TECHNICAL TRANSACTIONS | **CZASOPISMO TECHNICZNE**
FUNDAMENTAL SCIENCES | NAUKI PODSTAWOWE

3-NP/2014

ARTUR NIEWIAROWSKI, MAREK STANUSZEK*

PARALLELIZATION OF THE LEVENSHTEIN DISTANCE ALGORITHM

ZRÓWNOLEGLENIE ALGORYTMU ODLEGŁOŚCI EDYCYJNEJ LEVENSHTEINA

A b s t r a c t

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

S t r e s z c z e n i e

Artykuł przedstawia metodę zrównoleglenia algorytmu analizy odległości edycyjnej Levenshteina dedykowaną bardzo dużym ciągom tekstowym. Zaproponowane rozwiązanie zostało zaimplementowane na platformie .NET Framework 4.0 z uwzględnieniem metod dostępnych w przestrzeni nazw System.Threading.Task. Zastosowane algorytmy przetestowano na komputerze wysokiej wydajności, w oparciu o narzędzia Xamarin Mono (dla SO Linux RedHat/ Fedora). Otrzymane wyniki pokazują znacząco zwiększoną wydajność obliczeń dla przedstawionych w artykule rozwiązań.

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:

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

The main idea of the Levenshtein distance algorithm (*LevenshteinDistance* function) is described by the following pseudo-code:

```
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
```

```
                cost := 0 else cost := 1
                d[i, j] :=
                    Minimum(d[i - 1, j] + 1,
                    d[i, j - 1] + 1,
                    d[i - 1, j - 1] + cost)
            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: *Text1* and *Text2*,
M, N	–	lengths of two terms respectively,
$d[i, j] - (i, j)$	–	element of Levenshtein matrix d,
Minimum	–	a function to calculate a minimum of three variables,
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] :=
        Minimum(d[i - 1, j] + 1,
        z[i, j - 1] + 1,
        z[i - 1, j - 1] + cost)

   if i > 1 and j > 1 and substring of Text1 at (i-1) = substring
of Text2 at (j – 2) and substring of Text1 at (i-2) = substring of
Text2(j-1]) then
        z[i, j] := Minimum(z[i, j], z[i - 2, j - 2] + 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:

$$D = z[M, N] = LevensteinDamerauDistance\ (Text1,\ Text2)$$

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.

		y	e	s	t	e	r	d	a	y
	0	1	2	3	4	5	6	7	8	9
t	1	1	2	3	3	4	5	6	7	8
o	2	2	2	3	4	4	5	6	7	8
m	3	3	3	3	4	5	5	6	7	8
o	4	4	4	4	4	5	6	6	7	8
r	5	5	5	5	5	5	5	6	7	8
r	6	6	6	6	6	6	5	6	7	8
o	7	7	7	7	7	7	6	6	7	8
w	8	8	8	8	8	8	7	7	7	8

Fig. 1. Levenshtein matrix d, constructed for terms: yesterday and tomorrow[1]

		G	G	C	C	T	C	C
	0	1	2	3	4	5	6	7
T	1	1	2	3	4	4	5	6
C	2	2	2	2	3	4	4	5
C	3	3	3	2	2	3	4	4
A	4	4	4	3	3	3	4	5
A	5	5	5	4	4	4	4	5
T	6	6	6	5	5	4	5	5
A	7	7	7	6	6	5	5	6

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 1

Examples of Levenshtein distance between two strings

No.	string 1	string 2	Levenshtein distance
1	Car	Cars	1
2	University	Universities	3
3	Tom is writing a letter	Tom is writin letters	4

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[2]

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

Time consumption in the case of complex Levenshtein matrices calculation

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

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.

Fig. 3. Parallelization procedure of the Levenshtein distance matrix decomposition

