A New Dominant Point-Based Parallel Algorithm for Multiple Longest Common Subsequence Problem

Dmitry Korkin

This work introduces a new parallel algorithm for computing a multiple longest common subsequence (MLCS). Given a set of strings, the longest common subsequence can be obtained by removing a number of symbols from each sequence. Although there was a lot of research done in the parallelization of MLCS algorithms for the special case of two sequences, so far there were no any general parallel methods for computing MLCS of an arbitrary number of sequences. Our method is a parallel approach to dominant points-based method recently proposed by Hakata and Imai. The parallel algorithm is presented and the related theoretical results as well as the algorithm's implementation on IBM SP3 using MPI system are discussed.

longest common subsequence, dominant points, parallel algorithms, IBM SP3

I. Introduction

A. Basic definitions

The multiple longest common subsequence (MLCS) problem can be defined as follows: Let $A_1, A_2, \dots, A_d$ be a set of d sequences of length $n_1, n_2, \dots, n_d$, correspondingly, over alphabet $\Sigma = \{\sigma_1,\sigma_2, \dots, \sigma_s\}$. A *subsequence* of A_i can be obtained by removing zero or more symbols from A_i. More precisely, if $A = s_1 s_2 \dots s_n$, then $B = s_{i1} s_{i2} \dots s_{ik}$ is a subsequence of A if $\forall j \in \{1,2, \dots ,k\}$: $i_j \in \{1,2, \dots ,n\}$ and $\forall s,t \in \{1,2, \dots ,k\}, s<t: i_s<i_t$, where k is the length of B. The multiple longest common subsequence problem for a set of sequences $A_1, A_2, \dots, A_d$ is to find a sequence B such that B is a subsequence of each A_i and it has the largest length. For example, B = *'ce'* is the MLCS for A_1 = *'computer'* and A_2 = *'science'*. Note, that for a given set of sequences there can be more than one MLCS. In the case $d=2$ MLCS problem is simply called the longest common subsequence problem (LCS).

The MLCS is widely used in bioinformatics and computational biology, mostly in DNA and protein sequence analysis. One of MLCS most direct implementation in a protein sequence analysis is a search for a *motif* or sets of motifs given a protein family (motif is a short conserved region in a protein sequence with known or implied function).

B. Sequential algorithms for MLCS

The general idea is based on dynamic programming approach [1]. This approach is also widely used in the similar to MLCS problem – *multiple sequence alignment* (including pairwise alignment) of a set of sequences. In general, given d sequences, $A_1, A_2, \dots, A_d$, the dynamic programming algorithm builds a d-dimensional *score matrix* $L[0, \dots, n_1; 0, \dots, n_2; \dots ; 0, \dots, n_d]$, where $n_1, n_2, \dots, n_d$ are the corresponding lengths of the above sequences. The score matrix is computed iteratively. For example, in a case of two strings, $A[1, \dots ,n]$ and $B[1, \dots ,m]$, the score matrix L is computed as follows. First, we assign 0 values to the first row and column, i.e. $L[i, 0] = L[0, j] = 0, 0\le i,j\le n$. Then, the value of each element $L[i,j], 1\le i,j\le n$, of matrix L can be calculated iteratively via the values of elements that were computed before as follows:

$$L[i,j] = \begin{cases} L[i-1,j-1]+1 & \text{if } A[i] = B[j] \\ \max\{L[i-1,j],L[i,j-1]\} & \text{otherwise} \end{cases}$$

TABLE 2
THE PROCESS OF COMPUTING THE SCORE MATRIX FOR TWO SEQUENCES, A = 'AABCAABCAB' AND B = 'BACBAB'

		a	a	b	c	a	a	b	c	a	b
	0	0	0	0	0	0	0	0	0	0	0
b	0	0	0	1	1	1	1	1	1	1	
a	0	1	1	1	1	2	2	2	2		
c	0	1	1	1	2	2	2	2			
b	0	1	1	2	2	2					
a	0	1	2	2	2						
b	0	1	2	3							

