Levenshtein distance: parallel implementation in OpenMP and C++ threads

Nicola Carkaxhija
Dept. of Information Engineering, University of Florence
Via di S. Marta 3, 50139 – Florence, Italy
nicola.carkaxhija@stud.unifi.it

Abstract

Edit distance finds application in several fields, such as spell checking, information retrieval, spam filters, plagiarism detection and DNA analysis. The classic approach is not able to deal with long data in reasonable time, so that new techniques are required. In the experiment here described it is proposed the use of parallel computing employing OpenMP and C++ threads, whose results are measured in terms of speed-up.

1. Introduction

Given two strings over the same alphabet, the edit distance is a measure of how dissimilar they are. More formally, it is the minimum weight sum of changes that should be applied to one string in order to transform it into the other. The valid operations consist in insertion, deletion and substitution of a single character; when they have unitary cost the edit distance is also referred as Levenshtein distance.

Since brute force algorithms are infeasible, the distance is generally computed making use of dynamic programming, that involves the definition of a distance matrix having as dimensions the length of the two strings, where the entries represent the solution of sub-problems - i.e. the distance between two prefix sub-strings – and are computed filling the matrix row by row, from left to right according to the rule defined in the else clause of the pseudo-code depicted in Figure 1.

However, for long strings also this solution results computationally demanding and a different approach becomes necessary. In the following sections a dependencies analysis is carried out in order to establish how the code may be parallelized, and beyond the programming side, two solutions with different structure are presented: the diagonal and tile approaches.

```
int LevenshteinDistance ( char str1[ 1..lenStr1 ],
                          char str2[ 1..lenStr2 ] )

   int M, N := len(str1), len(str2)
   int d[ 0..M, 0..N ]

   for i from 0 to M
       d[ i, 0 ] := i
   for j from 0 to N
       d[ 0, j ] := j

   for i from 1 to M
       for j from 1 to N
           if str1[ i - 1 ] = str2[ j - 1 ] then
               d[ i, j ] := d[ i-1, j-1  ]
           else
               d[ i, j ] := minimum(
                                // inserimento
                                d[ i - 1, j     ] + 1,
                                // cancellazione
                                d[ i    , j - 1 ] + 1,
                                // sostituzione
                                d[ i - 1, j - 1 ] + 1)
   return d[ M, N ]
```

Figure 1: pseudo-code of a sequential version of Levenshtein distance.

2. Dependence analysis

As shown in Figure 2, from the graphic representation of the relations involved in the if-else clause formulas of the algorithm, it may be observed that each entry of the distance matrix depends only by the values of the upper, upper-left and left neighbors, then it can be computed just after them, and conversely may be computed as soon as they become available. As a consequence, entries of an anti-diagonal of the matrix can be evaluated independently, provided that previous diagonals have already been computed. This source of parallelism is exploited by implementing multi-threaded algorithms to compute the distance between very large strings.

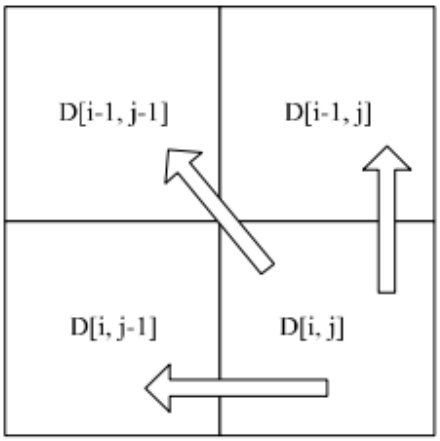

Figure 2: illustration of the distance matrix dependencies.

3. Parallelization, diagonal approach

A priori, it could be supposed that edit distance is not as well suited for parallelization as other common tasks: despite it is possible, because of the strictly binding dependencies the room for improvements is limited.

The parallelization process involves a drastic change in the algorithm structure: the anti-diagonals are computed serially, by increasing order, and for each of them the workload is split among a number of threads, i.e. the cells of the diagonal are computed independently.

