A Real Linear and Parallel Multiple Longest Common Subsequences (MLCS) Algorithm

Yanni Li
School of Software
Xidian University, China
yannili@mail.xidian.edu.cn

Hui Li
School of Cyber Engineering
Xidian University, China
hli@xidian.edu.cn

Tihua Duan
Information Center
Shanghai Finance University, China
duantihua@163.com

Sheng Wang
Faculty of Engineering, Environment and Computing
Coventry University, UK
sheng.wang@conventry.ac.uk

Zhi Wang
School of Computer Science and Technology
Xidian University
xd_zhiwang@163.com

Yang Cheng
School of Software
Xidian University, China
xd_chengyang@163.com

ABSTRACT

Information in various applications is often expressed as character sequences over a finite alphabet (*e.g.*, DNA or protein sequences). In Big Data era, the lengths and sizes of these sequences are growing explosively, leading to grand challenges for the classical NP-hard problem, namely searching for the Multiple Longest Common Subsequences (*MLCS*) from multiple sequences. In this paper, we first unveil the fact that the state-of-the-art *MLCS* algorithms are unable to be applied to long and large-scale sequences alignments. To overcome their defects and tackle the longer and large-scale or even big sequences alignments, based on the proposed novel problem-solving model and various strategies, *e.g.*, parallel topological sorting, optimal calculating, reuse of intermediate results, subsection calculation and serialization, etc., we present a novel parallel *MLCS* algorithm. Exhaustive experiments on the datasets of both synthetic and real-world biological sequences demonstrate that both the time and space of the proposed algorithm are only linear in the number of dominants from aligned sequences, and the proposed algorithm significantly outperforms the state-of-the-art *MLCS* algorithms, being applicable to longer and large-scale sequences alignments.

Keywords

Multiple Longest Common Subsequences (MLCS); Non-redundant Common Subsequence Graph (NCSG); Topological Sorting; Subsection Calculation and Serialization

1. INTRODUCTION

Information in various applications often can be abstracted as character sequences over a finite alphabet Σ, *e.g.*, DNA or protein sequences in biology. Searching for the Multiple Longest Common Subsequences from a group of sequences (*MLCS*) over a finite alphabet Σ is a classical NP-hard problem [13] and has found many important applications in many areas, *e.g.*, bioinformatics, computational genomics, pattern recognition, data mining, etc. For example, in bioinformatics, 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 is an important way to identify the sequence similarity, which can be utilized in gene discovery, the construction of an evolutionary tree, the evidence of the species' common origin [20], etc. With the successful implementation of the Human Genome Project, the lengths, sizes of biological [4, 8] or other types of sequences are growing explosively and exponentially [16]. Mining the *MLCS*s from these sequences is becoming a more and more important research topic and facing severe challenges.

In the past forty years, in order to efficiently tackle the *LCS/MLCS* problem, various types of *LCS/MLCS* algorithms [2, 3, 6, 7, 11, 12, 14, 15, 16, 17, 19] and tools (*e.g.*, SAMtools[1], BLAST[2], Clustal Omega[3], etc.) have been proposed, which can be divided into two categories: classical dynamic programming and dominant-point-based algorithms. It has been demonstrated that the dominant-point-based *LCS/MLCS* algorithms have an overwhelming advantage over classical dynamic programming ones due to their great reduction in the size of search space by orders of magnitude [16]. In particular, *FAST_LCS* [2] and *Quick-DPPAR* [16] are the most efficient parallel *MLCS* algorithms among dominant-point-based algorithms. However, it will be shown in this paper with both theoretical and empirical study that these algorithms suffer severely from many unnecessary and redundant storage, computation, comparison and deletion of multi-dimensional dominants. Therefore, this kind of algorithms is essentially inapplicable to long and large-scale sequences alignments. Moreover, both of the two types of algorithms are in fact not linear. To overcome this problem, we present a new problem-solving graphical model and propose *a real linear and parallel algorithm for multiple*

KDD '16, August 13-17, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2939672.2939842

[1] http://www.htslib.org/

[2] http://blast.ncbi.nlm.nih.gov/Blast.cgi

[3] http://www.ebi.ac.uk/Tools/msa/clustalo/

longest common subsequences mining, equipped with several carefully designed strategies. Our main contributions are as follows:

- We propose a novel problem-solving model, namely *Non-redundant Common Subsequence Graph, NCSG*, and introduce both a forward and a backward parallel topological sorting strategies on the *NCSG*, individually leading to efficiently eliminating existing leading dominant-point-based *MLCS* algorithm' defects and finding all *MLCS*s of the aligned sequences at once with a very low cost of time and space.
- With the introduction of series of well-designed strategies, *e.g.*, optimal computing, subsection calculation and serialization, reuse of intermediate results and multiple concurrent execution, we present an efficient parallel *MLCS* algorithm, which is essentially appropriate to longer and large-scale sequences alignments. Theoretical study of the time and space complexities for the proposed parallel *MLCS* algorithm indicates that the proposed algorithm is only linear in the number of dominants from the aligned sequences.
- We validate the proposed parallel *MLCS* algorithm on DNA and amino acid sample sequences from real biological datasets 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. Theoretical study and experimental results justify that our algorithm not only greatly outperforms state-of-the-art competitors, but also is practical for longer and large-scale or even big sequences alignments.

The rest of this paper is organized as follows. Section 2 first gives a formal definition of the *MLCS* problem, and then briefly reviews the related work on classic dynamic programming and dominant-point-based algorithms in introducing some necessary preliminaries, and finally reveals some inherent limitations of the leading dominant-point-based algorithms. To overcome the limitations, Section 3 discusses the proposed novel problem-solving model *NCSG* and two topological sorting strategies on the *NCSG*. Section 4 first elaborates upon several well-designed strategies tackling longer and large-scale sequences alignments, and then gives our novel efficient parallel *MLCS* algorithm, followed by a detailed study of the time and space complexities. Comprehensive experiments are conducted in Section 5. Finally, Section 6 concludes this work.

