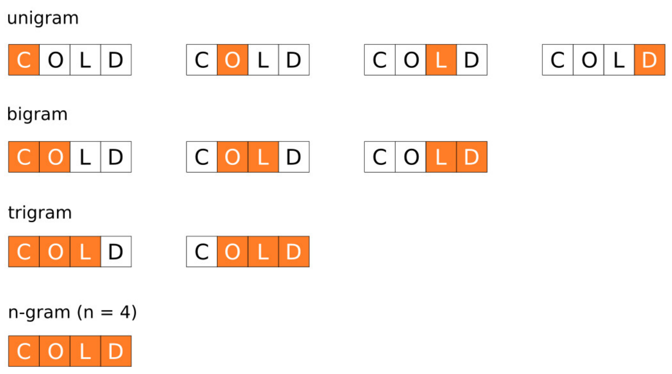

Figura 1. N-Gram example

2. About OPENMP

`OPENMP` is an `API` for shared parallel programming: it is designed for systems in which threads and processes can have access to all available memory. The strength of this `API` is the great simplification to implement concurring programming hiding details, at the expense of expressiveness and flexibility. Indeed `OPENMP` only requires to add compiler directives to construct from a serial program a parallel one. In addition to the compiler directives, there are also runtime library functions and environment variables, useful for the management of threads and processes.

`OPENMP` has an exact behavior which is derived from the fork-join model: there's a master thread that, at the beginning, executes the serial code of program; when it encounters the parallel region denoted by OPENMP's compiler directives, it creates slave threads; the master thread and the slave threads divide iterations of the parallel code and execute them concurrently; at the end of the parallel region there is an implicit barrier where each thread waits for all other ones; after this barrier, the slave threads disappear and the master one continues execution of the serial code after the parallel region.

After this little description, we will tell you how to do a program with a parallel approach whose aim is to find Bigrams and Trigrams: we must implement a serial approach of the search of Bigrams and Trigrams, find the region to be parallelized and, after this region is found, use the compiler directives to transform the region in a parallel one, following the already described fork-join model.

3. How to represent Bigrams and Trigrams

The best way found to collect all frequencies of all Bigrams is a matrix.

First, we have implemented a function, called in our program `StringToDecimal`, in order to have a number between 1-26 (26 is the number of alphabet letters) from each letter (for example letter A corresponds to the number 0, letter B to the number 1 and so on . . .). If the function has an accented letter as input, the function deals with this letter as if it were a letter without accent (for example the letter à corresponds to the number 0, as the letter A).

Thanks to this function, we have allocated a matrix 26x26 where we have collected the frequencies of Bigrams. Indeed, each cell corresponds to an exact Bigram, where row of the cell is the number related to first letter of Bi-

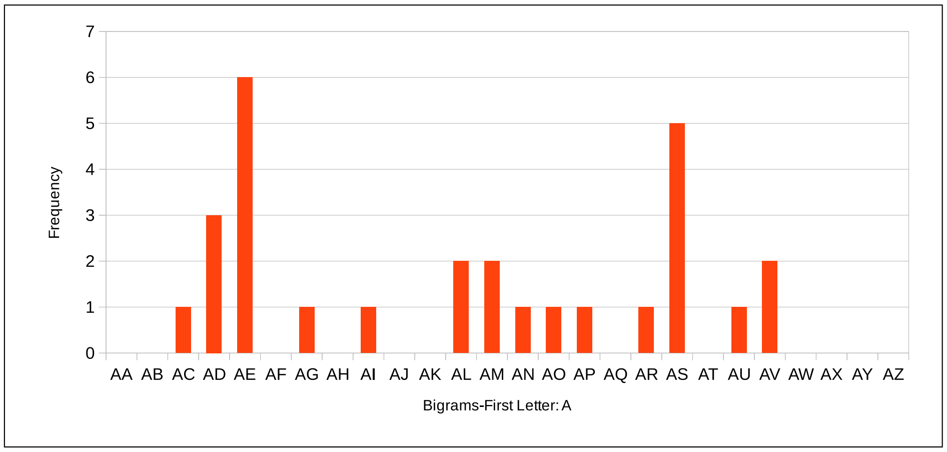

Figura 2. Bigram frequency concerning the letter ’A’

gram and column is the number related to second letter of Bigram. The matrix is an integer one: each cell has the frequency number of the Bigram to which is related.

For the Trigrams, we have used the same way as Bigrams. The only difference between this case and the previous one is the dimension of matrix. Indeed, in order to collect frequencies of all Trigrams, we have had to deal with a tri-dimensional matrix, where the first index of a specific cell is related to the first letter of a specific Trigram, the second index to the second one and the third index to the third one.

