A Novel Fast and Memory Efficient Parallel MLCS Algorithm for Long and Large-Scale Sequences Alignments

Yanni Li*, Yuping Wang†, Zhensong Zhang‡, Yaxin Wang§, Ding Ma¶ and Jianbin Huang ‖
*School of Software, Xidian University, China
†School of Computer Science and Technology, Xidian University, China
‡The Chinese University of Hong Kong, Hong Kong
§Henry Samueli School of Engineering and Applied Science, University of California, Los Angeles, USA
¶Department of Computer Science, Viterbi School of Engineering, University of Southern California, USA
‖School of Software, Xidian University, China
*yannili@mail.xidian.edu.cn, †ywang@xidian.edu.cn, ‡zszhang@cse.cuhk.edu.hk
§wangyaxinus@gmail.com, ¶dingma@usc.edu, ‖jbhuang@xidian.edu.cn

***Abstract*—Information usually can be abstracted as a character sequence over a finite alphabet. With the advent of the era of big data, the increasing length and size of the sequences from various application fields (*e.g.*, biological sequences) result in the classical NP-hard problem, searching for the *Multiple Longest Common Subsequences* of multiple sequences (*i.e.*, *MLCS* problem with many applications in the areas of bioinformatics, computational genomics, pattern recognition, etc.), becoming a research hotspot and facing severe challenges. In this paper, we firstly reveal that the leading dominant-point-based *MLCS* algorithms are very hard to apply to long and large-scale sequences alignments. To overcome their defects, based on the proposed problem-solving model and parallel topological sorting strategies, we present a novel efficient parallel *MLCS* algorithm. The comprehensive experiments on the benchmark datasets of both random and biological sequences demonstrate that both the time and space complexities of the proposed algorithm are only linearly related to the dominants from aligned sequences, and that the proposed algorithm greatly outperforms the existing state-of-the-art dominant-point-based *MLCS* algorithms, and hence it is very suitable for long and large-scale sequences alignments.**

***Keywords*—Multiple Longest Common Subsequences (MLCS); Irredundant Common Subsequence Graph (ICSG); Parallel Collection Chain (PCC); ICSG-PCC Model; Parallel Algorithm**

I. Introduction

Information usually can be abstracted as a character sequence over a finite alphabet Σ. Searching for the *Multiple Longest Common Subsequences* of multiple sequences (*MLCS* for short, where the *Longest Common Subsequence* of two sequences *LCS* is a special case of the *MLCS*) over a finite alphabet Σ is a classical NP-hard problem [1] and has found many significant applications in the areas such as bioinformatics, computational genomics, pattern recognition, etc. For example, in bioinformatics, the sequence is the most basic mathematical model, which can describe the primary structure of the nucleic acid and protein molecules. Searching for their *LCS*s/*MLCS*s from the biological sequences is an important means of the identification of the sequence similarity, which can be used as the gene discovery, the construction of an evolutionary tree, the evidence of the species' common origin [2], etc. With the successful implementation of the Human Genome Project and the advent of the era of big data, the length and size of biological sequences and other types of sequences from various fields are growing explosively and exponentially [3]. Therefore, designing more efficient *LCS/MLCS* algorithms for long and large-scale sequences alignments is becoming a more and more important research topic and facing severe challenges.

In the past forty years, in order to tackle the NP-hard problem, various types of relatively efficient serial and parallel *LCS/MLCS* algorithms (e.g., [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]) have been proposed, which can be divided into two categories: classical dynamic programming and dominant-point-based algorithms. It has been demonstrated that the leading dominant-point-based *LCS/MLCS* algorithms have an overwhelming advantage over classical dynamic programming ones due to their greatly reducing the size of search space by orders of magnitude [3]. In particular, *FAST_LCS* [11] and *Quick-DPPAR* [3] are the most efficient parallel *MLCS* algorithms among dominant-point-based algorithms available. However, it has been proven in this paper by theory and verified by experiments that these leading dominant-point-based algorithms suffer severely from many unnecessary and redundant storage, computation, comparison and deletion of multi-dimensional matched points. Therefore, this kind of algorithms is essentially inapplicable to long and large-scale sequences alignments. To overcome the defects of these algorithms and efficiently tackle long and large-scale sequences alignments, we established a new problem-solving model and proposed a novel fast and memory efficient parallel *MLCS* algorithm. Our main contributions are as follows:

- We introduce several key concepts: *irredundant common subsequence graph* (*ICSG*), *parallel collection*, and *parallel collection chain* (*PCC*); then we propose a new effective and efficient problem-solving model *ICSG-PCC*;
- Based on the model, we present a novel fast and memory efficient parallel *MLCS* algorithm, which is composed of the parallel construction of the *ICSG*

of aligned sequences and the forward/backward parallel topological sorting schemes. Unlike the existing dominant-point-based *MLCS* algorithms, our algorithm not only eliminates the needs for comparison, storage and computation of the massive multi-dimensional matched points, but also needs no multiple backtracking processes, resulting in finding all *MLCS*s at once with a very low cost of time and space;

- We also analyze the time and space complexities of the proposed parallel algorithm, which are only linear to the number of dominants from aligned sequences and only relevant to it;
- We validate the proposed parallel *MLCS* algorithm on the DNA and amino acid sample sequences from real biological ncbi and dip databases [14], [15] and random synthetic sequences, and make comparisons in time and space performance between the proposed algorithm and the existing state-of-the-art dominant-point-based algorithms: *FAST_LCS* and *Quick-DPPAR*. Theoretical analysis and experimental results show that our algorithm not only greatly outperforms existing state-of-the-art algorithms, but also is very suitable for long and large-scale sequences alignments.

The rest of this paper is organized as follows. Section II gives a formal definition of the *MLCS* problem and briefly reviews the related work on dynamic programming and dominant-point-based algorithms. Section III discusses the defects of the leading dominant-point-based *MLCS* algorithms. Section IV elaborates the proposed problem-solving model *ICSG-PCC*. Section V presents a novel parallel *MLCS* algorithm based on the proposed model and a detailed analysis of the time and space complexities of the algorithm. Comprehensive experiments are conducted in Section VI. Finally, Section VII concludes the research.

II. Preliminaries and related work

A. Formal Definitions of the LCS/MLCS Problems

The subsequence of a given sequence over a finite alphabet Σ is the one that can be obtained by deleting zero or more (not necessarily consecutive) characters from the sequence. Let $X = x_1x_2...x_n$ and $Y = y_1y_2...y_m$ be two sequences with lengths n and m, respectively, over a finite alphabet Σ, *i.e.*, $x_i, y_i \in \Sigma$, and the longest common subsequence (*LCS*) problem is to find out one or all of the longest common subsequences of X and Y. Similarly, the multiple longest common subsequence (*MLCS*) problem is to find out one or all of the longest common subsequences of d ($d \geq 3$) sequences with a length n, respectively. Obviously, the *LCS* is a special case of the *MLCS*.

Note that, given d sequences over a finite alphabet Σ, there exists more than one *MLCS* in general. For example, given three sequences, $S_1 = GTACTAGC$, $S_2 = ACTGTCAG$ and $S_3 = TCAGTGCA$ over a finite alphabet $\Sigma = \{A, C, G, T\}$, there are 4 *MLCS*s with a length of 4, which are $MLCS_1 = GTCA$, $MLCS_2 = ATGC$, $MLCS_3 = CTGC$ and $MLCS_4 = TCAG$, respectively.

B. Preliminaries and Related Algorithms

Existing *LCS/MLCS* algorithms can be divided into two categories: classical dynamic programming and dominant-point-based algorithms. Depending on whether parallelized or not, they can also be categorized into two types: serial and parallel algorithms.

1) *Classical Dynamic Programming Algorithms*: This kind of algorithms are based on dynamic programming methods [4], [5]. In the simplest case, given two sequences $X = x_1x_2...x_n$ and $Y = y_1y_2...y_m$ with lengths n and m, respectively, over a finite alphabet $\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_k\}$, where $X[i] = x_i, Y[j] = y_j, x_i, y_j \in \Sigma$, $1 \leq i \leq n$ and $1 \leq j \leq m$, a dynamic programming algorithm iteratively constructs an $(n+1)*(m+1)$ *score matrix* L, in which $L[i, j]$ is the length of an LCS between two prefixes $X' = x_1x_2...x_i$ and $Y' = y_1y_2...y_j$ of X and Y, and can be calculated below by (1) :

$$L[i,j] = \begin{cases} 0 & \text{if } i \text{ or } j = 0 \\ L[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \\ max(L[i-1, j], L[i, j-1]) & \text{if } X[i] \neq Y[j] \end{cases} \quad (1)$$

Once the score matrix L of the sequences X and Y is calculated, all of its *LCS*s can be obtained by tracing back from the end element $L[n, m]$ to the starting element $L[0, 0]$. Both the time and space complexities of this algorithm are all $O(mn)$. In general, given d sequences S_1, S_2, ..., and S_d with a length n, the matrix L can be naturally extended to d dimensions suitable for the *MLCS* problem, in which the element $L[i_1, i_2, ..., i_d]$ can be calculated by (2) in a similar manner to (1) with respect to both the time and space complexities of $O(n^d)$.

