This paper presents a method for the parallelization of the Levenshtein distance algorithm deployed on very large strings. The proposed approach was accomplished using .NET Framework 4.0 technology with a specific implementation of threads using the System.Threading.Task namespace library. The algorithms developed in this study were tested on a high performance machine using Xamarin Mono (for Linux RedHat/Fedora OS). The computational results demonstrate a high level of efficiency of the proposed parallelization procedure.

Keywords: Levenshtein distance, Levenshtein-Damerau distance, edit distance, very large strings, parallel computing, threads, high performance computing, Microsoft .NET Framework 4.0, mono-project

Streszczenie


Słowa kluczowe: odległość edycyjna Levenshteina oraz Levenshteina-Damerau, odległość edycyjna, duże ciągi tekstowe, obliczenia równoległe, wątki, obliczenia wysokiej wydajności, Microsoft .NET Framework 4.0, mono-project

* Institute of Computer Science, Faculty of Physics, Mathematics and Computer Science of Cracow University of Technology; aniewiarowski@pk.edu.pl, mareks@riad.pk.edu.pl.
1. Introduction

The Levenshtein distance [6] between two strings of characters is equal to the minimum number of insertions, deletions and substitutions of characters required to convert one string into the second string. The Levenshtein distance has applications in many areas, e.g. text analysis (detection of plagiarism) [3, 13], spell-checking in text processors [7], web mining (search engine robots) [9, 10], bioinformatics (Levenshtein-Damerau distance for DNA structure analysis [8, 11]), etc.

The algorithm for Levenshtein distance calculation creates a matrix (Levenshtein matrix) where its last element (Fig. 1) constitutes a solution. The asymptotic computational complexity of the algorithm assumes the order $O(NM)$, where $N$ and $M$ denote the lengths of the text strings (i.e. the number of characters in each strings).

Difficulties occur when long strings have to be analyzed (e.g. millions of characters in one string). In such cases, the Levenshtein matrix is complex, and to attain the final results, more time is required. Moreover, it is more complicated to allocate such large matrices in standard development environments. The problem of analysis of very long strings may occur when, for example, the same fragments of a book (whole terms or words instead of consecutive characters in the original algorithm [2, 12]) have to be compared or when DNA chains (Levenshtein-Damerau distance [8, 11]) are analyzed.

In this experiment, it was decided to design both the Levenshtein as well as the Levenshtein-Damerau algorithms with Microsoft .NET Framework [14] and run developed applications under Linux OS with the use of Xamarin Mono Project [5]. Such project environments allowed for an additional validation of efficiency and of speed of the proposed algorithms.

2. Description of the Levenshtein distance algorithm

The Levenshtein distance $K$ for two strings is the minimum number of operations – insertion, deletion and substitution required to convert one term (string) into the other. The Levenshtein distance $K$ is equal to the $d[M, N]$ element of the so-called Levenshtein matrix $d$:

$$K = d[M, N] = \text{LevenshteinDistance}\ (\text{String1}, \ \text{String2})$$

The main idea of the Levenshtein distance algorithm ($\text{LevenshteinDistance}$ function) is described by the following pseudo-code:

```plaintext
input variables: char Text1[0..M-1], char Text2[0..N-1]
declare: 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 substring of Text1 at (i - 1) = substring of Text2 at (j - 1) then

```
\[ \text{cost} := 0 \text{ else cost} := 1 \]
\[ d[i, j] := \begin{cases} 
\text{Minimum}(d[i - 1, j] + 1, \\
\text{Minimum}(d[i - 1, j - 1] + 1, \\
\text{Minimum}(d[i - 1, j - 1] + \text{cost})) 
\end{cases} \]
end for (variable j)
end for (variable i)

return \( d[M, N] \);

where:
\( d \) – Levenshtein matrix of the size \( N+1, M+1 \), formed for two terms: \( \text{Text1} \) and \( \text{Text2} \),
\( M, N \) – lengths of two terms respectively,
\( d[i,j] = (i,j) \) – element of Levenshtein matrix \( d \),
\( \text{Minimum} \) – a function to calculate a minimum of three variables,
\( \text{cost} \) – variable that gets values either 0 or 1.