```
 1  input variables: char Text1[0..M-1], char Text[0..N-1], number of
 2  parts, number of parallel threads
 3  declare: part_rangeX[0..number of ranges of horizontal parts], part_
 4  rangeY[0..number of ranges of vertical parts] as structure of int from
 5  and int to
 6  declare: results_of_matrix[0..number of ranges of horizontal parts,
 7  0..number of ranges of vertical parts] as structure of int[]
 8  arrVertical[0..length of Text1] and int[] arrHorizontal[0..length of
 9  Text2]
10
11  for vm from 0 to M do arrVertical[vm]:=vm
12  for vn from 0 to N do arrHorizontal[vn]:=vn
13
14  for mx from 1 to number of ranges of horizontal parts (i.e. number of
15  elements of part_rangeX)
16      for my from 1 to number of ranges of vertical parts (i.e. number
17  of elements of part_rangeY)
18        results_from_matrix[mx, my] :=
19          LevDistParallelParts(
20          substring of Text1 at (part_rangeX[mx-1].from-1,
21           part_rangeX[mx-1].to – part_rangeX[mx-1].from+1),
22          substring of Text2 at (part_rangeY[my-1].from-1,
23           part_rangeY[my-1].to – part_rangeY[my-1].from+1),
24          number of parallel Threads,
25          results_from_matrix[mx - 1, my].arrVertical,
26          results_from_matrix[mx, my - 1].arrHorizontal
27         )
28         clear results_from_matrix[mx - 1, my].arrVertical
29         clear results_from_matrix[mx, my - 1].arrHorizontal
30  end for (variable my)
31  end for (variable mx)
32
33  return the last element of arrVertical (or arrHorizontal) of the last
34  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][3]. 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

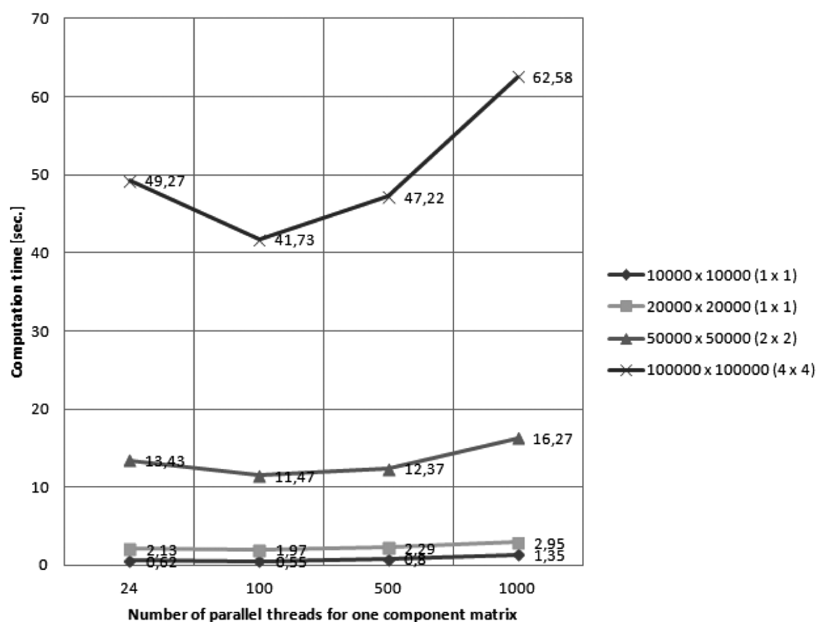

Fig. 4. Correlations between length of strings and computation time for different numbers of parallel threads for one component matrix

[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 ($1 \cdot 10^{12}$ 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 *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.