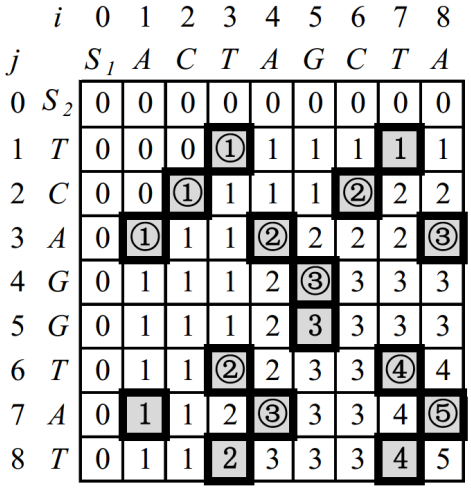
$$L[i_1,i_2,..,i_d] = \begin{cases} 0 & \text{if } \exists i_j = 0, 1 \leq j \leq d \\ L[i_1-1, i_2-1,.., i_d-1]+1 & \text{if } S_1[i_1] = ... = S_d[i_d] \\ max(\bar{L}) & \text{otherwise} \end{cases} \quad (2)$$

where $\bar{L} = \{L[i_1 - 1, i_2, i_3, ..., i_d], L[i_1, i_2 - 1, i_3, ..., i_d], ..., L[i_1, i_2, ..., i_{d-1}, i_d - 1]\}$.

Fig. 1 visualizes the score matrix L of two sequences $S_1 = ACTAGCTA$ and $S_2 = TCAGGTAT$ over the alphabet $\Sigma = \{A, C, G, T\}$ and the process of extracting an *LCS*=$CAGTA$ from the score matrix L of the sequences S_1 and S_2.

To further reduce time and space complexities of dynamic programming algorithms, various improved dynamic programming *LCS/MLCS* algorithms [6], [8], [16], etc. have been proposed. For example, Hirschberg [6] presented a new *LCS* algorithm based on the divide-and-conquer approach, which reduces the space complexity to $O(m + n)$ and gives a better solution to the problem of long sequences; however, its time complexity remains to be $O(mn)$. Masek and Paterson [5] put forward an improved dynamic programming *LCS* algorithm for two sequences with a length n by using a fast computing method of sequences editing distance, whose worst time complexity is $O(n^2/logn)$. Unfortunately, most of the aforementioned algorithms only address the *LCS* problem of two sequences and have high time and space complexities.

2) *Dominant-point-based Algorithms*: In order to clearly illustrate dominant-point-based *LCS/MLCS* algorithms, we first introduce following definitions.



i		0	1	2	3	4	5	6	7	8
j		S_1	A	C	T	A	G	C	T	A
0	S_2	0	0	0	0	0	0	0	0	0
1	T	0	0	0	①	1	1	1	1	1
2	C	0	0	①	1	1	1	②	2	2
3	A	0	①	1	1	②	2	2	2	③
4	G	0	1	1	1	2	③	3	3	3
5	G	0	1	1	1	2	3	3	3	3
6	T	0	1	1	②	2	3	3	④	4
7	A	0	1	1	2	③	3	3	4	⑤
8	T	0	1	1	2	3	3	3	4	5

(a) The score matrix L of sequences S_1 and S_2.

i		0	1	2	3	4	5	6	7	8
j		S_1	A	C	T	A	G	C	T	A
0	S_2	0	0	0	0	0	0	0	0	0
1	T	0			①					
2	C	0		①				②		
3	A	0	①			②				③
4	G	0					③			
5	G	0								
6	T	0			②				④	
7	A	0				③				⑤
8	T	0								

(b) An $LCS = CAGTA$ extracted from L.

Fig. 1. The score matrix L of the two sequences $S_1 = ACTAGCTA$ and $S_2 = TCAGGTAT$ over a finite alphabet $\Sigma = \{A, C, G, T\}$ and an $LCS = CAGTA$ extracted from L.

Definition 1: For a set of sequences $T = \{S_1, S_2, ..., S_i, ..., S_d\}$ over a finite alphabet Σ, $1 \leq i \leq d$, and $|S_i| = n$. Let $S_i[p_i] (S_i[p_i] \in \Sigma)$ denote the p_ith ($p_i \in \{1, 2, ..., n\}$) character in the sequence S_i. The point $p = (p_1, p_2, ..., p_d)$ is called a *matched point* of the sequences set $T = \{S_1, S_2, ..., S_i, ..., S_d\}$, if and only if $S_1[p_1] = S_2[p_2] = ... = S_i[p_i] = ... = S_d[p_d] = \sigma (\sigma \in \Sigma)$, where σ is the character corresponding to the matched point p.

Definition 2: For two matched points, $p = (p_1, p_2, ..., p_d)$ and $q = (q_1, q_2, ..., q_d)$, of the sequences $T = \{S_1, S_2, ..., S_i, ..., S_d\}$, we say that $p = q$ if and only if $p_i = q_i$ for $i = 1, 2, ..., d$. If $\forall i$, $p_i \leq q_i$ and $p \neq q$, we say that p weakly dominates q, which is denoted as $p \preceq q$, where p is referred to as a *dominating point* (*dominant* for short) and q as a *dominated point or successor* of p. If $\forall i$, $p_i < q_i$, we say that p strongly dominates q, which is denoted as $p \prec q$. Further, if there is no matched point $r = (r_1, r_2, ..., r_d)$ for $T = \{S_1, S_2, ..., S_i, ..., S_d\}$ such that $p \prec r \prec q$, we say that q is *an immediate successor* of p and that p is *an immediate predecessor* of q.

Definition 3: A matched point $p = (p_1, p_2, ..., p_d)$ is called the *kth dominant* (the *k-level dominant* for short), if $L[p_1, p_2, ..., p_d] = k$ (see (1) and (2)). The set of all the kth dominants is denoted as D^k, and the set of all dominants from the sequences set T as D.

Fig. 1 visualizes all the matched points (those greyed) and dominants (those circled) of the above sequences S_1 and S_2, respectively, where the regions of the same entry values in L are bounded by thick contours. And the circled numbers of the dominants indicate their corresponding kth dominants. Fig. 1 clearly shows that a dominant must be a match point, but not vice versa, and that the size of the dominant set is smaller than that of the matched points, which have been fully proven by [3], [7], [11].

The main motivation of dominant-point-based *LCS/MLCS* algorithms is to effectively reduce the high time and space complexities of dynamic programming *LCS/MLCS* algorithms. The dominant-point-based *LCS/MLCS* algorithms only need to calculate the dominants instead of all the elements in the score matrix L of the sequences set T over Σ. Based on the kth dominants $D^k (0 \leq k \leq |LCS/MLCS| - 1)$, the algorithm recursively calculates the $(k+1)$th dominants D^{k+1} until $k = |LCS/MLCS| - 1$ (the process is denoted as $D^k \rightarrow D^{k+1}$ for short). Finally, all of the *LCSs/MLCS*s of the sequences set T can be obtained by tracing back the elements from the last to the 0th dominant set. Because the size of the dominants set D is far smaller than that of the matrix L, *i.e.*, $|D| \ll |L|$, both theoretical analysis and experimental results have shown that the dominant-point-based *LCS/MLCS* algorithms are overwhelmingly faster than classical dynamic programming algorithms [3].

Hunt and Szymanski [7] first proposed a dominant-point-based *LCS* algorithm with the time complexity $O((r+n)logn)$, where r is the number of all dominants of two sequences with a length n. Afterwards, a variety of dominant-point-based *LCS/MLCS* algorithms have been presented [8], [16], etc. To further improve the time performance of the algorithms, some parallel dominant-point-based *LCS* algorithms [9], [10] and *MLCS* algorithms [3], [17], [11], etc. were proposed. Korkin [17] first proposed a new parallel *MLCS* algorithm with time complexity $O(s|D|)$, where $|D|$ is the total number of dominants of d sequences and s is the size of the alphabet Σ. Chen et. al [11] presented an efficient parallel *MLCS* algorithm over the alphabet $\Sigma = \{A, C, G, T\}$, *FAST_LCS* based on the proposed pruning rules. Wang et al. [3] developed an efficient parallel *MLCS* algorithm *Quick-DPPAR*, which indicates that the proposed algorithm has reached a near-linear speedup with respect to its serial algorithm *Quick-DP*. It is worth mentioning that Yang, Li et al. [12], [13] made new attempts to develop efficient parallel algorithms on GPUs for the *LCS* problem and on the Cloud Computing platform for the *MLCS* problem, respectively. But regretfully, the algorithm [12] is not suitable for the general *MLCS* problem, with the problem of a large amount of synchronous cost of the algorithm [13] remaining to be solved. For tackling large-scale *MLCS* problems in practice, Yang et al. [18] presented a new progressive algorithm *Pro-MLCS* with its efficient parallelization, which can find an approximate solution quickly.

Parallel dominant-point-based *MLCS* algorithms are currently a relatively better solution for the *MLCS* problem.

However, from the analysis in Section III, we will show that such algorithms are facing serious challenges with the explosive expansion of the length and size of sequences from varieties of application fields due to their fundamental flaws.

III. The dominant-point-based *MLCS* algorithm framework and its performance bottlenecks

A. Algorithm Framework

To analyze the performance of the dominant-point-based *MLCS* algorithms, we briefly introduce its algorithm framework first as follows:

1) *Constructing a directed acyclic graph of dominants.* With a forward iteration procedure (see Fig.2), the $(k+1)$th dominants D^{k+1} is calculated based on the kth dominants D^k (see Definition 3), and this procedure is denoted as $D^k \rightarrow D^{k+1}$, where $0 \leq k \leq |MLCS| - 1$. As a result, a directed acyclic graph consisting of all of the *MLCS*s of the aligned sequences set T (*MLCS-DAG*, for short) is constructed level by level in two steps:
   Step 1: Based on the kth dominants D^k, all of the immediate successors (see Definition 2) of each dominant from D^k are calculated, denoted as D^{k+1}_{init}.
   Step 2: There exist massive repeated/duplicate dominants and dominated points in D^{k+1}_{init}, collectively called redundant points for short, which not only have no contribution to the *MLCS*s of the aligned sequences set T, but also will seriously affect the time and space performance of the procedure $D^k \rightarrow D^{k+1}$, where $0 \leq k \leq |MLCS| - 1$. Hence, the operation of eliminating the massive redundant points is required, which is denoted as *Minimal*(D^{k+1}_{init}) (*Minimal*() for short). In general, the operation *Minimal*() is performed by the comparison among the points in D^{k+1}_{init} dimension by dimension, resulting in the non-redundant dominants D^{k+1} over D^{k+1}_{init}.
2) *Computing MLCS.* All of the *MLCS*s of *T* are then found by tracing back through the constructed *MLCS-DAG* from the final node to the source node iteratively.