There has been done a lot of work for implementing dynamic programming approach, mostly for the case of $d=2$ (see Table 2, [2]-[3]). Once obtained a score matrix, a MLCS can be easily derived by finding the lowest-cost path from point $[n_1, n_2, \dots, n_d]$ to point $[0,0, \dots ,0]$. The point

Dmitry Korkin is with the Faculty of Computer Science, University of New Brunswick, P.O.Box 4400, Fredericton, NB, Canada, E3B 5A3 (phone: 506-451-6931, e-mail: z17b3@unb.ca).

$p[i_1, i_2, \ldots i_d]$ corresponds to a position in a MLCS if when moving along the above path to point $[0,0, \ldots ,0]$, each time we go from position p to the next in the path position q the score for the latter is $L[q] = L[p]-1$ (see Table 3).

TABLE 2
SELECTED RESULTS FOR THE LCS PROBLEM ($D=2$). HERE P IS THE LENGTH OF LCS, S IS THE SIZE OF ALPHABET, M IS THE NUMBER OF MINIMAL MATCHES. NOTE THAT FOR CONVENIENCE WE ARE ASSUMING EQUAL LENGTHS OF THE SEQUENCES

N	**Year**	**Author(s)**	**Time**	**Ref.**
1	1974	Wagner, Fischer	$O(n^2)$	[2]
2	1975	Hirschberg	$O(n^2)$	[3]
3	1980	Masek, Paterson	$O(n^2 / \log n)$	[4]
4	1984	Hsu, Du	$O(pn \log(n/p)+pn)$	[5]
5	1986	Myers	$O(n(n - p))$	[6]
6	1992	Apostolico et al.	$O(n(n - p))$	[7]
7	1994	Rick	$O(ns + \min(ms,pn))$	[8]

The reason for not applying this method for three and more sequences is because of its time and space complexity. It is not hard to see that the straight-forward implementation of dynamic programming method for constructing the score matrix would lead to $O(n^d)$ time and space complexities. For the special case, $d=2$, there has been done a lot of successful work for improving both complexities (see Table 2, [4]-[8]). However, for the general case it is still an open question. On of the most promising approaches to the problem of finding MCLS is so-called *dominant points-based methods* [9,10]. As a base of our method we took one of the most advanced dominant points-based methods proposed by Hakata and Imai in [10,11].

TABLE 3
THE LOWEST-SCOREPATH IN THE SCORE MATRIX FOR TWO SEQUENCES, A = 'AABCAABCAB ' AND B = 'BACBAB' GIVES US A LCS = 'ACBAB'

		a	a	b	c	a	a	b	c	a	b
	0	0	0	0	0	0	0	0	0	0	0
b	0	0	0	1	1	1	1	1	1	1	1
a	0	1	1	1	1	2	2	2	2	2	2
c	0	1	1	1	2	2	2	2	3	3	3
b	0	1	1	2	2	2	2	3	3	3	4
a	0	1	2	2	2	3	3	3	3	4	4
b	0	1	2	3	3	3	3	4	4	4	5

II. DOMINANT POINTS-BASED SEQUENTIAL ALGORITHM

A. Method

The method is based on several ideas. In order to describe these ideas, let us introduce some definitions and notations. Given a set of sequences, $A_1, A_2, \ldots , A_d$, over alphabet $\Sigma = \{\sigma_1,\sigma_2, \ldots , \sigma_s\}$, the position p in the corresponding score matrix L is denoted as $p[p_1, p_2, \ldots , p_d]$, where each p_i is a coordinate of p for the corresponding string, A_i. For a sequence A we denote a symbol corresponding to the k-th position in A as $A[k]$.

Definition 1. Position p in L is called *a match* iff

$$A_1[p_1] = A_2[p_2] = \ldots = A_d[p_d].$$

A match p, corresponding to a symbol σ is denoted as $p(\sigma)$.

