PC-2018/19 Course Project: Parallel Levenshtein distance in C++11

Federico Vaccaro
E-mail address
firstauthor@i1.org

Giuseppe Palazzolo
E-mail address
secondauthor@i2.org

Abstract

The Levenshtein distance between two string is a common distance metric. A simple way to compute the edit distance between two given strings, is the Wagner-Fischer algorithm, which translate a recursive procedure to an iterative one through the dynamic programming. This procedure is sequential by its definition, but it can be slighty modified to take advantage of the modern multicore CPUs.

1. Introduction

Given two string x and y with length respectively $|x|$ and $|y|$, the Levenshtein distance between them is the minimum number of *elementary edits* necessary for transform the string y to the string x [4]. The elementary edits are:

- **Substitutions** of a character of y with one of x;
- **Deletions** of a character;
- **Insertions** of a character;

This cost, is summarized by this recursive definition:

$$lev_{x,y}(i,j) \begin{cases} max(i,j) \text{ if } min(i,j) = 0 \\ min \begin{cases} lev_{x,y}(i-1,j)+1 \\ lev_{x,y}(i,j-1)+1 \\ lev_{x,yb}(i-1,j-1) + \mathbf{1}_{(x_i \neq y_i)} \end{cases} \end{cases} \quad (1)$$

where $\mathbf{1}_{(x_i \neq y_i)}$ is the indicator function, equal to 1 if character $x_i \neq y_i$. The distance between x and y is $lev_{x,y}(|x|,|y|)$. Implementing this algorithm through the definition is inefficient, Wagner and Fischer proposed in 1974 [3] an algorithm (**Algorithm 1**) which make use of the dynamic programming, in order to avoid an exaggerated number of function call.

The Wagner-Fischer algorithm computes edit distance in a bottom-up method, based on the observation that if we reserve a matrix to hold the edit distances between all prefixes of the first string and all prefixes of the second, then we can compute the new values in the matrix reusing the old values through definition **(1)**, and thus find the distance between the two full strings as the last value computed.
The **Algorithm 1**, assuming that $n = |x| \cong |y|$, has $O(n^2)$ time complexity and $O(n^2)$ space complexity.

Algorithm 1 Wegner-Fischer algorithm

```
procedure LEVENSHTEINDISTANCE(x, y)
    D ← Matrix(|x| + 1, |y| + 1)
    for i = 0...|x| do
        D[i][0] ← i
    for j = 0...|y| do
        D[0][j] ← j

    for i = 1...|x| do
        for j = 1...|y| do
            if x_{i-1} ≠ y_{j-1} then
                D[i][j] = min{
D[i − 1][j],
D[i][j − 1],
D[i − 1][j − 1]} + 1
            else
                D[i][j] = D[i − 1][j − 1]
    return D[|x|, |y|]
```

		k	i	t	t	e	n
	0	1	2	3	4	5	6
s	1	1	2	3	4	5	6
i	2	2	1	2	3	4	5
t	3	3	2	1	2	3	4
t	4	4	3	2	1	2	3
i	5	5	4	3	2	2	3
n	6	6	5	4	3	3	2
g	7	7	6	5	4	4	3

		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

Figure 1. How the matrix D is filled [4]

1.1. Parallelization of the algorithm

Clearly the **Algorithm 1** is sequential, because at each step j of the inner loop, we might need to read $D[i][j-1]$, which has been computed at the step $j-1$. In general, to fill the element (i, j) of the matrix, we need to read the elements with index $(i-1, j)$, $(i, j-1)$, $(i-1, j-1)$, as shown in **Figure 2**. From a different perspective, when the first element of the matrix (*i.e.* $D[1][1]$) has been computed, we're able to compute both $(2, 1)$ and $(1, 2)$ in parallel, because there is no depandancy between what must be *read from* and *written to* (in order to fill a value), and each element **is written just once**. Then, we can process the indices $(3, 1)$, $(2, 2)$ and $(1, 3)$, still in parallel, and so on.
We identify these sets of *the elements that can be parallel computed*, with the **skew diagonals** of D. In the **Figure 2**, this idea is schematized.

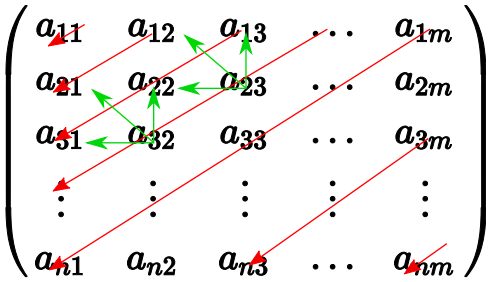

Figure 2. The red arrows are the skew diagonals of a matrix. Each element on a red arrow just need to access to the elements, following the pattern represented by the green arrows *i.e.* they just need to read previously computed elements.