2. PRELIMINARIES AND RELATED WORK

In this part, we first present series of preliminaries used in our model and then discuss some related works.

2.1 Definitions of LCS/MLCS Problems

A subsequence of a given sequence over a finite alphabet Σ 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.*, $x_i, y_j \in \Sigma$, and the *Longest Common Subsequence* (*LCS*) problem is to find out all the longest common subsequences of X and Y. Similarly, the *Multiple Longest Common Subsequence* (*MLCS*) problem is to find out all the longest common subsequences of d ($d \geq 3$) sequences with equal length n or unequal length. Obviously, the *LCS* is a special case of *MLCS*.

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

2.2 Preliminaries and Related Algorithms

Based on the methods adopted, existing *LCS/MLCS* algorithms can be divided into two categories: classical dynamic programming and dominant-point-based algorithms. Depending on whether they are parallelized, the algorithms can also be divided into two types: serial and parallel ones.

1) *Classical Dynamic Programming Algorithms*: These algorithms are based on dynamic programming [14, 15]. 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 Σ, 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, calculated as follows:

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

Once the score matrix L is calculated, all the *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 $O(mn)$. In general, given d sequences S_1, S_2, ..., S_d with arbitrary equal or unequal lengths, $n_1, \ldots, n_d$, the matrix L can be naturally extended to d dimensions for the *MLCS* problem, in which the element $L[i_1, i_2, ..., i_d]$ can be calculated by Eq. 2 in a similar manner to Eq. 1, the time and space complexities are therefore both $O(\prod_{i=1}^{d} n_i)$.

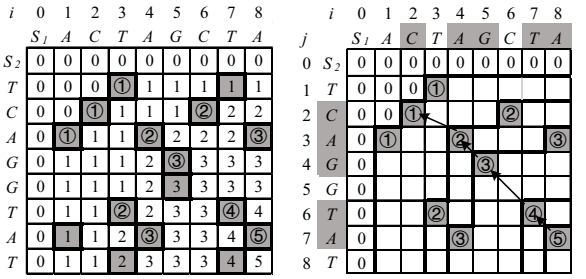
$$L[i_1, \ldots, i_d] = \begin{cases} 0 & \text{if } \exists i_j = 0, 1 \leq j \leq d \\ L[i_1 - 1, \ldots, i_d - 1] + 1 & \text{if } S_1[i_1] = \ldots = S_d[i_d] \\ \max(\bar{L}) & \text{otherwise.} \end{cases} \quad (2)$$

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

Fig. 1 illustrates 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 L.

To further reduce time and space complexities, various improved dynamic programming *LCS/MLCS* algorithms [1, 3, 6] have been proposed. For example, Hirschberg [3] presented a new *LCS* algorithm based on the divide-and-conquer approach, which reduces the space complexity to $O(m+n)$; however, its time complexity remains to be $O(mn)$. Masek and Paterson [14] put forward an improved dynamic programming *LCS* algorithm for two sequences with length n using a fast computing method of edit distance, whose worst time complexity is $O(n^2/\log n)$. Unfortunately, most of the aforementioned algorithms only address the *LCS* problem but not *MLCS* and have high time and space complexities.

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



(a) The score matrix L of sequences S_1 and S_2. (b) An $LCS = CAGTA$ extracted from L.

Figure 1: The score matrix L of $S_1 = ACTAGCTA$ and $S_2 = TCAGGTAT$ over a finite alphabet $\Sigma = \{A, C, G, T\}$ and LCS ($CAGTA$) extracted from L. The regions of the same entry values are bounded by thick contours; the corner points of the contours are dominant points (circled) and the greyed points are matched points.

Definition 1: For a sequences set $T = \{S_1, S_2, ..., S_i, ..., S_d\}$ over a finite alphabet Σ, and $|S_i| = n$.[4] Let $S_i[p_i](S_i[p_i] \in \Sigma)$ be the p_i-th ($p_i \in \{1, 2, ..., n\}$) character in S_i. The point $p = (p_1, p_2, ..., p_d)$ is called a *matched point* of T, if and only if $S_1[p_1] = S_2[p_2] = ... = S_i[p_i] = ... = S_d[p_d] = \sigma(\sigma \in \Sigma)$.

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

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

Fig. 1 shows that a dominant must be a matched point, but not vice versa, and the size of the dominant set D is smaller than that of the matched points, which have been proved by [2, 7, 16].

Comparing with 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. The algorithms consist of two procedures as follows.

(1) ***Constructing MLCS-DAG.*** First of all, introduce two dummy points, source point $(0, 0, ..., 0)$ and sink point $(\infty, \infty, ..., \infty)$, where the sink point is defined as the immediate successor of those points without an immediate successor. Afterwards, let $k = 0$, and $D^0 = \{(0, 0, ..., 0)\}$. Next, with a forward iteration procedure $(0 \rightarrow k)$, the $(k+1)$-th dominants D^{k+1} are computed based on the k-th dominants D^k, and this procedure is denoted as $D^k \rightarrow D^{k+1}$, where $0 \leq k \leq |MLCS| - 1$ ($|MLCS|$ denotes the length of *MLCS*s of T). As a result, a directed acyclic graph consisting of all the *MLCS*s of T (*MLCS-DAG*, for short) is constructed level by level in the following two steps:

[4]In fact, the algorithms apply to general case where the length of S_i may not be the same. The only reason we fix $|S_i| = n$ here is to facilitate the following discussions.