B. Performance Analysis of Dominant-point-based MLCS Algorithms

We first give an example to visualize the performance bottlenecks of the general dominant-point-based *MLCS* algorithms, and then make a further analysis.

Example 1: Given two sequences $S_1 = ACTAGCTA$ and $S_2 = TCAGGTAT$ over the alphabet $\Sigma = \{A, C, G, T\}$, find all of the *MLCS*s of aligned sequences S_1 and S_2.

We illustrate the construction of the MLCS-DAG of S_1 and S_2 in Fig. 2. Based on the above dominant-point-based *MLCS* algorithm, we start constructing the *MLCS-DAG* of S_1 and S_2. First of all, let $k = 0$, $D^0 = \{(0, 0)\}$, where $(0, 0)$ is introduced as a dummy source point or 0-level dominant of the *MLCS-DAG*. By Definition 2, all of the immediate successors ((1, 3), (2, 2), (5, 4), and (3, 1)) from D^0 are calculated, *i.e.*, D^1_{init} = {(1, 3), (2, 2), (5, 4), (3, 1)}. The operation *Minimal*() is then performed so as to eliminate the dominated point $(5, 4)$ which is redundant, accomplishing the procedure $D^0 \rightarrow D^1$, leading to the 1-level dominants D^1 = {(1, 3), (2, 2), (3, 1)}. Repeating the procedure $D^k \rightarrow D^{k+1}$, D^2, D^3, D^4, and

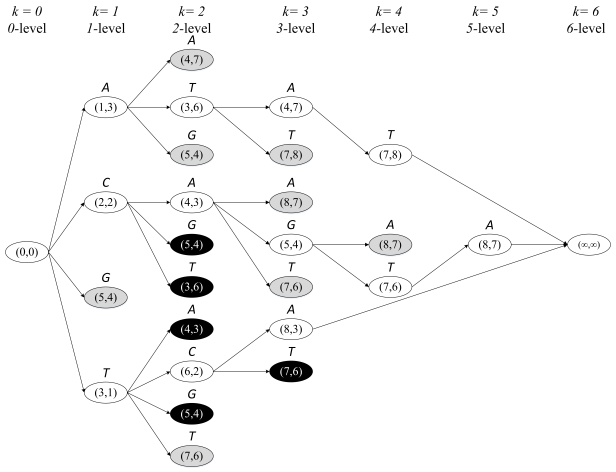


Fig. 2. The constructed *MLCS-DAG* of the sequences $S_1 = ACTAGCTA$ and $S_2 = TCAGGTAT$ by the general dominant-point-based algorithms, in which the redundant points with black and grey will be deleted by *Minimal*() procedure.

D^5 of 2-level to 5-level dominants can be obtained in turn, where D^2 = {(3, 6), (4, 3), (6, 2)}, D^3 = {(4, 7), (5, 4), (8, 3)}, D^4 = {(7, 8), (7, 6)}, and D^5 = {(8, 7)}, respectively. Since $D^6_{init} = \emptyset$, the procedure $D^k \rightarrow D^{k+1}$ ends. Note that the dummy sink point $(\infty, \infty, ..., \infty)$ in the *MLCS-DAG* is introduced as the immediate successor of the points without an immediate successor. The constructed *MLCS-DAG* of the sequences S_1 and S_2 is shown in Fig. 2, where the black points (repeated points) and the gray points (dominated points) would be deleted after the operation *Minimal*(). Then, based on the *MLCS-DAG*, all of the two *MLCS*s of S_1 and S_2 can be obtained by tracing back twice from the final node (∞, ∞) to the source node $(0, 0)$, which are *MLCS*$_1$ = *CAGTA* and *MLCS*$_2$ = *TAGTA*, respectively.

Fig. 2 clearly shows that there are a mass of redundant points in the D^{k+1}_{init} of each iteration $D^k \rightarrow D^{k+1}$, which would result in high time consumption due to tremendously repeated calculation, comparison among d-dimension points, and deletion of the massive redundant points. For example, in Fig. 2, the point (5,4) is repeatedly calculated 5 times and compared with the other points 12 times in the operation *Minimal*() of the process $D^k \rightarrow D^{k+1}$.

To further identify the performance bottlenecks of the general dominant-point-based *MLCS* algorithms, we analyze the statistical data of various types of the redundant points of the *MLCS-DAG* from various lengths and alphabet size sequences, with part of the statistical results shown in Fig. 3, from which we draw the following conclusions:

1) In each iteration, procedure $D^k \rightarrow D^{k+1}$ generates a great number of redundant points, leading to an excessive computational time in the operation *Minimal*().
   The statistical results show that in D^{k+1}_{init}, there exist two types of a significant number of redundant points (denoted as N_{redu}), *i.e.*, repeated points (denoted as N_{repeat}) and dominated points/successors (denoted as N_{suc}). Letting $N = |D^{k+1}_{init}|$, the average ratio of the redundant points N_{redu} to N reaches 59%, and the ratio can be up to 79%. These redundant points will result in an excessive

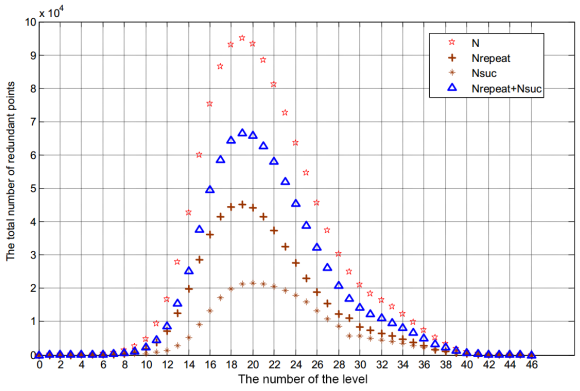


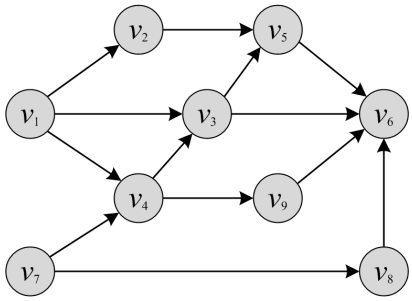
Fig. 3. The distributions of various redundant points in the kth level of the *MLCS-DAG* from the 5 sequences over the alphabet $\Sigma = \{A, C, G, T\}$ with the length of 75.

computational time in the operation *Minimal*(). Moreover, tremendous comparisons in d dimensions among N points are inevitable to delete the massive redundant points in the operation *Minimal*() of the process $D^k \rightarrow D^{k+1}$. For d sequences with the length $n(d \geq 3)$, it is proved that the time for the comparisons among N d-dimensional points making comparisons dimension by dimension in a brute-force manner is $O(dN^2)$ [19], [20]. Even if the divide-and-conquer strategy is adopted, $O(dNlog^{d-2}n)$ comparisons are still needed [3].

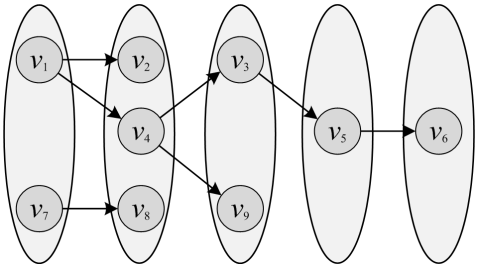
2) The constructed *MLCS-DAG* graph contains a large number of unrelated points not leading to any *MLCS*s , called non-critical points.
Our statistical results also show that the constructed *MLCS-DAG* graph contains a large number of unrelated points with *MLCS*s of aligned sequences. More specifically, the ratio of $|K|$ (the total number of the key points that would contribute the *MLCS*s of the aligned sequences) in *MLCS-DAG*) to $|D|$ (the total number of the points in *MLCS-DAG*) range s from only one tenth to one hundred thousandth. Moreover, the larger the d, n and $|\Sigma|$, the larger the ratio of $|K|/|D|$ is. This also results in massive non-critical points residing in the constructed *MLCS-DAG* graph and introduces another major time and space bottleneck when backtracking to find its *MLCS*s.
Above all, since *Minimal*() is the basic and key operation in the process $D^k \rightarrow D^{k+1}$ of the general dominant-point-based *MLCS* algorithms, and by the above argument, it is clear that the time complexity of the dominant-point-based *MLCS* algorithm is nonlinearly related to d and $|D|$. In addition, the length of the *MLCS* (denoted as $|MLCS|$) is proportional to the sequence length n, and $|D^k|$ tends to grow explosively in the range $1 \leq k \leq |MLCS|/2$. Therefore, both the above analysis and our comprehensive experimental results (see Section VI) show that *the dominant-point-based MLCS algorithms are not applicable to long and large-scale sequences alignments*.

IV. A NOVEL PROBLEM-SOLVING MODEL: *ICSG-PCC*

To overcome the bottlenecks of the dominant-point-based *MLCS* algorithms, we propose a novel model to solve the *MLCS* problem. Some necessary definitions from the graph theory and discrete mathematics are first introduced in Subsection IV-A, and then the proposed novel problem-solving graphical model for the *MLCS* problem will be discussed in detail in Subsection IV-B.



(a) A directed acyclic graph G with nine vertices $v_1 \sim v_9$.



(b) Five parallel collections ($\{v_1, v_7\}$, $\{v_2, v_4, v_8\}$, $\{v_3, v_9\}$, $\{v_5\}$ and $\{v_6\}$) and a parallel collection chain ($Sub = \{\{v_1, v_7\}, \{v_2, v_4, v_8\}, \{v_3, v_9\}, \{v_5\}, \{v_6\}\}$) on G.