Because the diagonals of a matrix exhibit non-uniform lengths, first growing and then diminishing as the matrix is traversed, computation is made more complicated: indeed, such an implementation does no less work than the classic version since the number of matrix entries is the same, but it requires more work due to the overhead of managing complicated indices. Furthermore, at the end of each diagonal, a synchronization barrier is required due to the diagonal dependence constraint.

4. First results

This approach has been implemented only in OpenMP, for which the synchronization resulted to be quite binding; this can be motivated as follows: the number of diagonals for a $M \times N$ table is $M + N - 1$, i.e. $2N - 1$ if for a matter of simplicity a squared matrix is considered, for which it could be stated that the number of barriers required is also of the order of N, so that it grows together with the dimension of the instance and brakes improvements.

Since such an implementation has to access cells from different rows and columns at each iteration, the high rate of cache misses is heavily compromising, making hard to attain a noticeable speed-up for most of the instances. For the same reason, slightly higher improvements are reached when working on strings of significantly different lengths, since in this case the cache memory is better exploited. However, for an excessive increase of the length ratio, the performances get closer to the ones of the sequential version, so that it is hard to formalize the behaviour of this approach.

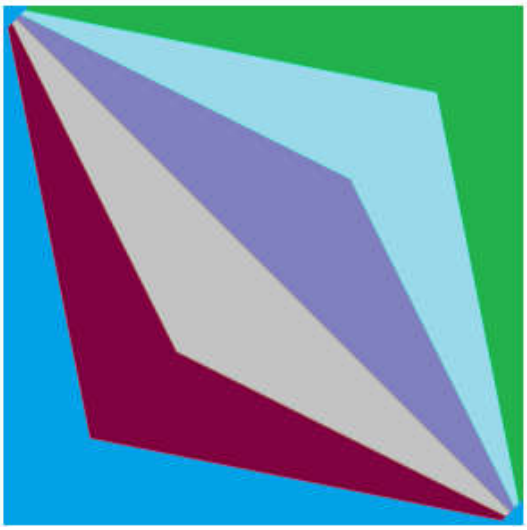

Figure 3: each color represents the portion of the distance matrix computed by a same thread: diagonals are split in equal slices, and threads always work on the same nth part according to its own id.

5. Parallelization, tile approach

The previous considerations suggest to introduce some changes, to alleviate and make less limiting the synchronization and caching problems.

An idea is to relax the constraint by considering tiles instead of single cells: by doing so the use of the cache memory is optimized, that is the hit rate is higher since each thread works on a spatially better distributed set of matrix entries. This may be thought as a generalization of the previous approach, involves a more coarse and then less onerous computation.

6. Final results

The sequential and parallel algorithms have been compared on strings randomly generated according to a uniform distribution. This phase is not taken into account in the execution time: it would be meaningless since in a realistic scenario the strings are given.

The second approach, since more promising, has been implemented both with OpenMP and C++ threads. The barrier, that is straightforward to set with OpenMP, in the case of C++ threads requires to be explicitly implemented; it has been used a simplified implementation of the class boost::barrier, whose code is available at [1].

The results were produced running the algorithms on an Intel 4-Core i5-6500 by varying string lengths, tile size and number of threads. For a sake of simplicity the charts refer to strings of the same length. The results are shown in Figures 4, 5, 6 and 7.

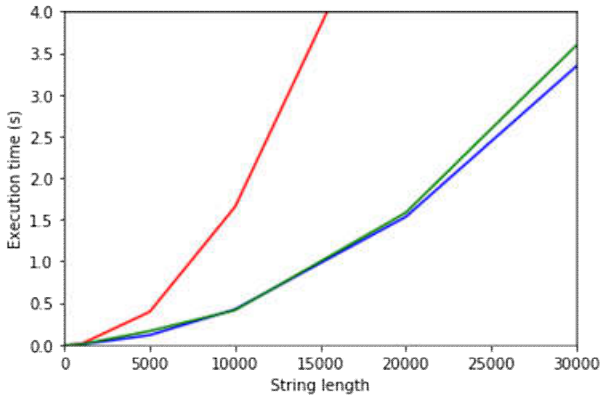

Figure 4: the red, blue and green lines refer to the execution time and string length relations of the sequential, OpenMP and C++ threads versions respectively.

