PC-2017/18 Edit Distance using MPI

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

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

1. Introduction

In computational linguistics and computer science, edit distance is a way of quantifying how dissimilar two strings (e.g., words) are to one another by counting the minimum number of operations required to transform one string into the other. This algorithm is used in many important fields: the most popular examples are natural language processing, where automatic spelling correction can determinate the right corrections for a misspelled word by selecting words from a dictionary that have a low distance to the word in question,and bioinformatics, where edit distance is used to quantify the similarity between two sequences of the DNA. We have dealt with Edit Distance's Levenshtein approach, in which three operations, that are the copy, insertion and the removal of a letter, are considered. We have used this approach to create a matching word application. In detail, the user can insert, as input, one or more words which he wants to find and the application returns, as output, a list of similar or identical words to the ones inserted by him. This search can be computed in one or more texts. In this project, we have created both a serial approach and a parallel approach of this application and we have made a comparison between these two implementations. The parallel approach is implemented using `MPI`, a distributed memory programming model and the comparison is done in terms of SpeedUp, by varying the length of the texts, using instances made available by the cloud computing Amazon Web Services (`AWS`).

2. MPI

The `MPI` (Message-Passing Interface) is a standardization of a message-passing library interface specification. Indeed, `MPI` defines the syntax and semantics of library routines for standard communication patterns in a distributed memory system. This one consists of a collection of core-memory pairs connected by a network, and the memory associated with a core is directly accessible only to that core. Process (typical 1 process per processor in a machine) has thread of control and local memory with local address space and the informations (or messages) is passed between nodes using the network. You should know that in this standardization all parallelism is explicit: the programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using `MPI` constructs. Indeed, each process has his own rank which is a identifying number: the process with the rank 0 is the master; the others are the workers. So, depending on the rank, a process must do different things in parallel with the other ones. In order to implement a such mechanism, we have used the following typical instructions of `MPI`.

1. `MPI_Init`: Initializes a session of `MPI`;

2. `MPI_Finalize` - Terminates the started session of `MPI`;

3. `MPI_Comm_size` - Determines the number of processes;

4. `MPI_Comm_rank` - Determines the label of calling process;

5. `MPI_Send` - Sends a message to a destination process; in our project, this instruction is a blocking synchronous operation because the return of control to the calling process indicates that all resources, such as buffers, specified in the call can be reused for other operations (this is why it is a blocking operation) and the completion of this operation indicates not only that the send buffer can be reused, but also that the receiving process has started the execution of the corresponding receive operation (this is why it is a synchronous operation).

6. `MPI_Recv` - Receives a message from a source process. This operation is blocking too in our project.

7. `MPI_Probe` - Returns a `MPI_Status` object, very useful to know the identifying rank of a source process.

We have distinguished the project between the master's implementation and the workers' one.

3. How to save data

The similar and identical words to the ones given by the user are stored to a map, whose the key is the input word and the value is a second map. In this latter, the key is an integer, that represents the number of text in which the application find words, and the value is a third map. In this one, the key is the word found (either similar or identical) and the value is a vector of two integer, that represent the edit distance between the found word and the input one and the times in which application has found the word respectively. With this complex map, you will know, for each your input word, in which text a similar ($editDistance \neq 0$) or identical ($editDistance = 0$) word is found and how many times this one is found.

4. Master

The master must:

1. Tell to the workers which word must be found and in which line of a specific text the worker must find;
2. collect the informations given by the workers;
3. tell to the workers when they must do a last search;
4. tell to a worker if it is useful or not;

In order to do the first task, the master sends to a worker: the index related to a vector in which the input words are stored, the indexes related to a string matrix in which the rows represent the texts' number, where the application must find the words, and the columns represent the lines of the specific text. The worker tells to the master if it has found a similar or identical word. If a word is found, the master must store this in the map (see the paragraph *How to store the data*). In order to discover which worker wait to send a message, the master use the `MPI_Probe` instruction to obtain the worker's rank, which is stored in the `MPI_Status` object. If a worker must do a last search, the master sends to it an integer, called `finalIteration`: if its value is 1 there is a last search to compute, if it is 0 the search to be computed is not the last one. Finally, it can happen that there is more workers than texts' lines. In this case, some workers can be not useful and they must finish to compute. So the master sends to them `finalIteration` with 2 as value.