Fig. 4. Parallel collections and a parallel collection chain on G.

A. Key Concepts

Definition 4: Given a directed acyclic graph $G = \langle V, \preceq \rangle$, where V is the set of vertices in graph G, and $\preceq$ is the partial order set of V, if any two elements in a subset C of V are independent, *i.e.*, there exists no *partial order relationship* [19], [20], C is called an *anti-chain* [21].

Definition 5: For any two anti-chains C_i and C_j in the anti-chain set $Sub = \{C_i, C_j, ..., C_k\}$ of $V(1 \leq i, j \leq k)$, if they satisfy the following conditions: 1) There exist a nonempty the edges set E from C_i to C_j; 2) the edges set from C_j to C_i is empty; and 3) $C_i \cap C_j = \emptyset$, C_i and C_j are called the *parallel collections* [21].

Definition 6: Given an anti-chain set $Sub = \{C_{i_1}, C_{i_2}, ..., C_{i_k}\}(1 \leq i_1 < i_2 < ... < i_k \leq m)$, which is the subset of the anti-chain set $S = \{C_1, C_2, ..., C_i, ..., C_m\}$ of V, iff the subset Sub conforms the following two conditions: 1) Any two adjacent elements C_{i_t} and $C_{i_{t+1}}(t = 1, 2, ..., k-1)$ are parallel collections; and 2) the union of all the elements in Sub is equal to V, Sub is called a *parallel collection chain* on G [21].

Definition 7: For a directed acyclic graph $G = \langle V, \preceq \rangle$, *topological sorting* is to find an overall order [19], [20] of the vertices in G from the partial order $\preceq$, whose result is a linear order, *i.e.*, forming a parallel collection chain in G.

Example 2: A directed acyclic graph $G = \langle V, \preceq \rangle$, where $V = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8, v_9\}$, and $\preceq = \{\langle v_1, v_2 \rangle, \langle v_1, v_3 \rangle, \langle v_1, v_4 \rangle, \langle v_2, v_5 \rangle, \langle v_3, v_5 \rangle, \langle v_3, v_6 \rangle, \langle v_4, v_3 \rangle,$

$\langle v_4, v_9\rangle, \langle v_5, v_6\rangle, \langle v_7, v_4\rangle, \langle v_7, v_8\rangle, \langle v_8, v_6\rangle, \langle v_9, v_6\rangle\}$, is shown in Fig. 4(a).

By the topological sorting algorithm [20], *i.e.*, iteratively performing the following two steps: 1) outputting the vertices with the in-degree being 0; and 2) deleting the edges connecting the vertices.), the graph G shown in Fig. 4(a) is converted into Fig. 4(b). Based on the above Definitions 4 to 7, it is clear that the vertices v_1 and v_7 shown in Fig. 4(b) form an *anti-chain*, and that the vertices sets $\{v_1, v_7\}$, $\{v_2, v_4, v_8\}$, $\{v_3, v_9\}$, $\{v_5\}$ and $\{v_6\}$ constitute five *parallel collections*, respectively, each of which is an *anti-chain*, with a *parallel collection chain* of G, *i.e.*, $Sub = \{\{v_1, v_7\}, \{v_2, v_4, v_8\}, \{v_3, v_9\}, \{v_5\}, \{v_6\}\}$.

B. Problem-Solving Model: ICSG-PCC

By the above argument, it is easy to see that the *MLCS* problem can be divided into three sub-problems to find the solution: 1) Determining the partial order relationship between the points contributing or possibly contributing to an *MLCS* in the problem-solving graphical model *MLCS-DAG*; 2) Sorting and layering the points in the *MLCS-DAG*; and 3) Searching for the longest paths of the *MLCS-DAG*.

Aiming at eliminating the aforementioned defects of the dominant-point-based algorithms and tackling the long and large-scale sequences alignments, we propose a novel problem-solving graphical model. Its basic idea and principle are as follows.

Firstly, all the *MLCS*s of the sequences set $T(|T| = d)$ are constructed by its relevant matched points, *i.e.*, dominants. Supposing that D denotes the set of all the dominants in T and that the immediate predecessor-successor relationships between the points constitute a partial order set $\preceq$ defined on D, we can represent the relationships by a directed acyclic graph $G = (D, \preceq)$ with all edges' lengths being 1. For the convenience of implementing the algorithm, two dummy d-dimensional points $(0, 0, ..., 0)$ (the source point) and $(\infty, \infty, ..., \infty)$ (the sink point) are introduced into D, with all the other points in D being the successors of $(0, 0, ..., 0)$ and the predecessors of $(\infty, \infty, ..., \infty)$. Such a graph is exactly an *MLCS-DAG*, and finding an *MLCS* can now be regarded as identifying the longest path from the source point $(0, 0, ..., 0)$ to the sink point $(\infty, \infty, ..., \infty)$, and vice versa.

It is worth noting that the constructed *MLCS-DAG* should be efficient, *i.e.*, it contains no redundant points, and each d-dimensional point in the graph needs to be computed and stored only once without comparing with other massive points dimension-by-dimension. The significant differences between the two *MLCS-DAG*s lie in the fact that our constructed *MLCS-DAG* contains no redundant points and operations, and that any path of the *MLCS-DAG* corresponds to a common subsequence of the sequences set T, *i.e.*, an *Irredundant Common Subsequence Graph*, abbreviated to the *ICSG* in this paper for solving the above sub-problem 1) of the *MLCS* problem.

Secondly, each layer's dominants set $D^k(1 \leq k \leq |MLCS| - 1)$ in the *MLCS-DAG* is exactly a *parallel collection*, and the *MLCS-DAG* constructed by the algorithms actually contains a hidden *parallel collection chain* of the dominants set D. Meanwhile, the length of the *parallel collection chain*, *i.e.*, the number of parallel collections in the chain, is exactly equal to $|MLCS|$+1 of the sequences set T.

Therefore, to efficiently solve the sub-problem 2) of the *MLCS* problem, based on the above definitions and our graphical model *ICSG*, a parallel forward topological sorting (from the single source point $(0, 0, ..., 0)$ to the sink point $(\infty, \infty, ..., \infty)$ of the *ICSG*) without comparison operations of massive d-dimensional points is adopted so that all of the points in the *ICSG* are efficiently sorted and layered, resulting in the construction of a parallel collection chain of the *ICSG*. Similarly, to efficiently solve the sub-problem 3) of the *MLCS* problem, a backward topological sorting (from the sink point $(\infty, \infty, ..., \infty)$ to the source point $(0, 0, ..., 0)$ of the *ICSG*) without comparison operations of massive d-dimensional points is performed to delete all of the non-critical points residing in the *ICSG* and to achieve all of the *MLCS*s of the sequences set T at once. Since above solutions of sub-problems 2) and 3) are based on the introduced key concepts, the parallel collection and the parallel collection chain (*PCC*), the sub-problems solving model is abbreviated to the *PCC*.

In a word, our *MLCS* problem-solving model proposed in this paper is composed of two sub-models: *ICSG* and *PCC*, referred to as the *ICSG-PCC* model, in which the *ICSG* sub-model is used to solve the sub-problem 1) and the *PCC* sub-model is used to solve sub-problems 2) and 3).

V. A Novel Parallel MLCS Algorithm Based on the *ICSG-PCC* Model

In this section, we elaborate our novel parallel *MLCS* algorithm based on the *ICSG-PCC* model. We begin with introducing the primary data structures used in our algorithm in Subsection V-A. Then we propose a novel serial algorithm in Subsection V-B and show a simple example of the proposed algorithm in Subsection V-C. We further parallelize the algorithm in Subsection V-D. Finally, we analyze the time and space complexities of our parallel *MLCS* algorithm in Subsection V-E.

A. The Primary Data Structures

The primary data structures designed or employed in our proposed algorithms are as follows:

1) ***Successor table ST***. According to the strategy of [11], the successor tables $\{ST_1, ST_2, ..., ST_d\}$ of the sequences set $T = \{S_1, S_2, ..., S_d\}$ will be built to support the compression of the data and quick search of the immediate successors of the points, *i.e.*, given a sequence $S_l = x_1, x_2, ..., x_n$ from the sequences set T over a finite alphabet $\Sigma = (\sigma_1, \sigma_2, ..., \sigma_k)$, its successor table ST_l of the sequence S_l is a two-dimensional array, where $ST_l[i, j]$ indicating the element of the ith row and the jth column in the successor table ST_l is defined as below:

$$ST_l[i, j] = min\{m | x_m = \sigma_i, m \geq 1, m \geq j, 1 \leq i \leq |\Sigma|, 0 \leq j \leq n\} \quad (3)$$

From (3), we can see that $ST_l[i, j]$ denotes the minimal subscript position m of the sequence S_l according to σ_i after position j when $x_m = \sigma_i$, see Fig. 5.
It has been proven in [11] that the set S_{suc} of immediate successors of a d-dimensional point $p = (p_1, p_2, ..., p_d)$ can be obtained efficiently in $O(d|\Sigma|)$. For

Algorithm 1 *Top_MLCS*($\{S_1, S_2, ..., S_d\}, \Sigma$)

1: Preparation of successor tables $\{ST_1, ST_2, ..., ST_d\}$ of $T = \{S_1, S_2, ..., S_d\}$
2: (*DM, ID, maxIndex*) ← *Construct_ICSG*($\{ST_1, ST_2, ..., ST_d\}, \Sigma$)
3: (*precursor, tlevel, maxLevel*) ← *ForwardTopSort(DM, ID, maxIndex)*
4: *BackwardTopSort(DM, precursor, tlevel, maxLevel)*
5: **return** the *MLCS*s of sequences set $T = \{S_1, S_2, ..., S_d\}$.

a d-dimensional point p, the operation of producing its S_{suc} can be characterized in (4).

$$S_{suc} = \{(ST_1[i', p_1], ST_2[i', p_2], ..., ST_d[i', p_d])\}$$
$$s.t. 1 \leq i' \leq |\Sigma|, \forall ST_l[i', p_i] \neq \emptyset, 1 \leq l, i \leq d \quad (4)$$

2) ***DM***. A bidirectional hash table, wherein the *Point* represents a d-dimensional point and the *Index* is a serial number corresponding to the point, is designed to store and efficiently lookup all the points of the constructed graph *ICSG* from the sequences set $T = \{S_1, S_2, ..., S_d\}$ in a two-way manner, *i.e.*, from a serial number of the point to its d-dimensional coordinates, and vice versa.
3) ***Queue***. A temporary queue is used to store the immediate successors of the existing points in the *ICSG*.
4) ***ID***. An object of the class *ArrayList* (a linked array) is designed to record the in-degree to each point in the *ICSG*.
5) ***tlevel***. An object of the class *ArrayList* is employed to store the corresponding number of forward levels of points in the *ICSG*.
6) ***precursor***. An object of the class *ArrayList* records the immediate predecessors of the points in the *ICSG*.