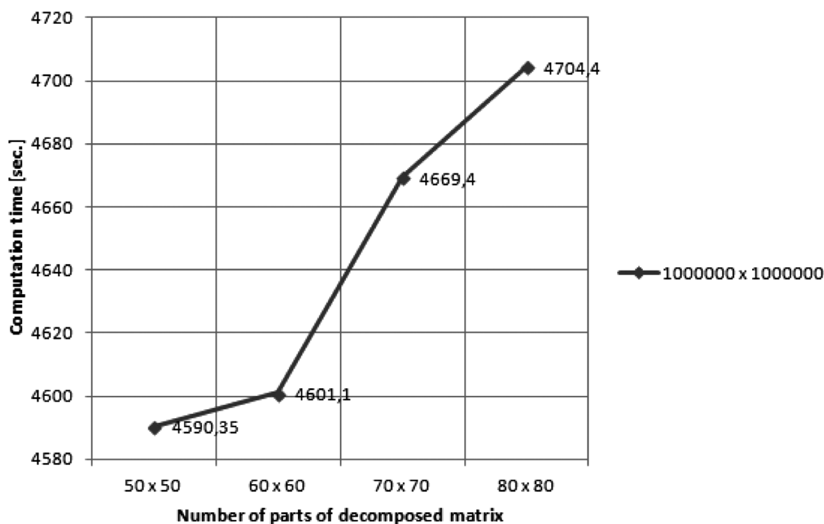

Fig. 5. Relationship between number of parts (blocks) of decomposed matrix and computation time for string size of $1 \cdot 10^6 \times 1 \cdot 10^6$

The results obtained for strings depicted with the Levenshtein matrix of $4 \cdot 10^{12}$ 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 3

Computation times for two long strings according to number of blocks of main matrix

String sizes	Number of parallel threads for component matrix	Number of parts of main matrix	Computation time [sec.]
1 000 000 × 1 000 000	100	50 × 50	4 590.35
1 000 000 × 1 000 000	100	60 × 60	4 601.1
1 000 000 × 1 000 000	100	70 × 70	4 669.4
1 000 000 × 1 000 000	100	80 × 80	4 704.4
2 000 000 × 2 000 000	100	50 × 50	out of memory exception
2 000 000 × 2 000 000	100	60 × 60	out of memory exception
2 000 000 × 2 000 000	100	70 × 70	17 360.4
2 000 000 × 2 000 000	100	80 × 80	17 630.58

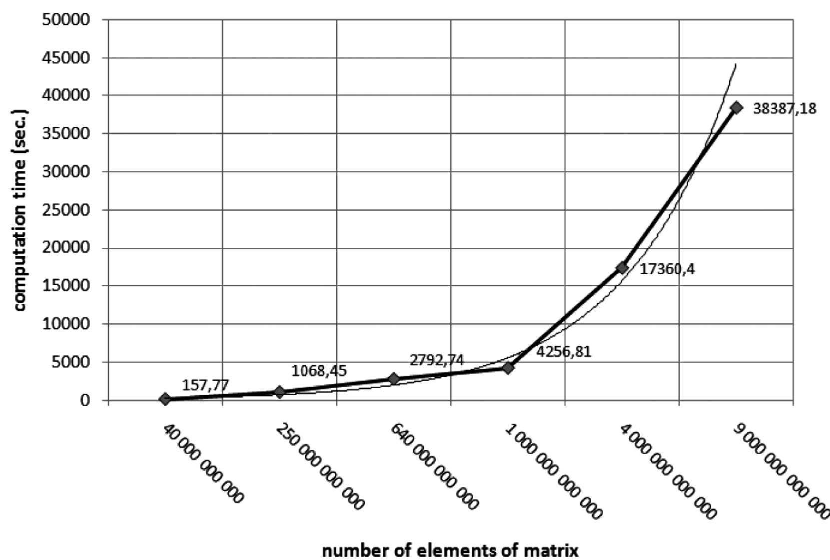

Fig. 6. Computation times of calculations for very long strings using matrix decomposition with the parallelization procedure