4. The serial approach

First of all, we will introduce you the serial approach to provide a better understanding of the parallel one. In the serial approach, after allocation of the matrix (either bidimensional or tri-dimensional), the program analyzes each row sequentially to find all the Bigrams/Trigrams. Each time the program successes on the search, it increases the frequency number in the cell of the matrix related to the specific Bigram or Trigram found.

To do this search, we have had to implement two for loop: the first one related to the rows of the text, that is the input of the program, and the second one related to the letters in a row. The program can consider the case in which a word starts in a row and ends in the next row: the Bigram, whose the first letter is the last one of the specified word in the considered row and the second letter is the first one of the same word in the next row, is considered in the count of the frequencies of Bigrams (the same example can be done in the search of Trigrams).

5. The parallel approach

For the reconstruction of this just described serial code into a parallel one, we have decided to parallelize the first for loop related to the rows of the input text. Indeed, to parallelize the second for, related to the letters of a row, has no sense. The best profit is obtained by assigning one thread to each row, thereby this thread is responsible to find all Bigrams/Trigrams in the assigned row. To do this parallelization, we have used typical compiler directives of `OPENMP`: `#pragma omp parallel for` and `#pragma omp atomic`.

5.1. #pragma omp parallel for

This directive indicates the beginning of the parallel region, which is the beginning of the first for loop, as just sayed. After the Master thread encounters this region and it creates Slave threads, it and the slave threads divide lines of the input text and execute them concurrently(fork). At the end of the parallel region (at the end of the input text) there is an implicit barrier, where once a thread finishes its iterations waits the end of the other threads (join).

When we have used this compile directive, we have had to declare which variable is private (a new version of the original variable, same type and size, is created in the memory of each thread belonging to the parallel region) or shared (threads can access and modify the same original variable in the shared memory). In our case the private variables are two index, used to find the second letter of a Bigram after finding the first letter (if you consider Trigram you have to add another two index to find the third letter of a Trigram), and the public variable is the bi/tri-dimensional matrix. This latter one is shared between all threads and all threads can access and modify its cells. For this reason, it can happen that two or more threads want to access or modify a same cell at the same time. This fact causes a race condition, a situation where the result is influenced by the order of the execution of threads. To avoid this drawback, we have used the atomic directive, described in the next paragraph.

5.2. #pragma omp atomic

The atomic construct executes an indivisible expression atomically. This indivisible expression can potentially be a load , store or update. In this case, when threads encounter this directive if they want to access or modify a same cell at the same time they must do this sequentially;instead if they want to access or modify different cells they must do this concurrently. So this directive avoids possible race condition in the management of a variable shared between all threads.

In our case, we have used this compiler directive to update a cell in the matrix as soon as a Bigram or Trigram is found by a thread. We have avoided to use the directive `#pragma omp critical` because, after many tests of the algorithm, we have discovered that this directive caused an increase of the computational time of the algorithm. Indeed the atomic construct instead of the critical one:

1. introduce less overhead.
2. do not enforce exclusive access to the matrix.

For this reason, an advantage of the atomic construct over the critical one is that parts of an array variable can also be specified as being atomically updated. The use of a critical construct would protect the entire array.

6. Experimental Results

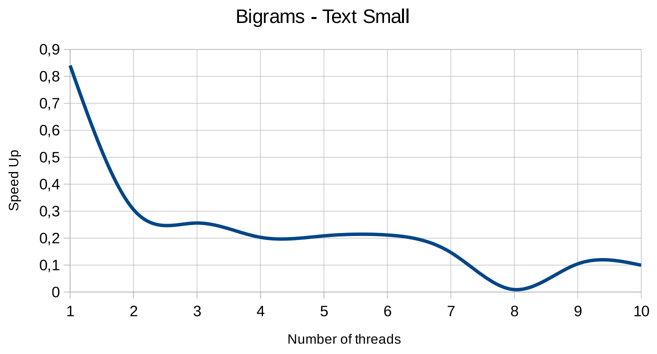

Figura 3.

In the tests of the application, we have studied the behavior of SpeedUp.

$$S_P = \frac{t_S}{t_P} \qquad (1)$$

This behavior is studied increasing the number of threads, concerning the parallel program, and the length of text. In the Figure 3 and Figure 6 the graphs refer to a 7 lines text, while the graphs in the Figure 4 and Figure 7 takes in consideration a text of 28337 rows and in the Figure 5 and Figure 8 a text of 58078 rows. For each number of threads, 5 different tests are conducted and the averages are reported.

Computer specifics:

- Intel Core i7-7700HQ 2.80GHz Octa-core
- 16 GB RAM
- 250 GB HDD

As shown in the figures, the parallel approach is less efficient in terms of computational time for the short texts; indeed, increasing the number of threads we have noticed that SpeedUp steeply decreased towards zero. So a sequential approach is preferred in this case.