The deference between the Levenshtein and the Levenshtein-Damerau distance algorithms is shown below as a part of the relevant pseudo-code with the definition of elements of the Levenshtein-Damerau matrix \( z \):

\[ z[i, j] := \begin{cases} 
\text{Minimum}(d[i - 1, j] + 1, \\
z[i, j - 1] + 1, \\
z[i - 1, j - 1] + \text{cost}) 
\end{cases} \]
if \( i > 1 \) and \( j > 1 \) and substring of \( \text{Text1} \) at \((i-1)\) = substring of \( \text{Text2} \) at \((j-2)\) and substring of \( \text{Text1} \) at \((i-2)\) = substring of \( \text{Text2} \)\((j-1)\) then
\[ z[i, j] := \text{Minimum}(z[i, j], z[i - 2, j - 2] + \text{cost}) \]

The Levenshtein-Damerau distance \( D \) is the minimum number of operations (insertion, deletion, substitution) required to change one term into the other, this is similar to the standard Levenshtein procedure, but additionally, it is necessary to account for the number of transpositions of neighboring characters. Consequently, the Levenshtein-Damerau distance between two sequences \( D \) is equal to the \( z[M, N] \) element of the suitable Levenshtein-Damerau matrix \( z \):

\[ D = z[M, N] = \text{LevensteinDamerauDistance} (\text{Text1}, \text{Text2}) \]

The figures below and the pseudo-codes above show that the value of element \([i,j]\) of matrix \( d \) in the current iteration is calculated based on the values: \( d[i - 1, j] \), \( d[i, j - 1] \) and \( d[i - 1, j - 1] \) for the Levenshtein distance and additionally, \( z[i - 2, j - 2] \) for the Levenshtein-Damerau distance. This means that each of these values must be calculated in the previous iterations of the algorithm.
Fig. 1. Levenshtein matrix $d$, constructed for terms: yesterday and tomorrow

Fig. 2. Levenshtein-Damerau matrix $z$, constructed for deoxyribonucleic acid (DNA) sequences: TCCAATA and GGCCTCC (where: T – thymine, A – adenine, G – guanine, C – cytosine)

<table>
<thead>
<tr>
<th>No.</th>
<th>string 1</th>
<th>string 2</th>
<th>Levenshtein distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Car</td>
<td>Cars</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>University</td>
<td>Universities</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>Tom is writing a letter</td>
<td>Tom is writing letters</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1 shows an example of the results of calculations of the Levenshtein distances using the conventional algorithm. In the first example, we need to add one character in string1 or remove one character in string2 to transform one string into the other. In the second example, we need to substitute one character and add two characters (string1) or substitute

\[1\] The application for both Levenshtein and Levenshtein-Damerau matrices calculations is available from the web site: www.pk.edu.pl/~aniewiarowski/publ/levenMatrix.exe.
one character and remove two characters (string2). In the last example, we need to remove four characters (‘g’, ‘a’, ‘s’ and space) in string1 or add four characters in string2.

3. Numerical implementation of parallelization algorithm

As was depicted above, difficulties occur when strings of millions of characters have to be analyzed. In such cases, the Levenshtein matrix becomes very large and more time is required to compute all its elements. Table 2 shows examples of time consumption requirements in the case of complex Levenshtein matrices calculation (without any parallelization procedure implemented). The whole series of experiments was performed on a computer with parameters: 32GB RAM; two physical processors (Intel(R) Xeon(R) CPU E5-2620 0 @ 2.00GHz, 24 threads).

**Table 2**

<table>
<thead>
<tr>
<th>No.</th>
<th>Length of string 1</th>
<th>Length of string 2</th>
<th>Number of elements of Levenshtein matrix</th>
<th>Computation time [sec.] (high performance computer)</th>
<th>Computation time [sec.] (standard PC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5 000</td>
<td>5 000</td>
<td>25 000 000</td>
<td>0.724</td>
<td>3.385</td>
</tr>
<tr>
<td>2</td>
<td>10 000</td>
<td>10 000</td>
<td>100 000 000</td>
<td>2.520</td>
<td>13.400</td>
</tr>
<tr>
<td>3</td>
<td>30 000</td>
<td>30 000</td>
<td>900 000 000</td>
<td>22.020</td>
<td>105.502</td>
</tr>
<tr>
<td>4</td>
<td>32 000</td>
<td>32 000</td>
<td>1 024 000 000</td>
<td>25.040</td>
<td>172.879</td>
</tr>
<tr>
<td>5</td>
<td>40 000</td>
<td>40 000</td>
<td>1 600 000 000</td>
<td>out of memory exception</td>
<td>out of memory exception</td>
</tr>
</tbody>
</table>