B. The Proposed Serial Algorithm Top_MLCS

The framework of the proposed serial algorithm *Top_MLCS* based on the model *ICSG-PCC* is as follows:

1) **Preparation of successor tables.** The successor tables $ST_1 \sim ST_d$ corresponding to d aligned sequences of T are first constructed to find all immediate successors of dominants quickly using (3).
2) **Construction of *ICSG*.** Based on the above constructed successor tables $ST_1 \sim ST_d$ of T and the model *ICSG*, the irredundant common subsequence graph (*ICSG*) of T is constructed step by step from the single dummy source point $(0, 0, ..., 0)$ to the single dummy sink point $(\infty, \infty, ..., \infty)$.
3) **Optimization of the *ICSG***. With a forward topological sorting, a parallel collection chain of the *ICSG* is efficiently obtained. And then with a backward topological sorting, all of the non-critical points in the *ICSG* are correctly and effectively identified and removed, resulting in an optimized *ICSG*.
4) **Obtaining all of the *MLCS*s of T**. Since the optimized *ICSG* does not contain any non-critical points, all *MLCS*s can be obtained by traversing any path of the optimized *ICSG* from the source point $(0, 0, ..., 0)$ to the sink point $(\infty, \infty, ..., \infty)$, and vice versa.

The pseudo codes of the proposed serial algorithm *Top_MLCS* based on the model *ICSG-PCC* are shown in Algorithms 1 $\sim$ 4.

Algorithm 2 *Construct_ICSG*($\{ST_1, ST_2, ..., ST_d\}, \Sigma$)

1: *Queue* ← {(0, 0, ..., 0)}
2: *indexOfPoint* ← 0
3: *DM* ← {⟨*indexOfPoint*, (0, 0, ..., 0)⟩}
4: *indexOfPoint* ← *indexOfPoint* +1
5: **while** *Queue* ≠ ∅ **do**
6: *q* = Pop(*Queue*)
7: **for** each v ← *Successor*(q) **do**
8: **if** $v \notin$ *DM* **then**
9: Push(*Queue*, v)
10: *DM* ← *DM* ∪ {⟨*indexOfPoint*, v⟩}
11: *ID*[v] ← 1
12: *indexOfPoint* ← *indexOfPoint* + 1
13: **else**
14: *ID*[v] ← *ID*[v] + 1
15: **end if**
16: **end for**
17: *DM* ← *DM* ∪ {⟨*indexOfPoint*, (0, 0, ..., 0)⟩}
18: **end while**
19: *maxIndex* ← *indexOfPoint* + 1
20: **return** (*DM, ID, maxIndex*)

Algorithm 3 *ForwardTopSort(DM, ID, maxIndex)*

1: D^0 ← {(0, 0, ..., 0)}
2: k ← 0
3: **for** i ← 0 to *maxIndex* **do**
4: *tlevel*[i] ← 0
5: *precursor*[i] ← ∅
6: **end for**
7: **while** $D^k \neq \emptyset$ **do**
8: D^{k+1} ← ∅
9: **for** $q \in D^k$ **do**
10: **for** each v ← *Successor*(q) **do**
11: *tlevel*[v] ← $k + 1$
12: **if** v is q's immediate successor **then**
13: *precursor*[v] ← [*precursor*[v]; q]
14: *tlevel*[v] ← $k + 1$
15: **end if**
16: *ID*[v] ← *ID*[v] - 1
17: **if** *ID*[v] == 0 **then**
18: D^{k+1} ← $D^{k+1} \cup q$
19: **end if**
20: **end for**
21: **if** q has no successor **then**
22: *precursor*[*maxIndex*] = *precursor*[*maxIndex*] ∪ q
23: **end if**
24: **end for**
25: k ← $k + 1$
26: **end while**
27: *maxLevel* ← $k - 1$
28: **return** (*precursor, tlevel, maxLevel*)

Algorithm 4 *BackwardTopSort(DM,precursor,tlevel,maxLevel)*

1: D^0 ← $\{(\infty, \infty, ..., \infty)\}$
2: k ← 0
3: **while** $D^k \neq \emptyset$ **do**
4: D^{k+1} ← ∅
5: **for** $q \in D^k$ **do**
6: **for** $p \in$ *precursor*[q] **do**
7: **if** *tlevel*[p]+$k \neq$ *maxLevel* **then**
8: kill p
9: **end if**
10: **end for**
11: **end for**
12: k ← $k + 1$
13: **end while**

Algorithm 2 *Construct_ICSG* responsible for the construction of the *ICSG* is mainly composed of the following two steps: 1) Initializing the relevant data structures *Queue* and

DM (lines 1 $\sim$ 3); and 2) Constructing the *ICSG* of the aligned sequences set $T = \{S_1, S_2, ..., S_d\}$ based on their successor tables $ST_1, ST_2, ..., ST_d$ and model *ICSG* (lines 4 $\sim$ 18).

Algorithm 3 *ForwardTopSort* accomplishes the forward topological sorting of the *ICSG* with the topological sorting algorithm. It sorts and layers all of the points from the point $(0, 0, ..., 0)$ to the point $(\infty, \infty, ..., \infty)$, and consists of the following steps: 1) Initializing the relevant data structures *tlevel* and *precursor* (lines 1 $\sim$ 6); and 2) Sorting and layering all of the points of the *ICSG* from the 0-level point set $D^0\{(0, 0, ..., 0)\}$ to the k-level point set $D^k\{(\infty, \infty, ..., \infty)\}$ level by level, $1 \leq k \leq |MLCS| - 1$, resulting in a parallel collection chain with a length $|MLCS|-1$ stored in the variable *maxLevel* (lines 7 $\sim$ 27).

Similarly, Algorithm 4 *BackwardTopSort* sorts inversely all of the points from D^k to $D^0(1 \leq k \leq |MLCS| - 1)$ of the *ICSG* with the topological sorting algorithm so that the non-critical points are identified and removed, resulting in an irredundant optimized *ICSG*, *i.e.*, in which any path is an *MLCS* of the aligned sequences. It mainly performs two steps: 1) Initializing the relevant data structures (lines 1 $\sim$ 2); and 2) identifying and killing all of the non-critical points in the *ICSG* from D^k to D^0 backward iteratively (lines 3 $\sim$ 13). It is noteworthy that our strategy of identifying and killing the non-critical points in the *ICSG* is based on our observation, and that for any non-critical points in the *ICSG*, the sum of the forward levels and the reverse levels obtained by performing algorithms *ForwardTopSort* and *BackwardTopSort*, respectively, is not equal to the length $|MLCS| + 1$ of aligned sequences (line 7). For lack of space, we do not prove it here but illustrate it by a simple example in Subsection V-C.

C. A Simple Example of Algorithm Top_MLCS

As discussed, the proposed serial algorithm *Top_MLCS* consists mainly of four procedures: preparation of successor tables, construction of the *ICSG*, optimization of the *ICSG*, and obtaining all the *MLCS*s of the sequences set T. In order to clearly explain the algorithm *Top_MLCS*, the following Example 3 is used to visualize it.

Example 3: Given the aligned sequences set $T = \{S_1 = ACTAGTGC, S_2 = TGCTAGCA, S_3 = CATGCGAT\}$ over the finite alphabet $\Sigma = \{A, C, G, T\}$, the execution steps of the algorithm *Top_MLCS* on T are as follows.

Step 1: *Preparation of Successor Tables*. With (3), the constructed successor tables ST_1, ST_2 and ST_3 corresponding to S_1, S_2, and S_3 of T are shown in Fig. 5.

Step 2: *Construction of ICSG.* Based on (4) and Algorithm 2 *Construct_ICSG*, the constructed *ICSG* of the sequences set T is shown in Fig. 6. Note that considering the storage efficiency of the *ICSG*, the serial numbers of dominants instead of their d-dimensional coordinates are saved in the *ICSG*. We can see that there are 16 points in the *ICSG*, including 14 dominants and two dummy points 0 (corresponding to the single source point $(0, 0, ..., 0)$) and 15 (corresponding to the single sink point $(\infty, \infty, ..., \infty)$). The tuples $\langle$ the serial number of a dominant, corresponding d-dimensional coordinates $\rangle$ and in-degrees of the dominants $0 \sim 15$ stored in the *DM* and *ID*, respectively, are as follows:

S_1 = A C T A G T G C

A	1	4	4	4	-	-	-	-	-
C	2	8	8	8	8	8	8	8	-
G	5	5	5	5	5	7	7	-	-
T	3	3	3	6	6	6	-	-	-

(a) Table ST_1

S_2 = T G C T A G C A

A	5	5	5	5	5	8	8	8	-
C	3	3	3	7	7	7	7	-	-
G	2	2	6	6	6	6	-	-	-
T	1	4	4	4	-	-	-	-	-

(b) Table ST_2

S_3 = C A T G C G A T

A	2	2	7	7	7	7	7	-	-
C	1	5	5	5	5	-	-	-	-
G	4	4	4	4	6	6	-	-	-
T	3	3	3	8	8	8	8	8	-

(c) Table ST_3

Fig. 5. The constructed successor tables ST_1, ST_2 and ST_3 corresponding to S_1, S_2 and S_3 of the aligned sequences set T.