Step 1: Based on D^k, all the immediate successors of each dominant from D^k are calculated, denoted as D^{k+1}_{init}.

Step 2: There exist massive repeated dominants and dominated points in D^{k+1}_{init}, both collectively called *redundant points* for short. Since the redundant points do not contribute to the *MLCS*s, the operation $Minimal(D^{k+1}_{init})$ (*Minimal*() for short) of eliminating them should be conducted in order to save time and space. In general, the operation *Minimal*() is performed by comparing between the dominants and redundant points in D^{k+1}_{init} one dimension after another, resulting in the baseline D^{k+1} over D^{k+1}_{init}.

(2) ***Computing MLCSs.*** All the *MLCS*s of T are then found by tracing back through the constructed *MLCS-DAG* from the last dominant set to D^0 iteratively. An instance of this kind of algorithms is shown in Example 1 and Fig. 2.

As 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 [16].

Hunt et al. [7] first proposed a dominant-point-based *LCS* algorithm with time complexity $O((r + n) \log n)$, where r is the number of all dominants of two sequences with length n. Afterwards, a variety of dominant-point-based *LCS/MLCS* algorithms have been presented [1, 6]. To further improve the efficiency, some parallel dominant-point-based *LCS* [12, 19] and *MLCS* algorithms [2, 10, 16] were proposed. Korkin [10] first proposed a parallel *MLCS* algorithm with time complexity $O(|\Sigma||D|)$. Chen et al. [2] presented an efficient parallel *MLCS* algorithm over the alphabet $\Sigma = \{A, C, G, T\}$, *FAST_LCS*, with series of pruning rules. Wang et al. [16] developed an efficient parallel *MLCS* algorithm *Quick-DPPAR*, claiming that the proposed algorithm has reached a near-linear speedup with respect to its serial version *Quick-DP*. It is worth mentioning that [11, 17] made attempts to develop efficient parallel algorithms on GPUs for the *LCS* problem and on cloud platform for the *MLCS* problem, respectively. Regretfully, [17] is not suitable for the general *MLCS* problem, as a large amount of synchronous cost [11] remains to be solved. For 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 better solution for the *MLCS* problem. However, as we will show in next subsection, such algorithms are facing serious challenges with the explosive expansion in the lengths and sizes of sequences from varieties of applications.

2.3 Limitations of the Leading MLCS Algorithms

We first give an example to illustrate the limitations of the leading dominant-point-based *MLCS* algorithms, and then make a further study.

Example 1: Given sequences $S_1 = TGACGATC$, $S_2 = ATGCTCAG$ and $S_3 = CTAGTACG$ over the alphabet $\Sigma = \{A, C, G, T\}$, construct their *MLCS-DAG*s and find out all the *MLCS*s of S_1, S_2 and S_3 by the general dominant-point-based algorithms.

Based on the dominant-point-based *MLCS* algorithm, the constructed *MLCS-DAG* of S_1, S_2 and S_3 is shown in Fig. 2. The construction process is as follows. First of all, let $k = 0$, $D^0 = \{(0, 0, 0)\}$ (0-level dominant). By Definition 2, all the

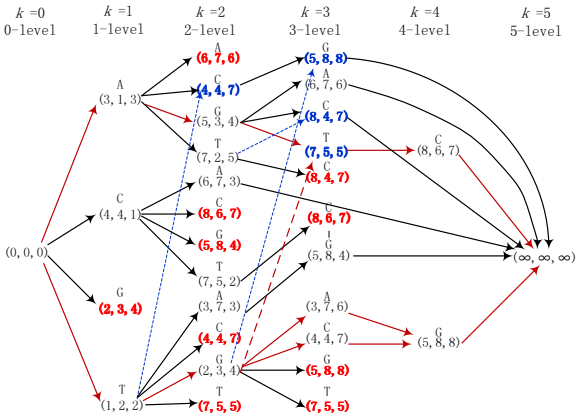


Figure 2: The constructed *MLCS-DAG* of S_1, S_2 and S_3 over $\Sigma = \{A, C, G, T\}$ by the dominant-point-based algorithms, the blue (red) points denote repeated (dominated) points, which would be deleted after the operation *Minimal*(), and a dashed arrow indicates an added edge after deleting a point. All the *MLCS*s (marked by red arrows) can be obtained by tracing back from the sink point (∞, ∞, ∞) to the source point $(0, 0, 0)$ on the *MLCS-DAG*.

immediate successors D^1_{init} = {(3, 1, 3), (4, 4, 1), (2, 3, 4), (1, 2, 2)} corresponding to the characters {*A*, *C*, *G*, *T*} from D^0 are calculated. The operation *Minimal*() is then performed so as to eliminate the redundant point $(2, 3, 4)$ (a dominated point of the point $(1, 2, 2)$), leading to the end of procedure $D^0 \rightarrow D^1$ (1-level dominants D^1 = {(3, 1, 3), (4, 4, 1), (1, 2, 2)}). Repeating the same procedure as above $D^k \rightarrow D^{k+1}$, D^2, D^3, D^4, and D^5 of 2-level to 5-level dominants can be obtained in turn, where D^2 = {(4, 4, 7), (5, 3, 4), (7, 2, 5), (6, 7, 3), (7, 5, 2), (3, 7, 3), (2, 3, 4)}, D^3 = {(5, 8, 8), (6, 7, 6), (8, 4, 7), (7, 5, 5), (5, 8, 4), (3, 7, 6), (4, 4, 7)} and D^4 = {(8, 6, 7), (5, 8, 8)}, respectively. Since $D^5_{init} = \emptyset$, let $D^5 = \{(\infty, \infty, \infty)\}$ and end the construction of *MLCS-DAG*. Finally, based on the *MLCS-DAG*, all the *MLCS*s of S_1, S_2 and S_3 corresponding to the character sequences of points on the longest paths (marked by red arrows) over the *MLCS-DAG*($MLCS_1 = AGTC, MLCS_2 = TGTC, MLCS_3 = TGAG$ and $MLCS_4 = TGCG$ with length 4) can be obtained by tracing back from the sink point (∞, ∞, ∞) to the source point $(0, 0, 0)$.