In Fig. 3, a graphical interpretation of the proposed solutions for large strings is presented. One very large matrix is built from the smaller component matrices resulting from the structure of the analyzed substrings. Each component matrix (i.e. values from last column and last row) is calculated and some of their values are transmitted to the next small matrix where they become initial values. This procedure reiterates through all component matrices. Each component matrix is calculated by the parallelized threads (1, 2 .. n) with the use of an array of locks algorithm where a younger thread (e.g. n – 1) waits for an older one (e.g. n – 2). Matrices which will not be used anymore are removed from the memory.

The pseudo-code below Fig. 3 describes a part of the function *LevDistDecomposition* where the input strings *Text1* and *Text2* are decomposed for the smaller substrings. Based on two subsequent substrings the component matrices are calculated and their final boundary elements are collected in two one-dimensional arrays: *arrVertical* and *arrHorizontal*. Next, these arrays are transmitted to the new small matrix in which they stay as the initial values for further calculations. Finally, the algorithm returns the Levenshtein distance as the result.

---

2 All the results for the described algorithms were calculated with the use of a 64-bit console application (written in C# language) available on website: www.pk.edu.pl/~aniewiarowski/publ/LevParallelCS.exe.
input variables: char Text1[0..M-1], char Text2[0..N-1], number of parts, number of parallel threads
declare: part_rangeX[0..number of ranges of horizontal parts], part_rangeY[0..number of ranges of vertical parts] as structure of int from and int to
declare: results_of_matrix[0..number of ranges of horizontal parts, 0..number of ranges of vertical parts] as structure of int[]
arrVertical[0..length of Text1] and int[] arrHorizontal[0..length of Text2]

for vm from 0 to M do arrVertical[vm]:=vm
for vn from 0 to N do arrHorizontal[vn]:=vn

for mx from 1 to number of ranges of horizontal parts (i.e. number of elements of part_rangeX)
for my from 1 to number of ranges of vertical parts (i.e. number of elements of part_rangeY)
results_from_matrix[mx, my] :=
LevDistParallelParts(
    substring of Text1 at (part_rangeX[mx-1].from-1, part_rangeX[mx-1].to - part_rangeX[mx-1].from+1),
    substring of Text2 at (part_rangeY[my-1].from-1, part_rangeY[my-1].to - part_rangeY[my-1].from+1),
    number of parallel Threads,
    results_from_matrix[mx - 1, my].arrVertical,
    results_from_matrix[mx, my - 1].arrHorizontal
)
clear results_from_matrix[mx - 1, my].arrVertical
clear results_from_matrix[mx, my - 1].arrHorizontal
end for (variable my)
end for (variable mx)

return the last element of arrVertical (or arrHorizontal) of the last
element of result_from_matrix
In the pseudo code-above, some designations are taken:

- **LevDistParallelParts** – function for calculation of the component matrices,
- **results_from_matrix** – two-dimensional array, collects elements of one-dimensional arrays: arrVertical and arrHorizontal,
- **part_rangeX, part_rangeY** – one-dimensional arrays with elements representing the range of calculated component matrices,
- **arrVertical, arrHorizontal** – one-dimensional arrays of results, input parameters for **LevDistParallelParts** function.

Additional complex operations, required for the implementation of the parallelization procedure are presented precisely in the pseudo-code in appendix A.

### 4. Results

In the present research, the **Microsoft .NET** technology was used (C# language) [14]. The proposed algorithm was tested with **mono-project** (cross platform, open source .NET development framework) [5] and **OS Fedora Linux**. The obtained results are presented in Fig. 4. The diagram shows the relationship between the length of analyzed strings (additionally described by the number of parts of the main matrix – right-hand legend in Fig. 4) and the number of parallel threads for one component matrix.

![Diagram showing correlations between length of strings and computation time for different numbers of parallel threads](image)

Fig. 4. Correlations between length of strings and computation time for different numbers of parallel threads for one component matrix

---

3 Some details of .NET’s threads and parallel technologies are presented in [15].
threads for one component matrix. As can be seen, the final computation time strongly depends on string sizes and on the number of parallel threads. In the presented examples, the best optimal results were obtained for about 100 threads applied for the component matrix in all cases of partitioning. For other numbers of threads, the parallelized parts of the component matrix were too large or too small and the procedure of threads construction was not cost-effective.