Increasing the number of rows, the scenario changes completely. In the tests done in a medium length text and in a long length text SpeedUp has reached the peak, in terms of performance, when we have used a number of thread nearly equal to the number of computer's core, and then has decreased its value again , with an ondulatory behavior, as the number of threads has got larger. However, in these cases it could be possible that SpeedUp reaches a new peak greater

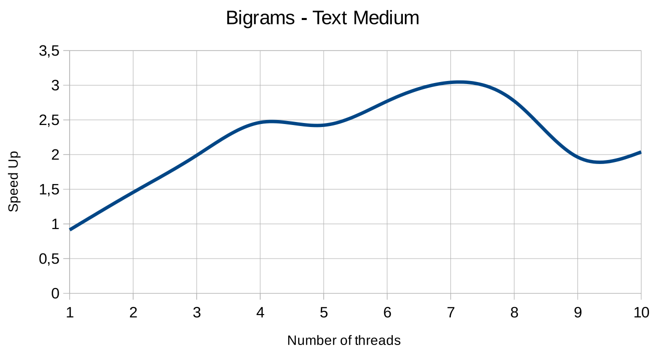

Figura 4.

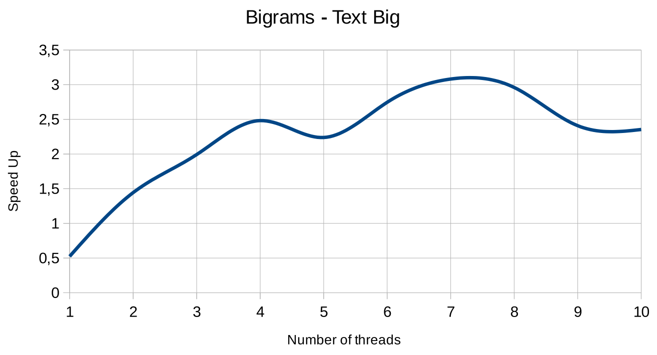

Figura 5.

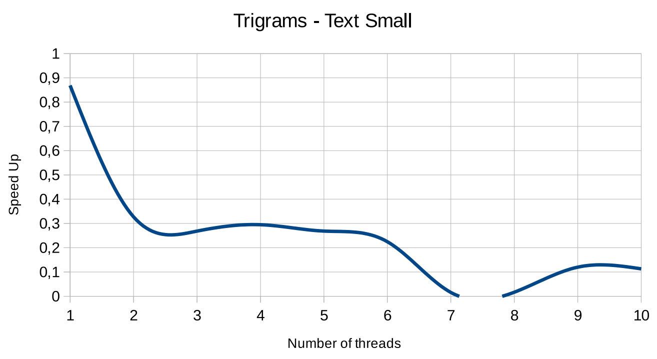

Figura 6.

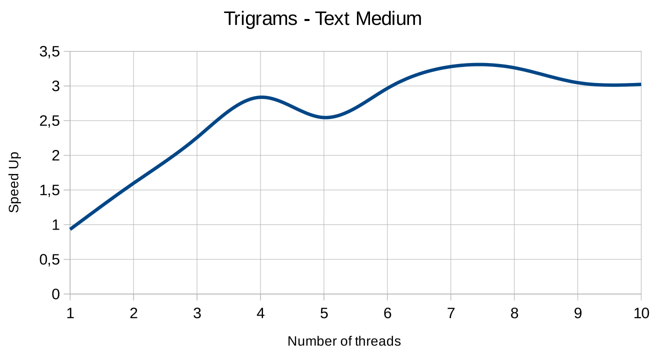

Figura 7.

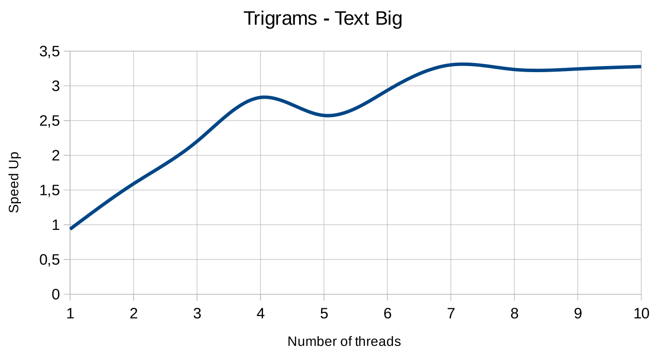

Figura 8.

than before with a number of threads higher than the number of computer's core: if it exists, this new peak usually has a value nearly the one found with the first peak. If we take in consideration a even higher number of threads, then we go back to the sequential time and, in some cases, we exceed this time.

To sum up, the sequential approach is better for short size text, while the parallel one improves in terms of execution time, as the size of text increases.

Riferimenti bibliografici

[1] M. Bertini, *8_shared_memory_openmp*, Università degli Studi di Firenze, 2017-2018

[2] M. Bertini, *9_shared_memory_openmp_directives*, Università degli Studi di Firenze, 2017-2018