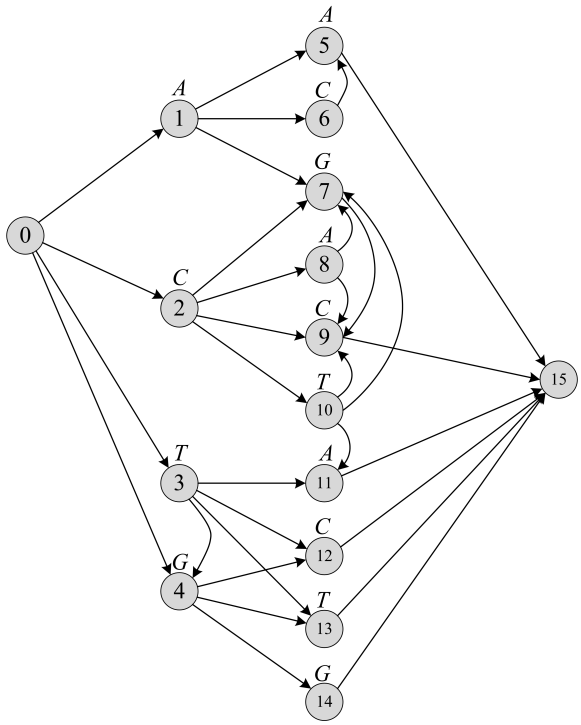


Fig. 6. The constructed *ICSG* of the sequences set T.

DM=$\{\langle 0, (0, 0, 0)\rangle, \langle 1, (1, 5, 2)\rangle, \langle 2, (2, 3, 1)\rangle, \langle 3, (3, 1, 3)\rangle, \langle 4,(5, 2, 4)\rangle, \langle 5,(4, 8, 7)\rangle, \langle 6,(2, 7, 5)\rangle, \langle 7,(5, 6, 4)\rangle, \langle 8,(4, 5, 2)\rangle, \langle 9, (8, 7, 5)\rangle, \langle 10, (3, 4, 3)\rangle, \langle 11, (4, 5, 7)\rangle, \langle 12, (8, 3, 5)\rangle, \langle 13, (6, 4, 8)\rangle, \langle 14, (7, 6, 6)\rangle, \langle 15, (\infty, \infty, \infty)\rangle\}$.

ID=$[0, 1, 1, 1, 2, 2, 1, 4, 1, 4, 1, 2, 2, 2, 1, 0]$.

Step 3: *Optimization of the ICSG.* This step is composed of two sub-procedures: a forward topological sorting and a

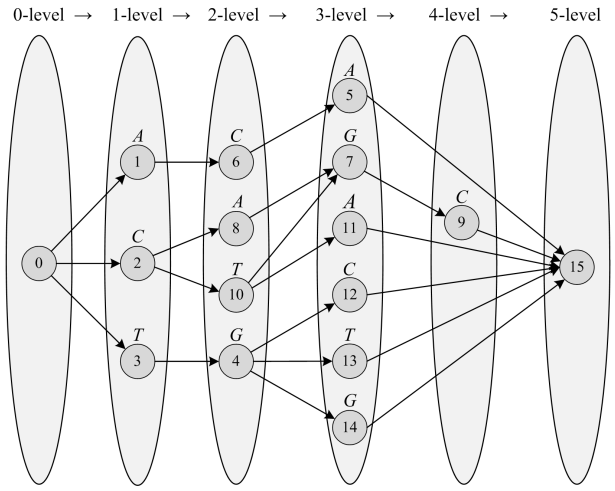


Fig. 7. The layered *ICSG* by Algorihtm 3 *ForwardTopSort*.

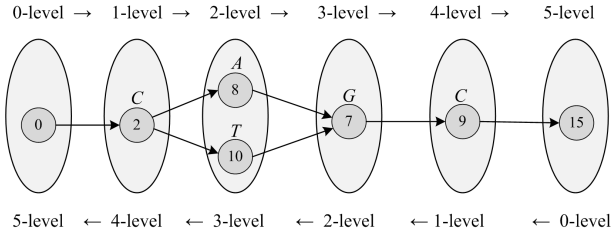


Fig. 8. The optimized *ICSG* by Algorithm 4 *BackwardTopSort*.

backward topological sorting described as follows.

Forward Topological Sorting. By executing Algorithm 3 *ForwardTopSort*, the constructed *ICSG* of the sequences set T is converted into Fig. 7.

As seen above, Fig. 7 shows the layered *ICSG* simplified by deleting all of the non-immediate predecessors of a point, *e.g.*, non-immediate predecessors 8 and 10 of the point 9 are deleted. What's more, all of the points in the layered *ICSG* are divided into 0-level to 5-level, respectively, forming five *parallel collection* and one *parallel collection chain*, and $|MLCS|$ of T obviously equals the max-level 5 minus 1. The levels of the dominants $0 \sim 15$ and their corresponding immediate predecessor lists stored in *tlevel* and *precursor* separately, are as follows:

tlevel=$[0, 1, 1, 1, 2, 3, 2, 3, 2, 4, 2, 3, 3, 3, 5]$.

precursor=$[\{\}, \{0\}, \{0\}, \{0\}, \{3\}, \{6\}, \{1\}, \{8, 10\}, \{2\}, \{7\}, \{2\}, \{10\}, \{4\}, \{4\}, \{4\}, \{5, 11, 12, 13, 14, 9\}]$.

Backward Topological Sorting. By executing Algorithm 4 *BackwardTopSort*, the final optimized *ICSG* of the sequences set T is shown in Fig. 8, in which we can easily see that the vast majority of the non-critical points (9 points out of 14 points which account for 64%) in the *ICSG* have been deleted.

Step 4: ***Obtaining all the MLCSs of T.*** With the final optimized *ICSG* shown in Fig. 8, each of its paths yields an *MLCS* of the sequences set T. Hence, two *MLCS*s of T, namely, $MLCS_1$ = *CAGC* and $MLCS_2$ = *CTGC*, are extracted, where $|MLCS| = 4$.

Algorithm 5 *Ptop_MLCS*$(\{S_1, S_2, ..., S_d\}, \Sigma)$

1: Parallelly construct successor tables $\{ST_1, ST_2, ..., ST_d\}$ of sequences set $T = \{S_1, S_2, ..., S_d\}$
2: (*DM, ID, maxIndex*) ← *PConstruct_ICSG*($\{ST_1, ST_2, ..., ST_d\}$, Σ)
3: (*precursor, tlevel, maxLevel*) ← *PForwardTopSort(DM, ID, maxIndex)*
4: *BackwardTopSort(DM, precursor, tlevel, maxLevel)*
5: Parallelly print the *MLCS*s of sequences set $T = \{S_1, S_2, ..., S_d\}$.

D. The Parallel Algorithm PTop_MLCS

In order to further improve the performance of the proposed algorithm *Top_MLCS* to meet the needs of long and large-scale sequences alignments in practice, based on the advantages of the multi-core computer, we propose a parallel algorithm *PTop_MLCS* by parallelizing *Top_MLCS*.

Our comprehensive experiments demonstrate that the algorithms *Construct_ICSG* and *ForwardTopSort* take the majority (averagely 95 percent) of running time in our proposed serial algorithm *Top_MLCS*, so the main problem we are facing now is how to parallelize these two algorithms. For the consideration of space, only the adopted strategies and inherent reasons of parallelized algorithms: *PConstruct_ICSG* and *PForwardTopSort* corresponding to the serial algorithms *Construct_ICSG* and *ForwardTopSort*, respectively, are described as follows.

1) ***Algorithm PConstruct_ICSG***. Given N_p+1 threads, the parallel algorithm *PConstruct_ICSG* uses one as the master $Proc_0$ and others as slaves $Proc_i$, $1 \le i \le N_p$. Since the main operations are to find immediate successors of dominants in the *ICSG* of the sequences set T and store necessary information in *DM* and *ID*, the design idea of the algorithm *PConstruct_ICSG* is to evenly distribute and parallelled-compute the workload of the operations with as low synchronization and mutex costs as possible (*i.e.*, the read lock is used in reading the *DM* and only the write lock is used in updating the *DM*). To this end, the parallel strategy is introduced: each slave thread calculates all the immediate successors of the dominants assigned by the master thread stored in $Queue_i$ independently, stores results in its temporary priority queue $tQueue_i$, and returns the results to the master thread to merge in *Queue* (a temporary priority queue used to store the merged results).

2) ***Algorithm PForwardTopSort.*** Similarly, the parallel strategies of the algorithm *PForwardTopSort* are as follows: the points in D^k are assigned evenly to slave threads. Each slave thread adopts the minimum size of row locks on the data structures of *tlevel*, *precursor* and *ID*, and updates its information about all the points in the given subset D_i^k, and stores points which satisfy the condition(*ID* = 0) in D_i^{k+1}. The master thread merges all the D_i^{k+1} into D^{k+1} so as to realize forward sorting and layering of the points in D^k in a parallel way level by level ($1 \le k \le |MLCS| - 1$).

The pseudo code of the proposed parallel algorithm *PTop_MLCS* is shown in Algorithms 5 $\sim$ 7.