A matrix of $M \times N$ shape, contains $M + N - 1$ skew diagonals. From here, we'll refer to $N = |x| + 1$ and $M = |y| + 1$. We can fill the matrix computing in parallel the indices belonging to each diagonal. Of course, is not possible to compute each skew diagonals in parallel but must be ensured that the order is followed: the processing must start from the first diagonal containing $a_{1,1}$ and must end with the last diagonal, containing a_{nm}.
The algorithm rewritten with the indices following each skew diagonal is reported below in the **Algorithm 2**. Its main inner loop, can be entirely computed in parallel. Between each iteration of the outer loop, **we need a form of synchronization** to avoid *race conditions*.

Algorithm 2 modified Wegner-Fischer

procedure MODIFIEDLEVENSHTEIN(x, y)
$D \leftarrow Matrix(|x|+1, |y|+1)$
$N \leftarrow |x|+1$
$M \leftarrow |y|+1$
for $i = 0...N-1$ **do**
$D[i][0] \leftarrow i$
for $j = 0...M-1$ **do**
$D[0][j] \leftarrow j$

$dMin = 1 - M$
$dMax = N$

for $d = dMin...dMax-1$ **do**
$iMin = max(d, 1)$
$iMax = min(M+d, N-1)$
for $i = iMin...iMax-1$ **do** ▷ Do this in parallel for each i
$j \leftarrow M + d - i + 1$
if $x_{i-1} \neq y_{j-1}$ **then**
$D[i][j] = min\{$
$D[i-1][j]$,
$D[i][j-1]$,
$D[i-1][j-1]\} + 1$
else
$D[i][j] = D[i-1][j-1]$
return $D[|x|, |y|]$

2. The implementation

The algorithm has been implemented on a shared memory system, using the C++11 `<thread>`, `<mutex>` and `<atomic>` from the standard library [1].
Before discussing the implementation, must be pointed out that actually the previous approach yields *really bad performance*, about 1000 times slower the single threaded version. The main issue is that the **Algorithm 2** has an irregular memory access pattern in terms of temporality and spaziality (the matrix is stored as a linear array); therefor, no thread is able to take advantage properly of optimizations like *RAM burst* or caching [2].
Fortunately, if we subdivide the main matrix into sub-matrices (or *tiles*), it's possible to reuse the previous idea and assign to each thread one or more sub-matrices, still belonging to a skew diagonal; *e.g.* a 4×4 matrix composed by 2×2 sub-matrices:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

then we can handle each sub-matrix as a simple element a_{ij}.
It works because to fill the elements along the first row or first column (of a certain sub matrix) requires just to read the values from a previously computed tile.
We propose two variant of the implementation, differing for the used synchronization mechanism.

2.1. Lock-based variant

The first one is the (Spin) lock-based variant. we first initialize each `std::thread` and the D matrix. Then, after defining a $TILE\_WIDTH$ (*i.e.* the side of the sub-matrices) we construct an array of `std::atomic_bool` of $\lceil \frac{N}{TILE\_WIDTH} \rceil \times \lceil \frac{M}{TILE\_WIDTH} \rceil$ dimension and we initialize each element to `false`. Each variable of this array is assigned to the corresponding sub-matrix. These values represent if a sub matrix has been computed *(true)* or not *(false)*. Then, we instantiate a simple `ConcurrentQueue` (made up with `std::mutex`, `std::conditional_variable` and `std::queue`). This queue is filled by the main thread with the indices of the sub-matrices and just then (to avoid contention) the threads are started. Each thread do these operations until it gets a *poison pill* (marked with $(i, j) = (-1, -1)$):

1. Get an index (i, j) for the SharedQueue (then the scheduling is *dynamic*);
2. Check if the sub-matrices $(i-1, j)$, $(i, j-1)$, $(i-1, j-1)$ have been computed looking on the atomics array, otherwise wait (*spin lock*);
3. Fill the (i, j) sub-matrix;
4. Set to `true` its index on the atomics array;
5. go to Step 1;

It's important to notice that the indices are enqueued following the order proposed by the **Algorithm 2**; otherwise it decrease the level of parallelism.
Before the main thread return the edit distance, it waits at a thread barrier until the other threads complete their work and reach that.

2.2. Barrier-based variant

The barrier-based variant is basically the **Algorithm 2** with tiles instead of single elements, but at the end of the inner loop there is a barrier. The needed synchronization is simpler but less grainer, because the threads don't wait once the index as been assigned. The disadvantage of this variant is that when some threads finished to fill a diagonal but others don't, these threads are idling at the barrier instead of start the computing of the first submatrices of the next diagonal. However, the two variants perform pretty the same, with the first variant resulting slighty faster.

3. Experimental results

Various experiments have been made to compare first of all the SpeedUp of the parallel implementation respect to the sequential version, written in C++. Also, has been analyzed the scaling of the algorithm respect to the dataset dimension and the performance gain using different tile dimension. The experiments have been conducted on Ubuntu 16.04 LTS, on a machine equipped with

- Intel Core i7 8700k @ 4ghz 6 core /12 threads processor
- 16 GB DDR4 RAM

The two compared strings have been random generated and are composed by characters in the set $\{A, T, C, G\}$, to mimic a DNA string. First the x and y strings are generated, then the y string characters are changed with a 5% probability, in order to produce an effective edit distance. Also, was a reasonable choice working with *16 bit unsigned integers*, to test bigger strings without exceed system memory and save bandwidth.