Definition 2. We say that point p *dominates* point q if $p_i \le q_i$ for all $i = 1,2, \ldots ,d$. We denote this fact as $p \le q$. The relation $p < q$ can be defined similarly.

Definition 3. A match p is called a *k-dominant* iff

$$L[p] = L[p_1, p_2, \ldots , p_d] = k.$$

The set of all k-dominants for a point p is denoted as $D^k(p)$. The set of all k-dominants is denoted as $D(p)$.

Definition 4. A match $p(\sigma)$ is called a *σ-parent* of a point q iff $q < p$ and there is no match $r(\sigma)$ such that $q < r < p$. The set of all σ-parents of q is denoted as $Par(q, \sigma)$.

Definition 5. A point p in a set of points S *is* a *minimal* element of S, if $\forall q \in S: \ q \not\le p$.

There are several main ideas leading to the above method. First, it is not hard to see that one should search among only matches since each position in a MLCS should at least be a match. Second, it can be shown that the 'special' points in the lowest-cost path corresponding to positions in the MLCS, which were discussed in the previous section, are k-dominants, for $k = 1, 2, \ldots,|MLCS|$. Next, one should note that those dominant points can be computed in a dynamic manner: each step we calculate the set of k-dominants for $k = 1,2, \ldots, |MLCS|$. Another important idea actually tells us how to compute the sets of k-dominants: given a set of k-dominants, the set of $k+1$-dominants is a subset of $Par(D^k, S)$. Finally, for each set $Par(D^k, S)$ only the minimal elements are dominant points and, thus, can be the candidates for positions in the MLCS (see Table 4). Based on these ideas and some more advanced properties of dominant points Hakata and Imai developed an algorithm for computing a MLCS of a set of d sequences.

TABLE 4
THE SET OF DOMINANTS AND MATCHES IN THE SCORE MATRIX FOR TWO SEQUENCES, A = 'AABCAABCAB ' AND B = 'BACBAB'. THE DOMINANT POSITIONS ARE CIRCLED WHILE THE REMAINING MATCHES THAT ARE NOT DOMINANT ARE SQUARED

		a	a	b	c	a	a	b	c	a	b
	0	0	0	0	0	0	0	0	0	0	0
b	0	0	0	①	1	1	1	$\boxed{1}$	1	1	$\boxed{1}$
a	0	①	$\boxed{1}$	1	1	②	$\boxed{2}$	2	2	$\boxed{2}$	2
c	0	1	1	1	②	2	2	2	③	3	3
b	0	1	1	②	2	2	2	③	3	3	④
a	0	$\boxed{1}$	②	2	2	③	$\boxed{3}$	3	3	④	4

b	0	1	2	③	3	3	3	④	4	4	⑤

B. *Sequential algorithm*

For the comparison with our parallel algorithm for MLCS, an algorithm discussed by Hakata and Imai in [10,11] was implemented. A simplified pseudocode for the above dominant points-based sequential algorithm is given below.

```
Algorithm Seq_MCLS({A1,A2,…,Ad},Σ)
(1)     $D^0$ = {[0,0,…,0]}; k=0;
(2)     while D k not empty do {
          A = B = ∅
(2.1)     for p ∈ $D^k$ do {
(2.1.1)     Par(q,Σ) = Parents(p)
(2.1.1)     A = A ∪ Par(q,Σ)
          }//for p
(2.2)     $D^{k+1}$ =Minima(A)
(2.3)     k = k + 1
        }//while
(3)     pick a point p ∈ $D^{k-1}$
        while k-1 > 0 do
         { current LCS position = $A_1[p_1]$
(3.1)      p = q, where q is such that
              p ∈ Par(q,Σ)
(3.2)      k = k – 1
         }
```