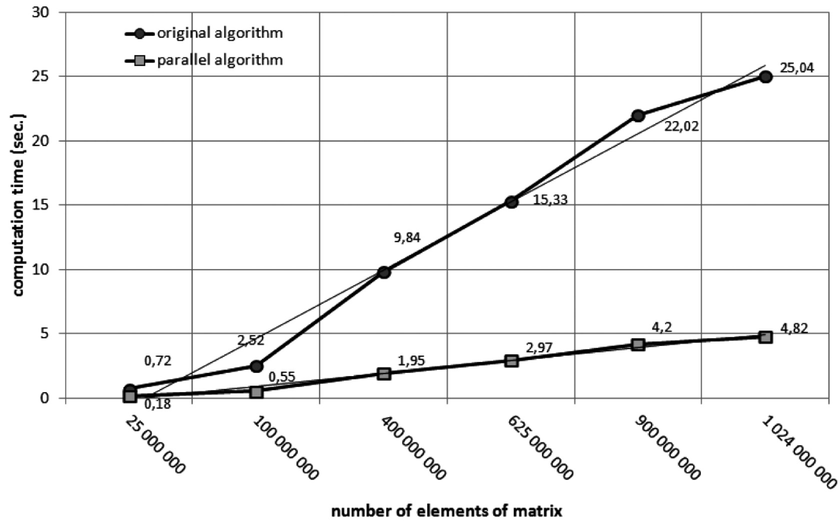

Fig. 7. Computation times of calculations of long strings using Levenshtein distance algorithm with and without parallelization for one decomposed matrix

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 4

Computation times of Levenshtein distance algorithm with or without parallel procedure

Chars in text *A*	Chars in text *B*	Comp. time [sec.] – with parallel procedure	Comp. time [sec.] – no parallel procedure
5 000	5 000	0.18	0.72
10 000	10 000	0.55	2.52
20 000	20 000	1.95	9.84
25 000	25 000	2.97	15.33
30 000	30 000	4.20	22.08
32 000	32 000	4.82	25.04
33 000	33 000	out of mem. exception	out of mem. exception

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).

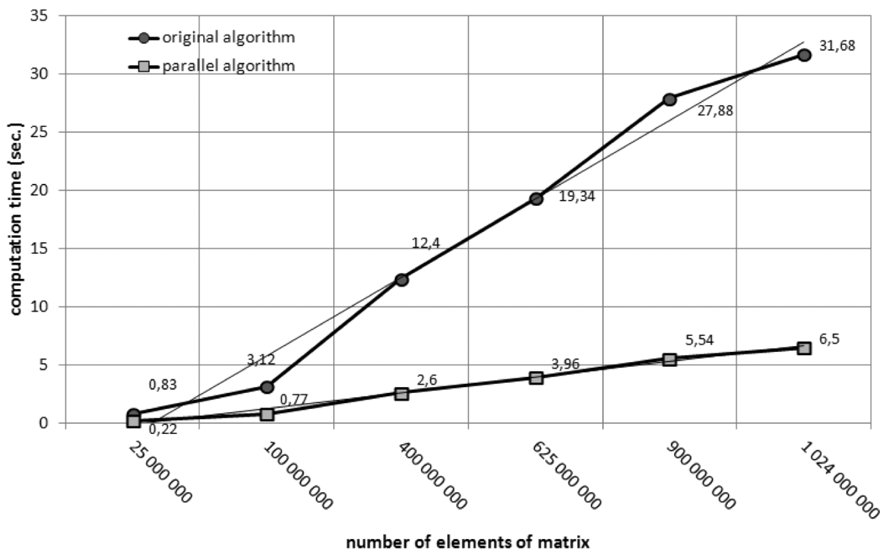

Fig. 8. Computation times of calculations of DNA sequences using Levenshtein-Damerau algorithm with and without the parallelization procedure for one decomposed matrix

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[4] 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

[1] Niewiarowski A., Stanuszek M., *The mechanism of identification and classification of content*, Studia Informatica, Vol. 34, 2B(112), Silesian University of Technology Press, Gliwice 2013, 205-222.

[2] Niewiarowski A., Stanuszek M., *Mechanism of analysis of similarity short texts, based on the Levenshtein distance*, Studia Informatica. Vol. 34, 1 (110), Silesian University of Technology Press, Gliwice 2013, 107-114.

[3] Niewiarowski A., *Term frequency optimization for the vector space model*, Czasopismo Techniczne, 9-M/2012, 155-165.