The performance is evaluated in terms of **SpeedUp** varying of the string length; also will analyzed the thread scaling for a fixed length; lastly, how varying the tile width can cope with the case of shorter strings. The speed up is computed as:

$$SpeedUp = \frac{t_s}{t_p} \tag{2}$$

Where t_s is the time of execution of the sequential implementation, and t_p the time of the parallel one. These times are the average of the results producted by 20 tests.

Str Length	ST Time	MT Time	$SpeedUp$
1000	0.0014s	0.0014s	**0.84**
5000	0.066s	0.02s	**3.3**
10000	0.27s	0.05s	**5.4**
20000	1.11s	0.016s	**6.93**
30000	2.50s	0.35s	**7.14**
40000	4.45s	0.62s	**7.17**
50000	6.96s	0.96s	**7.25**

Table 1. Times measured and SpeedUp obtained by the parallel implementation, varying the dataset dimension.

The results from the **Table 1** have been produced by measuring the lock-based variant with 12 active threads plus the main thread and $TILE\_WIDTH = 512$. We can notice that increasing the data size brings to an increase of the **efficiency** of each processor. When considering shorter strings we try to reduce the tile width to increase the number of parallel working threads. In fact, when $StrLength = 1000$ and $TILE\_WIDTH = 512$ there are at most 2 working threads. The **Figure 2** shows how reducing the tile width improves the performance even on short strings comparison. Also it's possible to notice how the

SpeedUp dramatically decrease when the tile width tends to zero (*i.e.* the single element), as said before.

Figure 3. SpeedUp varying the tile width on short strings.

As previously said, the lock-based variant is faster than the barrier-based, as shown in the **Table 2**.

Str Length	Lock-based	Barrier-based
1000	**0.84**	0.77
10000	**5.4**	4.5
20000	**6.93**	5.84
50000	**7.25**	6.54

Table 2. SpeedUp comparison between the two variants.

The **Figure 4** shows how the SpeedUp vary increasing the thread number when the string length is 50000.

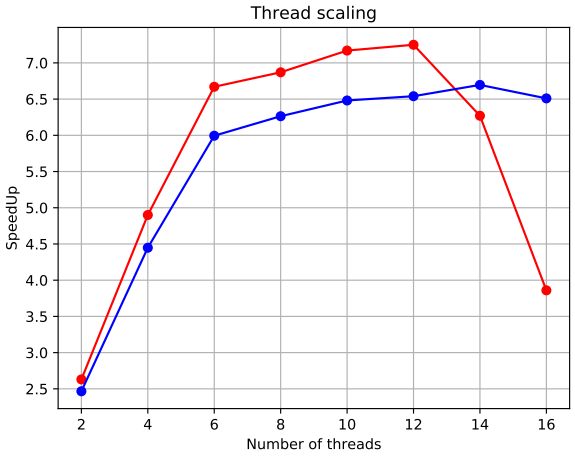

Figure 4. Speed-Up factor varying the number of threads. The red plot refers to the lock-based implementation, the blue to the barrier-based one.

It's interesting to notice that the speed-up is *superlinear*, since the core count is six. When the threads number is less-equal than the number of physical cores, the speed up curve is steeper, so the implementation take advantage (super)linearly of each core; the employed CPU has a fairly large multilevel caches, so it produces positive effects; once passed the number of physical core, the curve is flatter but does it shows slight benefits thanks to the Hyper Threading technology. When the threads number exceed the logical threads manageable by the CPU without doing context switch, the two variant behaves differently:

- the lock based variant decrease fastly its SpeedUp due to the spin lock, and halves the performance at 16 active threads: it happens that there may be threads which occupy logical cores, "stealing" resources from the threads, which actually are ready to compute.
- the second is more robust, keeping the SpeedUp approximately constant, since there is no thread spinning.

3.1. Conclusions

The algorithm for computing the Levenshtein edit distance, despite it is natively sequential, can be modified and be executed in parallel if the matrix is filled proceeding by its sub-matrices along the skew diagonals. When comparing short strings, the implementation doesn't have a properly SpeedUp and even if it can be slighty improved by tweaking with the tile width, it is clearly not "game-changer". However, a multicore implementation should more interesting on larger datas, and here it is where our implementation is more effective and is able to leverage the modern CPUs, showing even superlinear SpeedUp on longer strings.

3.2. Resources

Link to the C++11 implementation on GitHub https://github.com/fede-vaccaro/ParallelLevenshtein-CPP

References

[1] C++ standard library headers. https://en.cppreference.com/w/cpp/header, 2018. [Online; accessed 04-02-2019].

[2] T. Rauber and G. Rnger. *Parallel Programming: For Multicore and Cluster Systems*. Springer Publishing Company, Incorporated, 2nd edition, 2013.

[3] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, Jan. 1974.

[4] Wikipedia. Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance, 2019. [Online; accessed 04-02-2019].