The algorithm uses two functions, *Parents(p)* that gives a set of all parents of *p*, *Par(q,Σ)*, and *Minima(A)* that gives all the minimal elements for the given set of positions *A*. While the former algorithm is pretty straight forward the pseudocode for *Minima* is presented below.

```
Function Minima(S)
(1) A = ∅
(2) for i=1 to |S| do
      f [i]=1
(3) for i=1 to |S| do {
      if f[i]=1 then {
        for j=1 to |S| do {
          if y[j]=1 then {
            if S[i]=S[j] and j>i
              then f[j]=0
            if S[i]≠S[j] and S[i]≥S[j]
              then f[i]=0
            if S[i]≠S[j] and S[i]≤S[j]
              then f[j]=0
          } //if y[j]
          if f[i]=1 then A=A∪{S[i]}
        } //for j
      } //if f[i]
    } // for i
(4) return A
```

C. *Results*

The theoretical results for *Seq_MLCS* algorithm are presented by the following theorems (for convenience, here and below we assume the lengths of all sequences are the same and equal to n).

Theorem 1. Seq_MLCS algorithm correctly computes the LCS of sequences $A_1, A_2, \ldots, A_d$.

Lemma 1. Minima function takes $O(dn^2)$.

Hakata and Imai showed in [10,11]that the theoretical time complexity for the algorithm for finding the set of minimal points can be improved.

Lemma 2 [10,11]. For $d \geq 3$, the minima of n points in the d-dimensional space can be computed in $O(dn \log^{d-2} n)$ time by a divide-and-conquer algorithm.

Based on this results the theoretical time complexity for a slightly improved version of Seq_MLCS is given by the following theorem.

Theorem 2 [10]. The MLCS problem for $d \geq 3$ strings of length *n can be solved in time*

$$O(nsd + |D|sd (\log^{d-3} n + \log^{d-2} s)),$$ where $|D|$ is the size of the set of all dominant positions..

It is not hard to see that the size $|D|$ of the set of all dominants is much smaller than the size of set of all positions. Although a nontrivial (rather than by n^d) estimation of $|D|$ is still an open question the results obtained by the implementation of dominant point-based approach show the great advantage of this method in contrast to classical dynamic programming approaches. These results will be briefly discussed in *Section IV*.

III. PARALLELIZATION STRATEGY

Also there were series of successful parallelization approaches to the LCS problem, which is a special case of MLCS when $d=2$ (see, for example, [12]-[13]), we have not seen *any* approaches for the general case of MLCS problem.

A. *Method*

When designing our parallel algorithm based on dominant points approach we need, first, to analyze, which parts of *Seq_MCLS* algorithm can be parallelized, then, to implement the parallelization of the above parts, and finally, to combine the parallelized parts with the remaining parts of the algorithms. It turns out that because of the nature of dynamic programming-based approaches for MLCS problem (as well as for many others), it is very difficult and, sometimes, impossible to implement a parallelization approach based on the static partition of initial data (e.g. dividing some of the given sequences onto subsequences and distributing them among the processors). Taking this fact into consideration we developed a dynamic

distribution of data approach. Given *P* processors, one of which is a master and the rest of which are slaves, the main idea is to distribute each time all elements of a current *k*-dominant set, D^k, among *P* processors in order to parallelize the computation of sets of parents for each of the *k*-dominants (steps 2.1, 2.1.1, and 2.1.2 in *Seq_MCLS*). After calculating the sets of parents in parallel mode, all the slave processors return the parent sets back to the master processor, which performs, then, a computation of the next dominant set, D^{k+1}.
The data flow during execution of the algorithm can be illustrated by the following example of LCS problem for two sequences given in Table 4. Suppose, $P = 4$ processors. Then, based on the size of a current *k*-dominant set, $|D^k|$, we need to use $|D^k|$ processors, if $|D^k|<P$ and *P* processors otherwise (see Fig.1). Note, that in real examples (e.g. DNA or protein sequences), generally, the number of a current *k*-dominant set is relatively big (several hundreds and sometimes, thousands).