E. The Time and Space Complexities of the Algorithm PTop_MLCS

The time complexity of the proposed parallel algorithm *PTop_MLCS* in every stage is given first, followed by the total time complexity.

Algorithm 6 *PConstruct_ICSG*($\{ST_1, ST_2, ..., ST_d\}, \Sigma$)

Queue ← {(0,0,...,0)}
indexOfPoint ← 0
DM ← {⟨*indexOfPoint*, (0,0,...,0)⟩}
indexOfPoint ← *indexOfPoint* +1
while *Queue* ≠ ∅ **do**
  Proc$_0$: evenly distribute elements of *Queue*
  Each slave *Proc*$_i$, $1 \le i \le N_p$, performs:
  Start
    get *Queue*$_i$ from *Proc*$_0$
    tQueue$_i$ ← ∅
    q = Pop(*Queue*$_i$)
    for each $v \leftarrow Successor(q)$ **do**
      if $v \notin$ *DM* **then**
        Push(*Queue*, v)
        DM ← *DM* ∪ {⟨*indexOfPoint*,v⟩}
        ID[v] ← 1
        indexOfPoint ← *indexOfPoint* + 1
      else
        ID[v] ← *ID*[v] + 1
      end if
    end for
    Send *tQueue*$_i$ to *Proc*$_0$
  End
  Proc$_0$: calculate *Queue* ← $\cup_{1 \le i \le N_p}$ *tQueue*$_i$
end while
DM ← *DM* ∪ {⟨*indexOfPoint*,$(\infty, \infty, ..., \infty)$⟩}
maxIndex ← *indexOfPoint* + 1
return (*DM, ID, maxIndex*)

Algorithm 7 *PForwardTopSort(DM, ID, maxIndex)*

D^0 ← {(0,0,...,0)}
k ← 0
for i ← 0 to *maxIndex* **do**
  tlevel[i] ← 0
  precursor[i] ← ∅
end for
while $D^k \neq \emptyset$ **do**
  Proc$_0$: distribute elements of D^k
  Each slave *Proc*$_i$, $1 \le i \le N_p$, performs:
  Start
    Get D_i^k from *Proc*$_0$
    $D^{k+1} \leftarrow \emptyset$
    for $q \in D^k$ **do**
      for each $v \leftarrow$ *Successor*(q) **do**
        tlevel[v] ← $k+1$
        if v is q's immediate successor **then**
          precursor[v] ← [*precursor*[v]; q]
          tlevel[v] ← $k+1$
        end if
        ID[v] ← *ID*[v] - 1
        if *ID*[v] == 0 **then**
          $D^{k+1} \leftarrow D^{k+1} \cup q$
        end if
      end for
      if q has no successor **then**
        precursor[*maxIndex*] = *precursor*[*maxIndex*] ∪ q
      end if
    end for
    Send D_i^{k+1} to *Proc*$_0$
  End
  Proc$_0$: calculate $D^{k+1} = \cup_{1 \le i \le N_p} D_i^{k+1}$
  $k \leftarrow k+1$
end while
maxLevel ← $k-1$
return (*precursor, tlevel, maxLevel*)

For each sequence S_l of T over alphabet Σ with a length n, the time $O(|\Sigma|(n))$ is needed at most for constructing its successor table IT_l by (3). Therefore, the time complexity of serially constructing d sequences of T is at most $O(d|\Sigma|(n))$.

The main operations of serially constructing the *ICSG* consist of establishing the predecessor-successor relationship among dominants and computing the in-degree of each point of the *ICSG*. Therefore, the time complexity of serially constructing the *ICSG* should be $O(|E|)$, where $|E|$ is the size of edges set E of the *ICSG*.

Given the points set V of the *ICSG*, since both a forward and backward topological sorting need to traverse every point in V of the *ICSG*, both the forward and the backward topological sorting take the time of $O(|V|)$.

Thus, the total time complexity of the proposed serial algorithm *Top_MLCS* equals $O(d|\Sigma|n) + O(|E|) + 2O(|V|)$. Since theoretical analysis and experimental results show that $O(d|\Sigma|n) \ll O(|E|)+2O(|V|)$, thus, $O(d|\Sigma|n+|E|+|V|) \approx O(|E| + |V|)$. Moreover, with $O(|E|)$ being the same order as that of $O(|V|)$ and $|E|$ at most several times of $|V|$, the time complexity of *PTop_MLCS* is $\frac{1}{N_p}O(|V|) + T_{com}$, where T_{com} is the communication overhead of the parallel execution algorithm *PTop_MLCS*. From the above discussion, we can see that the time complexity of both *PTop_MLCS* and *Top_MLCS* is linear in $|V|$.

Similarly, the storage space of successor tables is $O(d|\Sigma|n)$, and the storage space of the *ICSG* with $|V|$ points and $|E|$ edges is $O(d|V| + |E|)$, the space complexity of *PTop_MLCS* is $O(d|\Sigma|n + d|V| + |E|) \approx O(d|V| + |E|) = O(d|V|)$.

It is particularly worth mentioning here that the existing state-of-the-art dominant-point-based *MLCS* algorithms, *FAST_MLCS* [11] with effective pruning techniques and *Quick-DPPAR* [3] with a fast divide-and-conquer technique in performing the calculation of the dominants, result in the superior performance of the algorithm time. However, both the algorithms *FAST_MLCS* and *Quick-DPPAR* do not eliminate the inherent defects of the general dominant-point-based *MLCS* algorithms (see Section III). The linear time complexity $O(|MLCS|)$ without considering the time of computing dominant points is not reasonable for *FAST_MLCS*, while the time complexity of *Quick-DPPAR*, $1/N_p(1+\alpha(n))O(n|\Sigma|d+|D||\Sigma|d(log^{d-2}n + log^{d-2}|\Sigma|))$, where $\lim_{n\to\infty} \alpha(n) = 0$, is obviously non-linear, as $|D|$ is the number of the vertices of *MLCS-DAG* which is much larger than $|V|$ of *ICSG*. Thus the complexities of time and space of the proposed algorithm are much lower than those of *FAST_MLCS* and *Quick-DPPAR*.

VI. Experimental results

In the experiments, all of the algorithms were run on a Dell Precision T5600 Workstation Processors Intel (R) CPU (2.93GHz), 64GB of RAM platform, by adopting Java in JDK 1.7 and tested on the benchmark datasets provided by real biological sequences datasets ncbi [14] and dip [15] including a set of synthetic sequences randomly drawn from alphabets of the DNA and amino acid, wherein ten groups of real biological sequences and synthetic sequences are randomly selected from its datasets for experiments, respectively, and each group consists of 5 or d sequences. We tested the algorithms 10 times on above 20 groups of benchmark datasets, and the following algorithms' running times are their average running times in milliseconds (ms).

TABLE I. THE RUNNING TIME OF *FAST_LCS* (A1), *Quick-DPPAR* (A2) AND *PTop_MLCS* (A3) ALGORITHMS OF 5 SEQUENCES WITH VARIOUS LENGTHS ON EIGHT THREADS

$\|\Sigma\| = 4$				$\|\Sigma\| = 20$			
$\|S_i\|$	**A1(ms)**	**A2(ms)**	**A3(ms)**	$\|S_i\|$	**A1(ms)**	**A2(ms)**	**A3(ms)**
25	31	28	6	40	24	27	6
50	133	131	62	80	156	83	50
75	1640	1323	187	120	1131	454	118
100	11107	8018	612	160	6537	1319	337
120	43803	28567	1492	200	29835	18386	1068
140	121450	82936	3128	240	106018	63600	2534
160	291760	204173	5904	280	455342	306465	4169
180	1079686	641105	11027	320	1384197	1065327	11089
200	–	–	19382	360	3039639	2547178	18727
220	–	–	31575	400	5985971	4379261	30708
240	–	–	47993	440	12289282	8059969	50876
260	–	–	79748	480	19845857	12220583	76560
280	–	–	121682	520	–	–	117382
300	–	–	194065	560	–	–	163564
320	–	–	286617	600	–	–	231337
340	–	–	418350	700	–	–	595948
360	–	–	655370	800	–	–	1632850

TABLE II. THE RUNNING TIME OF *FAST_LCS* (A1), *Quick-DPPAR* (A2) AND *PTop_MLCS* (A3) ALGORITHMS OF D SEQUENCES WITH 100 AND 250 LENGTHS, RESPECTIVELY, ON EIGHT THREADS

$\|S_i\| = 100$ and $\|\Sigma\| = 4$				$\|S_i\| = 250$ and $\|\Sigma\| = 20$			
d	**A1(ms)**	**A2(ms)**	**A3(ms)**	d	**A1(ms)**	**A2(ms)**	**A3(ms)**
3	141	144	75	3	984	2130	156
4	1818	2631	206	4	18823	23119	949
5	32394	30311	924	5	879434	624429	5093
6	985023	653235	3620	6	9573349	7451484	19688
7	+	+	10927	7	+	+	69440
8	+	+	31532	8	+	+	187688
9	+	+	54676	9	+	+	420122
10	+	+	103598	10	+	+	861038
60	+	+	199638	60	+	+	18122
100	+	+	62526	100	+	+	5536
150	+	+	23170	150	+	+	2946
200	+	+	17275	200	+	+	1611
300	+	+	15064	300	+	+	657
400	+	+	13560	400	+	+	512
500	+	+	11900	500	+	+	456
600	+	+	11532	600	+	+	424
700	+	+	9954	700	+	+	337
800	+	+	9361	800	+	+	300
900	+	+	9510	900	+	+	306
1000	+	+	9292	1000	+	+	268

We comprehensively validated our serial algorithm *Top_MLCS* and parallel algorithm *PTop_MLCS* and compared our *PTop_MLCS* algorithm with the existing state-of-the-art dominant-point-based parallel *FAST_LCS* [11] and *Quick-DPPAR* [3] algorithms (these algorithms were reported to have been realized in corresponding literature and run on the same hardware platform). For the limitation of the paper space, only parts of the experimental results are shown in Table I and Table II, where the notation '–' represents the intolerable running time ($>$ 24 hours or $\gg$ 24 hours), and '+' stands for the memory overflow resulting in the algorithm failure, respectively.