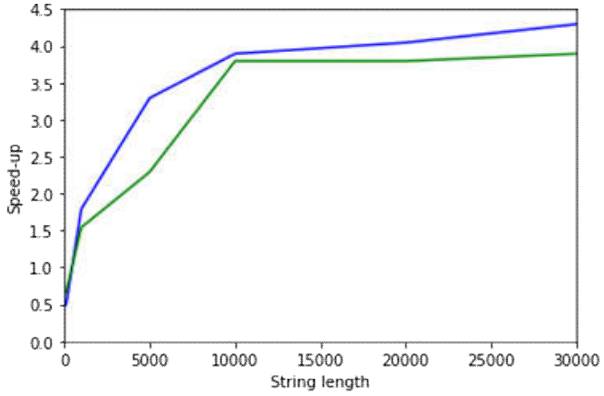

Figure 5: the blue and green lines refer to the speed-up and string length relations of the OpenMP and C++ threads versions respectively.

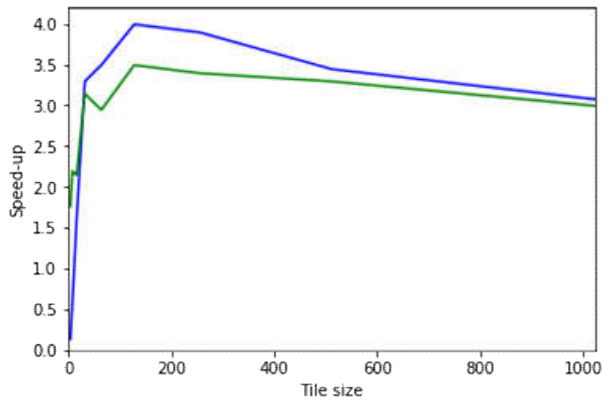

Figure 6: the blue and green lines refer to the speed-up and tile size relations of the OpenMP and C++ threads versions respectively.

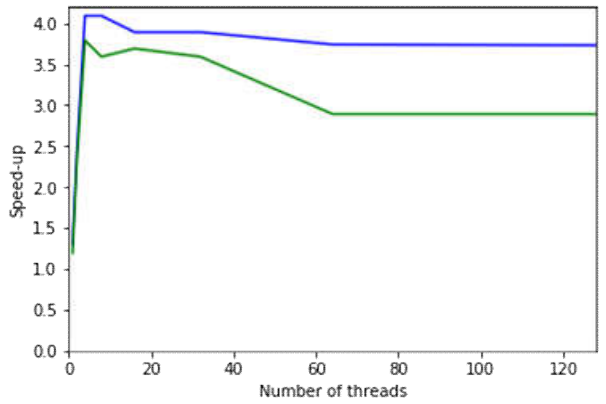

Figure 7: the blue and green lines refer to the speed-up and number of threads relations of the OpenMP and C++ threads versions respectively.

7. Conclusions

Because of its rigidity, i.e. indexes overhead, synchronization and cache problems, the diagonal version struggles to achieve a valuable speed-up, whilst the tiled algorithm outperforms it by means of a coarser approach.

The most remarkable consideration about the two tiled versions is that, for equal results, OpenMP requires less coding.

The choice of the number of threads is not critical, as long as all the processors are exploited. This involves that the other plots are independent by this value and would not be too different; hence, it can be stated without need to compute further charts that for sufficient numerous points (around 10.000) and tile size between 128 and 1024, there is at least a x4 speed-up factor.

References

[1] boost::barrier code:

https://www.daniweb.com/programming/software-development/threads/498822/c-11-thread-equivalent-of-pthread-barrier