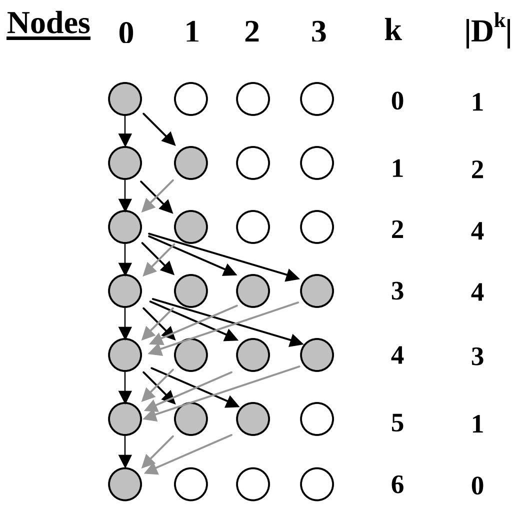

Fig. 1 Data flow for the example of LCS problems for two strings presented in Table 4. Black arrows show incoming data flow, gray arrows show outcoming data flow. The active processors are colored gray.

B. Parallel algorithm

Summarized the ideas discussed above, we designed and implemented the parallel algorithm for MLCS problem. The algorithm was implemented on the MPI (message-passing interface) system and run on local IBM SP3 machine. The simplified pseudocode of the algorithm is given below.

```
Algorithm Par_MCLS({A1,A2,…,Ad},Σ)
(1)     $D^0$ = {[0,0,…,0]}; k=0;
(2)     while D k not empty do {
          A = ∅
(2.1)     Proc0: Distribute elements of $D^k$
          // Each processor performs:
(2.2)     Get my_$D^k$: a subset of elements
                                          of $D^k$
          for p ∈ my_$D^k$ do {
(2.2.1)      Par(q,Σ) = Parents(p)
(2.2.2)      my_A = my_A ∪ Par(q,Σ)
          }//for p
          //Each slave processor finishes
(2.3)     Proc0: Gather elements of my_A
                 into A
(2.4)     $D^{k+1}$ =Minima(A)
(2.3)     k = k + 1
          }//while
(3)     pick a point p ∈ $D^{k-1}$
        while k-1 > 0 do
          { current LCS position = $A_1[p_1]$
(3.1)       p = q, where q is such that   p
                 ∈ Par(q,Σ)
(3.2)       k = k – 1
        }
```

C. Implementation issues

Since all slave processors during the while-loop in the above algorithm perform computation of the parents sets based on the elements of current k-dominant set, D^k, the process of sending-receiving data between the master processor, on one side, and the slave processors, on the other, requires synchronization. The synchronization is performed via a standard barrier implementation in a message-passing system. The process of data exchange is performed via MPI_Bcast and MPI_Gather routines.

IV. Results and Discussions

Since in the *Par_LCS* algorithm we were able to parallelize only a part of the sequential *Seq_LCS* algorithm, we cannot expect a linear speedup even in the theoretical analysis of the time complexity. The best performance will obviously be in a case when, for each current k-dominant D^k, all *P* available processors participate in computing of the parent sets for the elements of D^k, that is, when for each k: $|D^k| \geq P$. However, as it was mentioned in the previous section, it turns out, that for the most of real applications the most of the *k*-dominants have several hundreds and even thousands of elements, so the following estimation of time complexity for the parallelized part of *Par_LCS* algorithm is useful..

Lemma 2. If $\forall k$: $|D^k| \geq P$, then the parallelized part of *Par_CLS* takes $O(T/P)$, where T is a time taken by the corresponding part in a sequential algorithm, and T is $O(s|D|)$, where D is a set of all dominant positions, and s is the size of alphabet Σ.

To investigate the practical behavior of *Par_MLCS* algorithm we tested it on two main types of examples:

- Examples where the size of a current *k*-dominant set is small and might not exceed the total number of processors available (Figs. 2,3);

- Examples where the size of current k-dominants for most k is much larger than the number of processors available (Figs. 3,4). As the example of the ‘real world ‘ application we took a family of 6 protein sequences with the maximal sequence length equal to 63.