In Fig. 5, the relationship between the number of parts of a large decomposed matrix (1·10^{12} elements) and the computation time is presented. The obtained results show that the speed of calculations strongly depends on the number of parts (i.e. sizes of component matrices) of the decomposed matrix as well. This effect influences the main decomposition algorithm (in function \textit{LevDistDecomposition}), in which the one-dimensional array of results becomes the input data for the next iteration, and some old data are removed from memory. If there are too many parts, the transfer of partial results (and other operations) will be too frequent.

![Fig. 5. Relationship between number of parts (blocks) of decomposed matrix and computation time for string size of 1·10^6 × 1·10^6](image)

The results obtained for strings depicted with the Levenshtein matrix of 4·10^{12} elements, are presented in Tab. 3. It turns out that for parts 50 × 50 and 60 × 60, the component matrices were too large and the system could not allocate them in memory.

In Fig. 6, the consumption times of computations of the Levenshtein distance for very long strings with the use of optimized parameters (the best values of number of parallel threads for one component matrix and number of parts of decomposed main matrix) are presented. It is worth underlining that the calculation time versus the string’s length grows approximately according to the power function.
Next, the calculation times of the Levenshtein distance for very long texts with or without the parallelization procedure were compared. Different length texts were analyzed and the obtained results are presented in Tab. 4 and Fig. 7.

<table>
<thead>
<tr>
<th>String sizes</th>
<th>Number of parallel threads for component matrix</th>
<th>Number of parts of main matrix</th>
<th>Computation time [sec.]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 , 000 , 000 \times 1 , 000 , 000$</td>
<td>100</td>
<td>$50 \times 50$</td>
<td>4.590.35</td>
</tr>
<tr>
<td>$1 , 000 , 000 \times 1 , 000 , 000$</td>
<td>100</td>
<td>$60 \times 60$</td>
<td>4.601.1</td>
</tr>
<tr>
<td>$1 , 000 , 000 \times 1 , 000 , 000$</td>
<td>100</td>
<td>$70 \times 70$</td>
<td>4.669.4</td>
</tr>
<tr>
<td>$1 , 000 , 000 \times 1 , 000 , 000$</td>
<td>100</td>
<td>$80 \times 80$</td>
<td>4.704.4</td>
</tr>
<tr>
<td>$2 , 000 , 000 \times 2 , 000 , 000$</td>
<td>100</td>
<td>$50 \times 50$</td>
<td>out of memory exception</td>
</tr>
<tr>
<td>$2 , 000 , 000 \times 2 , 000 , 000$</td>
<td>100</td>
<td>$60 \times 60$</td>
<td>out of memory exception</td>
</tr>
<tr>
<td>$2 , 000 , 000 \times 2 , 000 , 000$</td>
<td>100</td>
<td>$70 \times 70$</td>
<td>17.360.4</td>
</tr>
<tr>
<td>$2 , 000 , 000 \times 2 , 000 , 000$</td>
<td>100</td>
<td>$80 \times 80$</td>
<td>17.630.58</td>
</tr>
</tbody>
</table>

Fig. 6. Computation times of calculations for very long strings using matrix decomposition with the parallelization procedure
Fig. 7. Computation times of calculations of long strings using Levenshtein distance algorithm with and without parallelization for one decomposed matrix

Based on Fig. 7, it can be seen that the parallelized Levenshtein distance algorithm is about 4–5 times faster than that without the parallelization procedure applied.

<table>
<thead>
<tr>
<th>Chars in text $A$</th>
<th>Chars in text $B$</th>
<th>Comp. time [sec.] – with parallel procedure</th>
<th>Comp. time [sec.] – no parallel procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 000</td>
<td>5 000</td>
<td>0.18</td>
<td>0.72</td>
</tr>
<tr>
<td>10 000</td>
<td>10 000</td>
<td>0.55</td>
<td>2.52</td>
</tr>
<tr>
<td>20 000</td>
<td>20 000</td>
<td>1.95</td>
<td>9.84</td>
</tr>
<tr>
<td>25 000</td>
<td>25 000</td>
<td>2.97</td>
<td>15.33</td>
</tr>
<tr>
<td>30 000</td>
<td>30 000</td>
<td>4.20</td>
<td>22.08</td>
</tr>
<tr>
<td>32 000</td>
<td>32 000</td>
<td>4.82</td>
<td>25.04</td>
</tr>
<tr>
<td>33 000</td>
<td>33 000</td>
<td>out of mem. exception</td>
<td>out of mem. exception</td>
</tr>
</tbody>
</table>