Table I shows that the time performance of the proposed parallel *PTop_MLCS* algorithm is superior to that of the two other algorithms, reaching up to 2-3 orders of magnitude faster on the long sequences. For example, the running time of our algorithm *PTop_MLCS* shown in Table I has an average time 31 times / 100 times shorter than that of the algorithm *FAST_LCS* (ranging from 5-98 times / 4-259 times), and 21 times / 69 times shorter than that of the algorithm *Quick-DPPAR* (ranging from 5-58 times / 5-160 times) in different

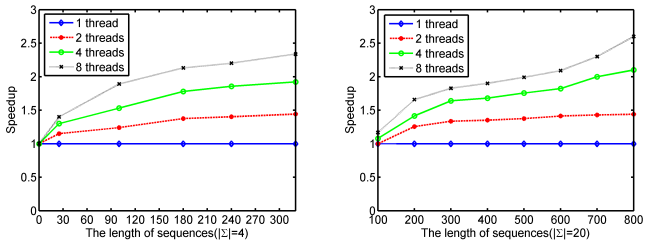


Fig. 9. The speedup of *PTop_MLCS* over *Top_MLCS* on *MLCS* problem of 5 sequences.

$|\Sigma|$ individually. Since the running time of the *MLCS* algorithm is proportional to the aligned sequence' length, in terms of the growth trends of the three algorithms' (*FAST_LCS*, *Quick-DPPAR*, and *PTop_MLCS*) running time with the length of the aligned sequences, the average growth rate of the algorithms' running times with every increase in length of 20 for aligned sequences are 5 times / 4 times (ranging from 2-12 times / 2-7 times), 5 times / 4 times (ranging from 2-10 times / 2-14 times), and 3 times / 3 times (ranging from 2-10 times / 2-8 times) in different $|\Sigma|$ individually. Hence, the theoretical estimation of the average running time of the algorithm *PTop_MLCS* approximates 406 times (ranging from 5-3911 times) shorter than that taken by the algorithm *FAST_LCS* and 296 times (ranging from 4-2740 times) shorter than that taken by the algorithm *Quick-DPPAR*. Moreover, with the increasing length of aligned sequences, the advantage of the algorithm *PTop_MLCS* in time performance is even more obvious compared with algorithms *FAST_LCS* and *Quick-DPPAR*. But above all, it is clear that our algorithm *PTop_MLCS* is more suitable for the long sequences.

As discussed above, due to the efficient elimination of the storage, the calculation of redundant dominants, and the tremendous dimension-by-dimension comparisons of dominants of the algorithm *PTop_MLCS*, Table II reveals that the advantage of our algorithm *PTop_MLCS* gets more obvious with an increasing number of sequences alignments, e.g., when testing on 6 sequences ($|\Sigma| = 4$) with a length of 100, the time performance of the *PTop_MLCS* has already become up to 272 times better than that of the *FAST_MLCS* and 181 times better than that of the algorithm *Quick-DPPAR*. It is a remarkable fact that the algorithms *FAST_MLCS* and *Quick-DPPAR* cannot work due to memory overflow when the number of aligned sequences is larger than 6. However, our proposed algorithm *PTop_MLCS* can also run correctly and efficiently even though the number of aligned sequences reaches 1000. Moreover, with the increasing number of aligned sequences, the number of dominants firstly increases and then usually decreases. So the experimental results of our algorithm *PTop_MLCS* shown in Table II is reasonable, and the algorithm is very suitable for the large-scale sequences alignments.

Besides the above time performance comparison experiments of algorithms, we further evaluated the speedup of the parallel algorithm *PTop_MLCS* over the serial algorithm *Top_MLCS* with a variety of lengths and sizes of aligned sequences. The experimental results shown in Fig. 9 and Fig. 10 fully demonstrate the effective speedup of our algorithm *PTop_MLCS* over the serial algorithm *Top_MLCS*.

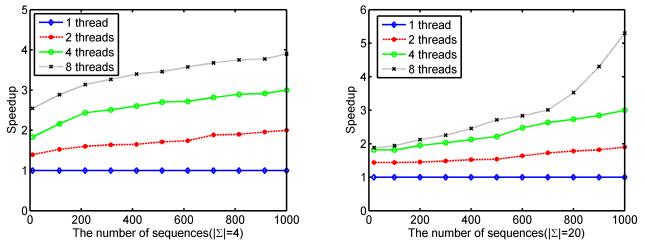


Fig. 10. The speedup of *PTop_MLCS* over *Top_MLCS* on *MLCS* problem of 3 $\sim$ 1000 sequences.

VII. Conclusion

In order to overcome the disadvantages of the leading dominant-point-based *MLCS* algorithms that are not applicable to the *MLCS* problems with long and large-scale sequences alignments, we developed an efficient parallel algorithm for this NP-hard problem, and the performance bottlenecks of the leading dominant-point-based *MLCS* algorithms are first revealed in this paper. Then a new generalized problem-solving model *ICSG-PCC* is proposed by introducing the following key concepts: irredundant common subsequence graph (*ICSG*), parallel collection, and parallel collection chain (*PCC*). Furthermore, based on the model, a fast and memory efficient parallel *MLCS* algorithm *PTop_MLCS* is proposed. In addition to the external communication overhead T_{com} of parallel execution, theoretical analyses show that the time and space complexities of the proposed algorithm are only relevant and linear to the number of the dominants from aligned sequences, *i.e.*, $\frac{1}{N_p}O(|V|) + T_{com}$ and $O(d|V|)$, respectively, where $|V|$ is the number of dominants of the constructed graph *ICSG* of d aligned sequences with a length n. Finally, the proposed algorithm is evaluated by comprehensive experiments on the benchmark datasets of both random synthetic and real biological sequences. The results show that the proposed problem-solving model *ICSG-PCC* for the *MLCS* problem is efficient and effective, and that the proposed algorithm *PTop_MLCS* greatly outperforms the leading state-of-the-art dominant-point-based parallel algorithms and are very suitable for long and large-scale sequences alignments, resulting in a positive push in efficiently tackling the NP-hard problem.

In the future work, we will further improve the efficiency of the proposed algorithm *PTop-MLCS* and make it suitable for big data analysis.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No.61472296 and 61472297).

References

[1] D. Maier, "The complexity of some problems on subsequences and supersequences," *Journal of the ACM (JACM)*, vol. 25, no. 2, pp. 322–336, Apr. 1978.

[2] M. Zvelebil and J. Baum, *Understanding bioinformatics*. Garland Science, 2007.

[3] Q. Wang, D. Korkin, and Y. Shang, "A fast multiple longest common subsequence (mlcs) algorithm," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 23, no. 3, pp. 321–334, 2011.

[4] D. Sankoff, "Matching sequences under deletion/insertion constraints," *Proceedings of the National Academy of Sciences*, vol. 69, no. 1, pp. 4–6, 1972.

[5] W. J. Masek and M. S. Paterson, "A faster algorithm computing string edit distances," *Journal of Computer and System sciences*, vol. 20, no. 1, pp. 18–31, 1980.

[6] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, vol. 18, no. 6, pp. 341–343, 1975.

[7] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Commun. ACM*, vol. 20, no. 5, pp. 350–353, May 1977.

[8] W. Hsu and M. Du, "Computing a longest common subsequence for a set of strings," *BIT Numerical Mathematics*, vol. 24, no. 1, pp. 45–59, 1984.

[9] M. Lu and H. Lin, "Parallel algorithms for the longest common subsequence problem," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 5, no. 8, pp. 835–848, 1994.

[10] T. K. Yap, O. Frieder, and R. L. Martino, "Parallel computation in biological sequence analysis," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 9, no. 3, pp. 283–294, 1998.

[11] Y. Chen, A. Wan, and W. Liu, "A fast parallel algorithm for finding the longest common sequence of multiple biosequences," *BMC bioinformatics*, vol. 7, no. Suppl 4, p. S4, 2006.

[12] J. Yang, Y. Xu, and Y. Shang, "An efficient parallel algorithm for longest common subsequence problem on gpus," in *Proceedings of the World Congress on Engineering*, vol. 1, 2010.

[13] Y. Li, Y. Wang, and L. Bao, "Facc: a novel finite automaton based on cloud computing for the multiple longest common subsequences search," *Mathematical Problems in Engineering*, vol. 2012, 2012.

[14] Pseudomonas aeruginosa pao1 chromosome, complete genome. [Online]. Available: http://www.ncbi.nlm.nih.gov/nuccore/110645304?report=fasta

[15] Database of interacting proteins. [Online]. Available: http://dip.doe-mbi.ucla.edu/dip/Download.cgi

[16] A. Apostolico, S. Browne, and C. Guerra, "Fast linear-space computations of longest common subsequences," *Theoretical Computer Science*, vol. 92, no. 1, pp. 3–17, 1992.

[17] D. Korkin, "A new dominant point-based parallel algorithm for multiple longest common subsequence problem," Technical Report TR01-148, Univ. of New Brunswick, Tech. Rep., 2001.

[18] J. Yang, Y. Xu, G. Sun, and Y. Shang, "A new progressive algorithm for a multiple longest common subsequences problem and its efficient parallelization," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 5, pp. 862–870, 2013.

[19] E. Horowitz and S. Sahni, *Fundamentals of data structures*. Pitman, 1983.

[20] D. E. Knuth, *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.

[21] K. Rosen, *Discrete Mathematics and Its Applications 7th edition*. McGraw-Hill Science, 2011.