Fig. 2 clearly shows that there are numerous redundant points in the *MLCS-DAG*, and that many points, *e.g.*, $(2, 3, 4)$, $(4, 4, 7)$ and $(5, 8, 8)$, have been recalculated and compared with other points many times. Moreover, there exist many points in the *MLCS-DAG*, which are useless to the *MLCS*s of sequences S_1, S_2 and S_3.

To further show the limitations of the dominant-point-based *MLCS* algorithms, we conducted a statistical study with various types of the redundant points of the *MLCS-DAG* from sequences with various lengths and sizes of alphabet over the real-world and synthetic datasets utilized in Section 5, 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 data 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}). Let $N = |D^{k+1}_{init}|$, 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 computational time in the operation *Minimal*(). Moreover, tremendous comparisons in d dimensions among N points are inevitable besides the massive amount of redundant points in the operation *Minimal*() of the process $D^k \rightarrow D^{k+1}$. For d sequences with length n ($d \geq 3$), it is proved that the time for the comparisons among N d-dimensional points dimension by dimension in a brute-force manner is $O(dN^2)$ [5, 9]. Even if the divide-and-conquer strategy is adopted, $O(dN \log^{d-2} n)$ comparisons are still needed [16].

2) The constructed *MLCS-DAG* contains a large number of useless points not contributing to any *MLCS*s, called *non-critical points*.

The statistical data shows that the ratio of $|K|$ (the total number of the *key points* that contribute to the *MLCS*s in the *MLCS-DAG*) to $|D|$ (the total number of the points in *MLCS-DAG*) ranges from only 1 : 10 to 1 : 100,000. Moreover, the larger the d, n and $|\Sigma|$, the smaller the ratio of $|K|/|D|$ is. The massive non-critical points in the *MLCS-DAG* introduce another serious space problem of storing the *MLCS-DAG* in RAM and time problem when tracing back to find its *MLCS*s on the *MLCS-DAG*.

Above all, since *Minimal*() is the 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* algorithms are nonlinearly related to d and $|D|$. In addition, the length of *MLCS*, $|MLCS|$ is proportional to the aligned sequences 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 5) show that ***the leading dominant-point-based MLCS algorithms are not applicable to long and large-scale sequences alignments***.

3. A NOVEL PROBLEM-SOLVING MODEL AND MINING MLCS STRATEGY

3.1 A Novel Problem-Solving Model: NCSG

As mentioned above, all the *MLCS*s of aligned sequences set T are constructed by their relevant dominants. For the dominants set D from T, since the immediate predecessor-successor relationships between the dominants in D constitute a partial order set $\preceq$, we can represent the relationships by a directed acyclic graph $G = (D, \preceq)$. Similar to the construction of *MLCS-DAG*, 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 dominants in D being the successors of $(0, 0, ..., 0)$ and the predecessors of $(\infty, \infty, ..., \infty)$. The construction of G is as follows. 1) From the dummy source point $(0, 0, ..., 0)$, calculate all of its immediate successors and connect them with directed edges from the source point to the immediate successors. 2) From these calculated immediate successors, compute all their immediate successors. If the immediate successor found is already in G, only add a directed edge to it; otherwise, add it to G and connect it by a directed edge. If a successor has no immediate successors, connect it with the sink point by a directed edge. 3) Repeat above procedure 2) until all points in G have computed their immediate successors. Notably, such G is in fact an *MLCS-DAG* but with

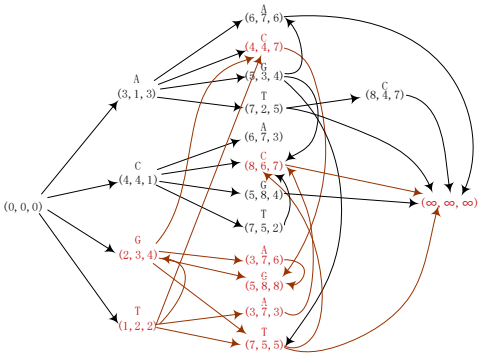


Figure 3: The constructed *NCSG* of S_1, S_2 and S_3 over the alphabet $\Sigma = \{A, C, G, T\}$, where *the optimal subgraph Sub-NCSG*$_4$ of the *NCSG* (marked red) is constructed from *the optimal dominant* (1, 2, 2).

the following properties: no repeated dominants and comparison of dominants in the dimension by dimension manner on constructing G. Due to that, finding an *MLCS* can now be regarded as identifying the longest path over G from the source point to the sink point, and vice versa. Since G has no repeated dominants, and any path of G corresponds to a common subsequence of T, we refer to it as *Non-redundant Common Subsequence Graph*, abbreviated as *NCSG* in the following. The constructed *NCSG* of the above sequences S_1, S_2 and S_3 is shown in Fig. 3.

3.2 A Strategy to Mine MLCSs: Forward/ Backward Topological Sorting on NCSG

Given a constructed *NCSG*, we need to design an efficient and effective strategy to extract all *MLCS*s from it. To this end, let's start by reviewing the following concepts from the graph theory.

Definition 4: For a directed acyclic graph $G = \langle V, \preceq \rangle$, *topological sorting* is to find an overall order of the vertices V in G from the partial order $\preceq$ [5, 9].