5. Worker

Received the informations by the master, the worker must find the similar or identical words to the input word considered in a text line. In order to do this, it catches each word in the line and stores it in a vector. After, it computes edit distance between each word in this vector and the given input word: if the edit distance is less than a tolerance (that is chosen by the user) then the worker tells to the master that it has found a word; otherwise, the worker continues its computation without telling anything to the master. Furthermore, the worker must tell to the master if it has finished to do the search in a given line. For this reason, it sends an integer called `response`: if its value is 2 then the worker has finished, ready to start a new search of an input word in a new line of a text. In addition, this integer is used by the worker to tell if it has found a word or not. Indeed, if its value is 0 the worker has not found a word; otherwise, if it is 1, it has found a word.

6. Experimental Results

We analyzed the behaviour of the SpeedUp, which aim is to compair the computational time between the sequential and the parallel approaches :

$$S_P = \frac{t_S}{t_P} \tag{1}$$

We used Amazon Web Services (`AWS`) to implement a reliable cloud computing service, executing the program code in several machines. To implement this process, we had to initialize some instances, based on the computer specifics that we decided to take in consideration; in particular, the following specifics :

- Ram : 61 GB
- CPU : 8 cores

The results are shown in the table 6, studying the behaviour of Edit Distance using two different text sizes (Small text: 70000 lines; Large text: 250000 lines).

Cores	Small text		Large text	
	Time	SpeedUp	Time	SpeedUp
5	52.41	0.31	192.03	0.33
10	27.07	0.61	97.97	0.64
15	20.59	0.80	74.70	0.84
20	16.02	1.03	61.08	1.03
25	17.29	0.96	64.99	0.97
30	21.11	0.78	72.31	0.87
35	19.69	0.84	71.94	0.87
40	20.56	0.80	79.01	0.80

Text size	Time
Small text	16.6
Large text	63.1

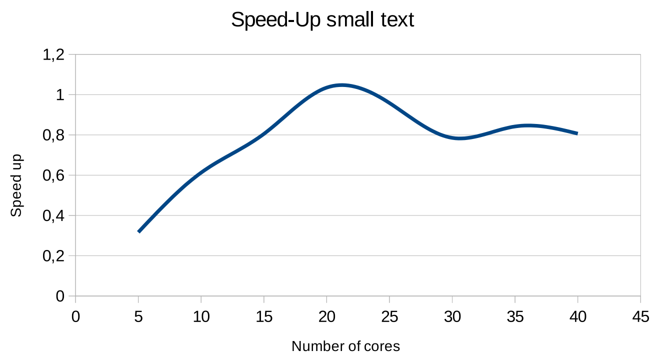

Figura 1.

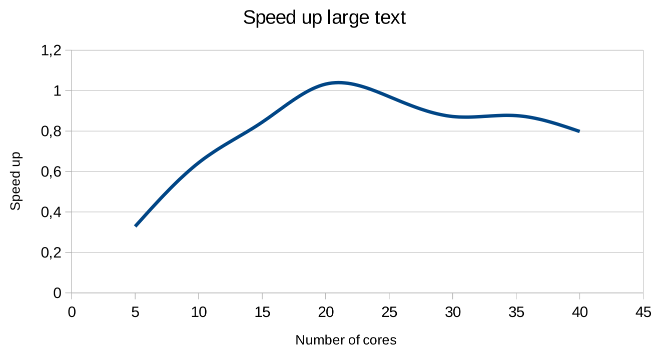

Figura 2.

We used five instances, because there are several limitation for the number of instances to use. Anyway, the results show how the executional time gets lower as we reach the number of 5 cores used per machine. The SpeedUp doesn't increase that much, due to the number of instances used. We could reach better performances fot the parallel code with a greater number of machine, but we can already notice a little improvement for 20 processes. Both small and large text are pretty similar, which reflects the fact that with more instances we could get better results in terms of SpeedUp, especially because we are using a distributed memory system.

Riferimenti bibliografici

[1] M. Bertini, *21_distributed_memory_mpi.pdf*, Università degli Studi di Firenze, 2017-2018

[2] M. Bertini, *22_AWS_Hadoop_MPI.pdf*, Università degli Studi di Firenze, 2017-2018