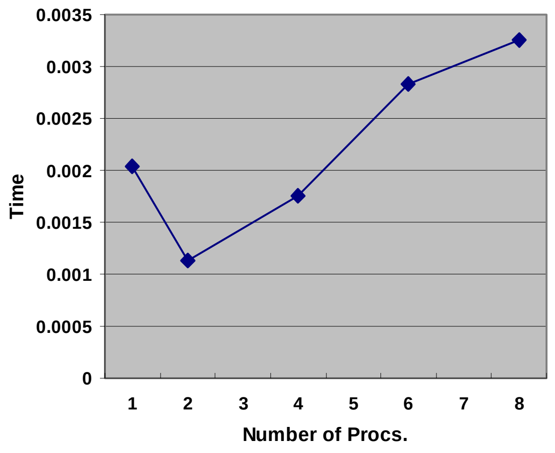

Fig. 2 Time table for a MLCS problem; *d = 3*, *n = 10*, and the maximal size of *k*-dominant is 4.

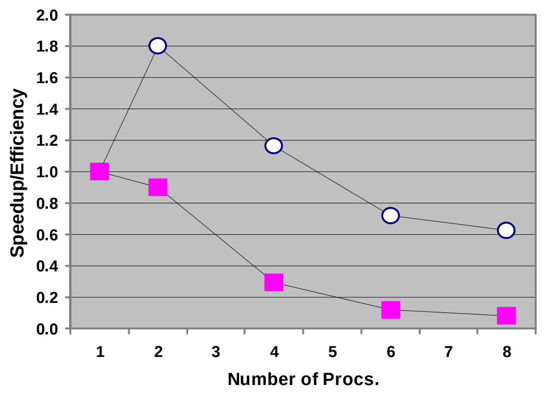

Fig. 3 Speedup (circled) and efficiency (squared) for a MLCS problem; *d = 3*, *n = 10*, and the maximal size of *k*-dominants is 4.

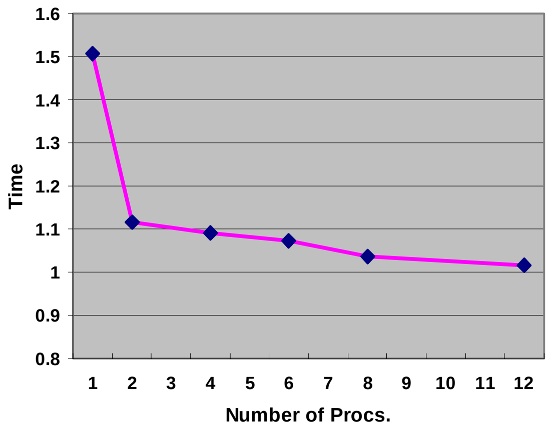

Fig. 4 Time table for a MLCS problem; *d = 6*, *n = 63*.

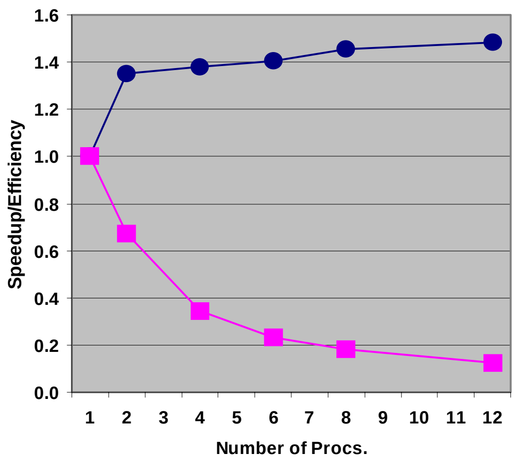

Fig. 5 Speedup (circled) and efficiency (squared) for a MLCS problem; *d = 6*, *n = 63*.