Definition 5: The topological sorting algorithm [9] is to complete the topological sorting over the vertices V in G from the partial order $\preceq$. To this end, it iteratively performs the following two steps until all the vertices V in G have been traversed and processed: 1) output the vertices with in-degree 0; 2) delete the edges connecting to the vertices.

We found that the topological sorting over the vertices V in G from the partial order $\preceq$ is only associated with the cardinality of the in-degree of vertices, but not related to the dimensions of vertices. We do not need to perform the comparison of the dominants dimension by dimension to sort and layer them as in the construction of *MLCS-DAG* for leading dominant-point-based *MLCS* algorithms. Inspired by this observation, we present a novel efficient method to sort and layer all the dominants of the *NCSG*, *i.e.*, sorting and layering all of the dominants of the *NCSG* from the 0-level dominant set D^0 to the k-level dominant set D^k level by level with the topological sorting algorithm (called *a forward topological sorting algorithm*, denoted as Algorithm *ForwardTopSort*), $1 \leq k \leq |MLCS| - 1$. An example of sorting and layering all the dominants on the *NCSG* of the sequences S_1, S_2 and S_3 is shown in Fig. 4.

Moreover, based on the sorted and layered *NCSG* shown in Fig. 4, how do we find all *MLCS*s at once with a very low cost of time and space without multiple backtracking

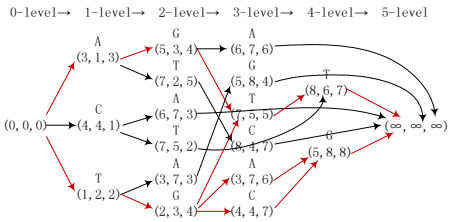


Figure 4: Sorting and layering all the dominants on *NCSG* of S_1, S_2 and S_3 by *ForwardTopSort*, in which all the longest paths are marked by red arrows.

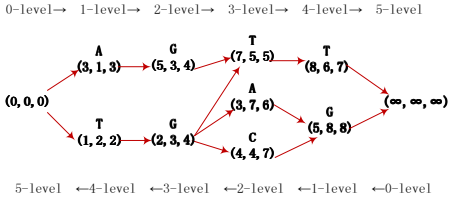


Figure 5: With *BackwardTopSort* to the sorted and layered *NCSG* of S_1, S_2 and S_3 shown in Fig. 4, the optimized *NCSG* can be obtained, in which any path corresponds to an *MLCS* of the sequences.

processes as in the leading dominant-point-based *MLCS* algorithm? By investigating the sorted and layered *NCSG*, we find that *the sum of the numbers of the forward levels (from* $\{(0, 0, ..., 0)\}$ *to* $\{(\infty, \infty, ..., \infty)\}$*) and the backward levels (from* $\{(\infty, \infty, ..., \infty)\}$ *to* $\{(0, 0, ..., 0)\}$*) of those dominants* (*called key points*) *residing in the longest paths corresponding to the MLCSs is exactly equal to* $|MLCS| + 1$*; however non-critical points would not meet the property* (see Fig 5)[5]. Based on the observation, we replace the in-degree with the out-degree and layer the *NCSG* by the topological sorting algorithm from the point $\{(\infty, \infty, ..., \infty)\}$ to $\{(0, 0, ..., 0)\}$ (called *a backward topological sorting algorithm*, denoted as Algorithm *BackwardTopSort*). Thanks to that, all the non-critical points in the *NCSG* are now identified and can be easily removed. Fig. 5 (called *the optimized NCSG*) demonstrates the result with *BackwardTopSort* to Fig. 4. In particular, the *NCSG* shown in Fig. 5 contains only those key points, that is, each path in the optimized *NCSG* corresponds to an *MLCS* of S_1, S_2 and S_3.

In summary, based on our novel problem-solving model *NCSG* and the *MLCS*s mining strategy, Algorithms *ForwardTopSort* and *BackwardTopSort*, we can overcome the defects of the leading dominant-point-based *MLCS* more efficiently and effectively, which is verified by our extensive experiments. However, unfortunately, the theoretical and experimental results also show that the proposed model and the strategy are impractical to big sequences (*e.g.*, the genome sequences with length $10^3 - 10^{11}$) alignments, which motivates us to explore more efficient method. In the next section, we shall present a parallel solution towards this end.

4. A PARALLEL ALGORITHM RLP_MLCS

4.1 Strategies against the Challenges

For the convenience of discussion, we first introduce following key concepts, and give a part of our statistical study in Table 1 over synthetic and real-world biological sequences datasets (see Section 5).

[5]It can be easily proved, we select not to show the detailed proof here due to the limit of space.

Table 1: The total number of dominants of aligned sequences with various lengths and alphabet sizes

	$\|\Sigma\| = 4$				$\|\Sigma\| = 20$		
$\|S_i\|$	N_1	N_2	$N_1/N_2(\%)$	$\|S_i\|$	N_1	N_2	$N_1/N_2(\%)$
25	748	1131	66	50	46	101	45
50	24642	36028	68	100	4714	7150	65
100	2022260	2159886	93	180	262594	316244	83
140	18338817	19546754	93	260	2840112	3188919	89
180	86256948	91655681	94	340	16523403	17940691	92
220	210103979	221208508	94	420	68350594	72256797	94
260	300603814	313783560	95	500	164981091	172150132	95
300	446319986	463079501	96	580	373671085	388190591	96
320	554032873	566657853	97	620	554532996	566558951	97

Definition 6: Given a *NCSG*, a dominant is called *the optimal dominant* from the *NCSG*, if and only if the dominant belongs to 1-level dominants ($\in D^1$) and has minimal coordinates values compared with the other 1-level dominants. if a subgraph *Sub-NCSG*$_i$ of the *NCSG* is constructed starting from optimal dominant i, the *Sub-NCSG*$_i$ is called the *optimal subgraph* of the *NCSG*.