Figure 8 presents the computation times of DNA sequences for one decomposed Levenshtein-Damerau matrix with the use of assumed (100) number of parallel threads. As can be seen, the parallelized algorithm is again about 4-5 times faster than the algorithm not being parallelized. DNA sequences consist of chars T, A, G, C (described in Fig. 2).
5. Conclusions and further work

The research presented in this paper results in a method for the parallelization of the Levenshtein distance algorithm. Its implementation allows for the improvement of the speed of calculating the similarity measure of two long strings. In the presented examples, a high efficiency of the proposed techniques was achieved and very good results on a high-performance computer were confirmed [4].

The algorithm proposed in the paper was implemented in the mechanism of automatic selection of promotors and reviewers of diploma thesis within the system “Diplomas’ Manager”. This system was implemented at the Faculty of Physics, Mathematics and Computer Science of the Cracow University of Technology as a tool for diploma thesis management. The solution was also implemented within the anti-plagiarism system of “Diplomas’ Manager” and high efficiency in the case of searches for plagiarism was achieved.

In our future research, efforts will be undertaken towards improving the Levenshtein-Damerau algorithm for analyzing very long DNA sequences by decomposing the main matrix in accordance with the proposed algorithms. Moreover, the MPI technology will be implemented for computing each part of a large matrix. Furthermore, the introduction of measures of the distance between text elements (terms) in the analyzed text documents to build its internal specific characteristic and document structure is also anticipated [1].

4 System is available on web page: https://administracja.fmi.pk.edu.pl/~dyplomy.
References


Appendix 1

The pseudo-code below describes part of the function *LevDistParallelParts* which provides the parallelization procedure of component matrix *dpart*. The procedure calls threads that are assigned to insert the one-dimensional array *pth*. For iteration of all elements of matrix *dpart* two loops for loop in the line 16 and loop in the line 2, are required. The number of iterations within the loop in line 21 is restricted by the number of threads. Additionally, in line, 19 the automat of waiting mechanism is implemented and a loop with the array of locks *point_lock* is introduced. Based on this, the thread number *p* in loop for (line 16) waits for parallel thread number *p-1* (line 19) until column *i* in thread *p-1* has all values calculated.

```plaintext
input variables: number of parallel threads nTh, char fragText1[0..tM-1] char fragText2[0..tN-1], initial values included in arrVertical and arrHorizontal arrays
declare: array of threads pth[0..nTh], array arrRanges[0..tN] of structure int form and
int to, array dpart (part of Levenshtein distance matrix)
calculate ranges for Y size of matrix dpart and save in arrRanges
for i from 0 to tM
    d[i, 0] := arrHorizontal[i]
for j from 0 to tN
    d[0, j] := arrVertical[j]
for p from 1 to nTh
    pth[p] := new Thread({
        int cost
        for i from 1 to tM
            point_lock[num] = i
            wait if i >= point_lock[num - 1]
            for j from arrRanges[num - 1].from to arrRanges[num - 1].to
                if substring of Text1 at (i - 1) = substring of Text2 at (j - 1)
                    then
                        cost := 0  else  cost := 1
                dpart[i, j] := Minimum(
                    dpart[i - 1, j] + 1,
                    dpart[i, j - 1] +1,
                    dpart[i - 1, j - 1] +cost)
            end for (variable j)
        end for (variable i)
        increment +1 of point_lock[num]
    }
run thread pth[p]
end for (variable p)
wait for finish thread pth[nTh] (i.e. wait for all threads)
return structure of (array dpart[0..tM-1, tN-1], array dpart[tM-1, 0..tN-1])
```
In the pseudo code-above, some designations are taken:

- pth – one-dimensional array of thread objects,
- point_lock – one-dimensional array of locks,
- arrRanges – one-dimensional array of calculated ranges of areas of component matrix,
- num, p – thread number (i.e. number of part matrix),
- dpart – component matrix of Levenshtein matrix x,
- LevDistParallelParts – the Levenshtein distance obtained with the parallelization procedure.