The results presented on the Figs. 2-5 confirm all our theoretical predictions. If in the case of small-size example it is not efficient to use a lot of processors for parallel computing of MLCS, in a case of ‘real world’ we can see a quite opposite picture. The results obtained, for example, in the case of $P = 12$ processors were almost twice better than the results obtained by Hakata and Imai in [10,11] for the same data set even although they were using an optimized version of the algorithm that we used as a base for our approach. Note, that for the same problem a classical dynamic programming algorithm would need about 190 Mb of memory and take more than 70 hours to run.

V. Conclusions and Future Research

Although the MLCS problem and its special case, LCS problem, have been studied by a relatively large number of researchers, and there has been some success reached in the parallelization of LCS problem, currently we don’t know any satisfactory approaches for parallelization of the general MLCS problem. We proposed a novel approach for parallelization of the one of most recent and advanced method for computing MLCS, proposed by Hakata and Imai and based on the concept of dominant points. We discussed the idea , implementation issues, and both, theoretical and experimental results of the new parallel approach to MLCS problem.

In the future we will be continuing work on the complete parallelization of the dominant point-based approach. In fact, the following result will allow us to parallelize the remaining part of the Seq_MLCS algorithm that has not been parallelized.

Proposition. If $D^k = D_1^k \cup D_2^k \cup \ldots \cup D_P^k$ and for each i: D_i^{k+1} is the k+1-dominant based on D_i^k, then:

$$D^{k+1} = \text{Minima}\ (D_1^{k+1} \cup D_2^{k+1} \cup \ldots \cup D_P^{k+1}),$$

where Minima(S) is a set of minimal elements of S.

VI. Reference

[1] Sankoff, D, “Matching sequences under deletion/insertion constraints,” *Proc. Natl. Acad. Sci. USA*, 69,4-6, 1972.

[2] Wagner, R. A., and M. J. Fischer, “The string-to-string correction problem,” *J. ACM*, Vol.21, No.1, 1974, pp.168-173.

[3] Hirschberg, D. S., “A linear space algorithm for computing maximal common subsequences”, C. J. Kaufman, Rocky Mountain Research Laboratories, Boulder, CO, 1992.

[4] Masek, W. J., and M. S. Paterson, “A faster algorithm computing string edit distances,” *JCSS*, 1980, pp.18-31.

[5] Hsu, W. J., and M. W. Du, “Computing a longest common subsequence for a set of strings,” *BIT*, Vol.24, 1984, pp.45-59.

[6] E. W. Myers, “An O(ND) Difference Algorithm and Its Variations,” *Algorithmica*, Vol.2, 1986, 251-266.

[7] Apostolico, A., and C. Guerra, “The longest common subsequence problem revisited,” *Algorithmica*, Vol.2, 1987, pp.315-336.

[8] Claus Rick, “New Algorithms for the Longest Common Subsequence Problem”, *Research Report No. 85123-CS*, University of Bonn, October 1994.

[9] Chin, F., and C. K. Poon, “Performance analysis of some simple heuristics for computing longest common subsequences,” *Algorithmica*, Vol.12, No.4-5, 1994, pp.293--311.

[10] K. Hakata and H. Imai, “Algorithms for the Longest Common Subsequence Problem for Multiple Strings Based on Geometric Maxima,” *Preprint*, 1998.

[11] K. Hakata and H. Imai, “The Longest Common Subsequence Problem for Small Alphabet Size between Many Strings,” *Proceedings of the 3rd International Symposium on Algorithms and Computation*, Nagoya, Lecture Notes in Computer Science Vol.650, December 1992, pp.469-478.

[12] Tieng K. Yap and Ophir Frieder and Robert L. Martino, “Parallel Computation in Biological Sequence Analysis,” *IEEE Transactions on Parallel and Distributed Systems*, Vol.9, No. 3, pp.283-294, 1998.

[13] Myoupo J-F. and Seme D., “Time-Efficient Parallel Algorithms for the Longest Common Subsequence and Related Problems,” *Journal of Parallel and Distributed Computing*, 57, 212-223, 1999.