[4] Kobzdej P., Waligóra D., Wielebińska K., Paprzycki M., *Parallel Application of Levenshtein Distance to Establish Similarity Between Strings*, International Journal of Computer Research, Vol. 12, No. 4, 2003, 625-633.

[5] Mono-project (www.mono-project.com/What_is_Mono).

[6] Левенштейн В.И., *Двоичные коды с исправлением выпадений, вставок и замещений символов*, Доклады Академий Наук СССР 163 (4), 1965, 845-848.

[7] Wypych M., *Stochastic Spelling Correction of Texts in Polish*, Institute of Linguistics, Adam Mickiewicz University, Poznań, Poland; Speech and Language Technology. Volume 6, Poznań 2002.

[8] Damerau F.J., *A technique for computer detection and correction of spelling errors*, Communications of the ACM, 7 (3), 1964, 171-176.

[9] Runkler T.A., Bezdek J.C., *Web mining with relational clustering*, International Journal of Approximate Reasoning, Vol. 32, Issues 2–3, February 2003, 217-236.

[10] Niewiarowski A., *Działanie parsera ‚Part-of-Speech Tagging' w ujęciu mechanizmu Web Content Mining*, Wydawnictwo VI Ogólnopolskiej Konferencji Naukowej Nauka i Przemysł, Politechnika Krakowska im. Tadeusza Kościuszki, Kraków 2011, 93-100.

[11] Niewiarowski A., Stanuszek M., *Parallelize edit distance algorithm*, Proceedings. Seventh ACC Cyfronet AGH Users' Conference, Academic Computer Centre Cyfronet AGH, Zakopane 2014, 31-32.

[12] Niewiarowski A., Stanuszek M., *Performance and quality of method for short text similarity algorithm based on edit distance and thesaurus*, Proceedings. Seventh ACC Cyfronet AGH Users' Conference, Academic Computer Centre Cyfronet AGH, Zakopane 2014, 33-34.

[13] Ramos J., *Using tf-idf to determine word relevance in document queries*, Proceedings of the First Instructional Conference on Machine Learning, 2003.

[14] Campbell C., Johnson R., Miller A., Toub S., *Parallel Programming with Microsoft. NET. Design Patterns for Decomposition and Coordination on Multicore Architectures*, Microsoft Press, 2010.

[15] Niewiarowski A., *Szybko zrozumieć Visual Basic 2012*, Self Publishing. Kraków 2013, 66-73.

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.

```
 1   input variables: number of parallel threads nTh, char fragText1[0..tM-
 2   1] char fragText2[0..tN-1], initial values included in arrVertical and
     arrHorizontal arrays
 3   declare: array of threads pth[0..nTh], array arrRanges[0..tN] of
     structure int form and
 4   int to, array dpart (part of Levenshtein distance matrix)
 5
 6   calculate ranges for Y size of matrix dpart and save in arrRanges
     for i from 0 to tM
 7           d[i, 0] := arrHorizontal[i]
 8   for j from 0 to tN
 9           d[0, j] := arrVertical[j]
10
11   for p from 1 to nTh
12    pth[p] := new Thread((num) of
13    {
14    int cost
15    for i from 1 to tM
16
17       point_lock[num] = i
18       wait if i >= point_lock[num - 1]
19
20       for j from arrRanges[num - 1].from to arrRanges[num - 1].to
21
22       if substring of Text1 at (i – 1) = substring of Text2 at (j – 1)
23   then
         cost := 0  else  cost := 1
24
25       dpart[i, j] := Minimum(
26                      dpart[i - 1, j] + 1,
27                      dpart[i, j - 1] +1,
28                      dpart[i - 1, j - 1] +cost)
29
30       end for (variable j)
31    end for (variable i)
32    increment +1 of point_lock[num]
33    }
34   run thread pth[p]
35   end for (variable p)
36
37   wait for finish thread pth[nTh] (i.e. wait for all threads)
38   return structure of (array dpart[0..tM-1, tN-1], array dpart[tM-1,
39   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.