For example, by Definition 6, dominant $(1, 2, 2)$ (the fourth dominant from 1-level dominants D^1) shown in Fig. 3 is an optimal dominant, from which the constructed subgraph (marked red) of the *NCSG* is an optimal subgraph denoted as *Sub-NCSG*$_4$.

The most fundamental challenges to longer and large-scale sequences alignments for *MLCS* algorithms are their unbearable huge time and insufficient space for calculating and storing the massive dominants of the *NCSG*. The statistical data shown in Table 1(N_1 denotes the total number of the dominants from the *optimal subgraph* of the *NCSG*, and N_2 is the total number of the dominants from the *NCSG*) bring out the following facts: 1) With the increase in the length of sequences, the number of dominants from the *NCSG* gets an exponential explosive growth; 2) Most of the dominants come from the optimal subgraph *Sub-NCSG*$_i$, *e.g.*, Table 1 shows that the ratio of N_1/N_2 is as high as 97%. From the above facts, we present the following strategies so as to tackle the challenges. Notably, in line with other parallel algorithms towards big data, our model is based on the intuition that the results of parallel algorithms should be compatible in divide-and-conquer style and support combinative.

Strategy 1: *Successor Table, ST*. One of the fundamental needs in constructing *NCSG* is to search all the immediate successors for each dominant efficiently due to the massive number of dominants in the *NCSG*. To achieve that, according to the searching strategy from [2], the successor tables $\{ST_1, ST_2, ..., ST_d\}$ of the aligned sequences set $T = \{S_1, S_2, ..., S_d\}$ should be built first, *i.e.*, given a sequence $S_l = x_1, x_2, ..., x_n$ from T over a finite alphabet $\Sigma = \{\sigma_1, \sigma_2, ..., \sigma_k\}$, its successor table ST_l is a two-dimensional array, where $ST_l[i, j]$ denotes the element of the i-th row and the j-th column in ST_l, defined as below:

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

Obviously, $ST_l[i, j]$ is in fact the minimal subscript position m of the sequence S_l after position j according to σ_i when $x_m = \sigma_i$, see an example in Fig. 6.

It has been proved in [2] that all the immediate successors of a d-dimensional dominant $p = (p_1, p_2, ..., p_d)$ can be obtained efficiently in $O(d|\Sigma|)$ based on the successor tables. For example, for the dominant $(2, 3, 4)$ of the sequences S_1, S_2 and S_3, we can couple the corresponding lines 1-4 of the second, third and forth columns from the successor tables ST_1, ST_2 and ST_3 to obtain all its immediate successors $(3, 7, 6)$, $(4, 4, 7)$, $(5, 8, 8)$ and $(7, 5, 5)$ corresponding to the characters A, C, G, and T, respectively, while there is no immediate successor for the dominant $(6, 7, 3)$ due to the coupling results $(\_, \_, 6), (8, \_, 7)$, $(\_, 8, 4)$ and $(7, \_, 5)$, which indicates none of them is an immediate successor according to Eq. 3.

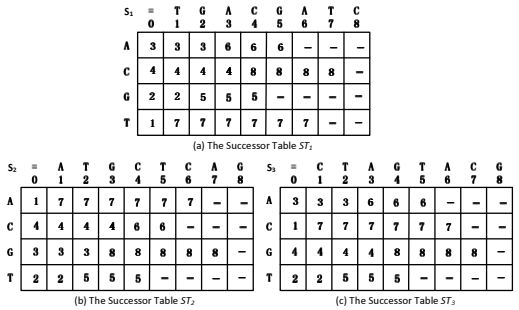


Figure 6: The constructed successor tables ST_1, ST_2 and ST_3 corresponding to the sequences S_1, S_2 and S_3, where the notation "–" indicates $\emptyset$.

Strategy 2: *DM(Index, Point)*. During the construction and processing of *NCSG*, we have to access the coordinates based on the corresponding index, and vice versa, respectively. Hence, we present a bidirectional hash table *DM(Index, Point)*, wherein the *Point* represents a d-dimensional coordinates of a dominant and *Index* is a serial number corresponding to the dominant. As a result, we can compressingly store all the dominants of the *NCSG* with serial numbers instead of their d-dimensional coordinates. The dominants shown in Fig. 7 stored in *DM* are as follows:

DM={$\langle 0, (0, 0, 0)\rangle$, $\langle 1, (3, 1, 3)\rangle$, $\langle 2, (4, 4, 1)\rangle$, $\langle 3, (2, 3, 4)\rangle$, $\langle 4, (1, 2, 2)\rangle$, $\langle 5, (3, 7, 3)\rangle$, $\langle 6, (4, 4, 7)\rangle$, $\langle 7, (7, 5, 5)\rangle$, $\langle 8, (5, 8, 4)\rangle$, $\langle 9, (5, 8, 8)\rangle$, $\langle 10, (3, 7, 6)\rangle$, $\langle 11, (8, 6, 7)\rangle$, $\langle 12, (\infty, \infty, \infty)\rangle$, $\langle 13, (5, 3, 4)\rangle$, $\langle 14, (7, 2, 5)\rangle$, $\langle 15, (6, 7, 6)\rangle$, $\langle 16, (8, 4, 7)\rangle$, $\langle 17, (6, 7, 3)\rangle$, $\langle 18, (7, 5, 2)\rangle$, }.

Strategy 3: *The Optimal Calculation and Reuse of Intermediate Results*. Our extensive experiments and analysis reveal the fact that the optimal subgraph *Sub-NCSG*$_i$ of *NCSG* not only contains most of the dominants of the *NCSG* (45%-97%, average 88%, see Table 1), but also contributes to most of the *MLCS*s (75%-100%, average 85%). For example, the optimal dominant $(1, 2, 2)$ shown in Fig. 3 contributes to three *MLCS*s of the sequences S_1, S_2 and S_3, which is 3 out of 4 *MLCSs*, accounting for 75%. Moreover, as a dominant may locate in different paths of the *NCSG*, *the number of levels of a dominant located in the longest paths (corresponding to MLCSs' paths) of NCSG must be greater than that in the non-longest paths*. For example, the dominants $(4, 4, 7)$ and $(5, 8, 8)$ shown in Fig. 7 simultaneously locate in the non-longest path ($(0, 0, 0) \rightarrow (3, 1, 3) \rightarrow (4, 4, 7) \rightarrow (5, 8, 8) \rightarrow (\infty, \infty, ..., \infty)$, denoted as path-1) and in the longest path ($(0, 0, 0) \rightarrow (1, 2, 2) \rightarrow (2, 3, 4) \rightarrow (4, 4, 7) \rightarrow (5, 8, 8) \rightarrow (\infty, \infty, ..., \infty)$, denoted as path-2). It is clear that the numbers of levels of the dominants $(4, 4, 7)$ and $(5, 8, 8)$ resided in path-1 (being 2 and 3, respectively) are smaller than the numbers of levels they reside in path-2, namely 3 and 4, respectively. With the above observations, in order to save time and space of constructing *NCSG* of aligned sequences, we construct the *NCSG* in the following manner: 1) let D^0={$0, 0, ..., 0$} and perform the procedure $D^0 \rightarrow D^1$; 2) calculate the optimal dominant i from D^1; 3) construct the optimal subgraph *Sub-NCSG*$_i$, the step called

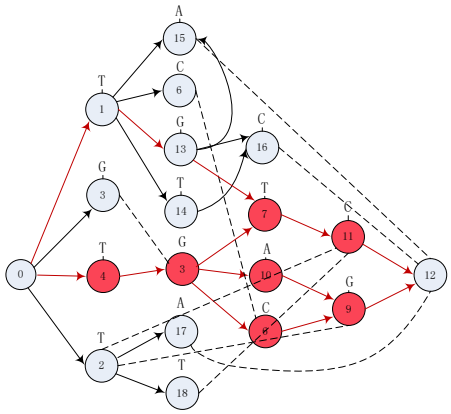


Figure 7: With Strategy 3, we can construct the *NCSG* of S_1, S_2 and S_3 quickly, the red arrows indicate the longest paths of the *NCSG*, the dash lines mean reuse of the intermediate results available (the levels information from the optimized subgraph *Sub-NCSG*$_4$) and red nodes are key points of *Sub-NCSG*$_4$.

optimal calculating; 4) sort and layer the *Sub-NCSG*$_i$ by Algorithm *ForwardTopSort*; 5) identify and remove all of the non-critical dominants on the *Sub-NCSG*$_i$ by Algorithm *BackwardTopSort* resulting in an *optimized Sub-NCSG*$_i$, i.e., all dominants are key points in the *optimized Sub-NCSG*$_i$; 6) construct another subgraph of *Sub-NCSG*$_j$ using the levels information of the dominants from the optimal subgraph *Sub-NCSG*$_i$, which have already been computed in previous steps, thus this step is called *reuse of intermediate results*. Since the *NCSG* of T is equal to the sum of the above subgraphs, with Strategy 3, we can construct the *NCSG* of T quickly and effectively. For convenient of further discussion, the above steps 1)-6) in Strategy 3 are denoted as ***Function OptCalReusing1***(), while the above steps 1)-4) and 6) in Strategy 3 (not including step 5)) are denoted as ***Function OptCalReusing2***(). An example of using Strategy 3 to construct the *NCSG* of the sequences S_1, S_2 and S_3 is shown in Fig. 7.

Example 2: With Strategy 3, construct the *NCSG* of S_1, S_2 and S_3 and find out all of the *MLCS*s of the sequences.

From Fig. 7, we can easily see that dominant 4 corresponding to dominant $(1, 2, 2)$ (see the contents of DM) is an optimal dominant. Hence, with step 3) of Strategy 3, we can construct the optimal subgraph *Sub-NCSG*$_4$ from the optimal dominant 4. And then with steps 4) and 5) of Strategy 3, we can get all the *MLCS*s of *Sub-NCSG*$_4$. Next, with step 6) of Strategy 3, we can easily construct other non-optimal subgraphs *Sub-NCSG*$_1$, *Sub-NCSG*$_2$ and *Sub-NCSG*$_3$ from the non-optimal dominants 1, 2 and 3 in turn and quickly find a new *MLCS* (the path $1 \rightarrow 13 \rightarrow 7 \rightarrow 11 \rightarrow 12$). From above procedures and Fig. 7, we can easily see that the amount of calculations of constructing non-optimal subgraphs (*Sub-NCSG*$_1$, *Sub-NCSG*$_2$ and *Sub-NCSG*$_3$) from the non-optimal dominants 1, 2 and 3 and searching for all of their *MLCS*s is very small due to the reuse of intermediate results available, the levels information of key points of optimized subgraph *Sub-NCSG*$_4$.

Strategy 4: *Subsection Calculation and Serialization*. As discussed above, for longer aligned sequences ($n \geq 10^3$), the challenge is that their *MLCS-DAG* is too large to be stored and calculated in RAM at all (see Table 1), resulting in leading dominant-point-based *MLCS* algorithms' failure in this

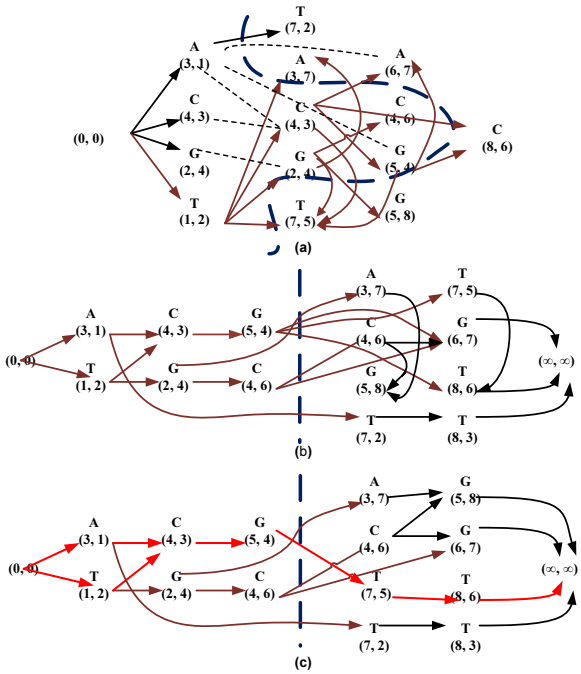


Figure 8: The sketch of searching for all the *MLCS*s of S_4 and S_5 with Strategy 4. ((a) The left side of dashed line is the first subsection *Sub-NCSGs*$_1$ as well as its construction process; (b) The left side of dashed line is serialized *Sub-NCSGs*$_1$ and the right side is the second subsection *Sub-NCSGs*$_2$ as well as its construction process; (c) The concatenation of *Sub-NCSGs*$_1$ and *Sub-NCSGs*$_2$ and all the *MLCS*s with red arrows.)

case. Therefore, we must devise a new subsection calculation method to deal with the case. And yet, we cannot use general subsection method to obtain all the *MLCS*s of the longer sequences in the case because the *MLCS*s of the sequences are certainly not equal to the connection of the *MLCS*s of their sub-sequences in most cases. For example, we split the sequences $S_4 = TGACGATC$ and $S_5 = ATCGTCAG$ into two subsections $S_{41} = TGAC$ and $S_{51} = ATCG$ with $S_{42} = GATC$ and $S_{52} = TCAG$ in their subscripts 4, and all of the *MLCS*s of the sequences S_4 and S_5 is not equal to the connection of the *MLCS*s of the splitted subsections S_{41} and S_{51} with S_{42} and S_{52} as some dominants from the sequences S_4 and S_5 are lost in such splitting manner. Hence, how to split these longer sequences to accurately obtain their *MLCS*s is a non-trivial problem, which motivates us to develop a new subsection calculation method to tackle it. We found that as long as all of the dominants are not lost from aligned sequences and the immediate predecessor-successor relationships among dominants are preserved in the splitting position, we can solve the non-trivial problem efficiently and effectively with our proposed subsection calculation and serialization methods, which are explained with Example 3 and Fig. 8.

Example 3: With Strategy 4, find out all of the *MLCS*s of the sequences $S_4 = TGACGATC$ and $S_5 = ATCGTCAG$.

For the sake of generality, suppose we split the sequences S_4 and S_5 into two subsections in their subscripts 6 on the premise of no loss of dominants of the two sequences. To this end, we first split the dominants those all their coordinates are less than or equal to 6 in the *NCSG* of the

sequences to the first subsection, otherwise to the second subsection. Therefore, call function *OptCalReusing2*() (not function *OptCalReusing1*(), as no dominant is lost from the first subsection), we can construct the first subsection subgraph of the *NCSG*, denoted as *Sub-NCSGs*$_1$ (Fig. 8(a)), and then serialize the constructed *Sub-NCSGs*$_1$ (left part shown in Fig. 8(b)) to the disk. Next, we continue constructing the second subsection subgraph *Sub-NCSGs*$_2$ by calling function *OptCalReusing1*() (right part shown in Fig. 8(b)). To find all of the *MLCS*s of the sequences S_4 and S_5, we first backward sort the *Sub-NCSGs*$_2$ by Algorithm *BackwardSortTop*. Secondly, we deserialize the *Sub-NCSGs*$_1$ and perform *Sub-NCSGs*$_1$ = *Sub-NCSGs*$_1$ $\cup$ *Sub-NCSGs*$_2$ shown in Fig. 8(c), and then continue backward sorting the *Sub-NCSGs*$_1$ by Algorithm *BackwardSortTop*, after which all of the *MLCS*s of the sequences S_4 and S_5 are obtained (denoted as red arrows shown in Fig. 8(c)).

Remark: As discussed in Strategy 4, for the longer sequences, their *NCSG* will be too large to be stored and calculated in memory resulting in the leading dominant-point-based *MLCS* algorithms' failure. To tackle the challenges, we propose Strategy 4 by splitting the *NCSG* of the sequences into subsections. Experimental results justify that the memory overflow for large *NCSG* has been well avoided by Strategy 4 as expected. More interestingly, the efficiency is also improved due to the strategy. For instance, given $|\Sigma| = 20$ with length 300 of 5 aligned sequences, the running time of our algorithm is 3.449s, 1.969s, 1.807s and 0.846s, respectively, if the number of the splitting subsections for the aligned sequences is $2, 3, 4$ and 5, respectively. Such an interesting phenomenon results from the following facts. Firstly, with algorithm *ForwardSortTop* for an arbitrary subsection, many useless edges will be eliminated (see Fig. 4). *e.g.*, with algorithm *ForwardSortTop*, the edges $(0,0,0) \rightarrow (2,3,4)$, $(4,4,1) \rightarrow (8,6,7)$ etc., shown in Fig. 3, have been deleted), which saves much time in *BackwardSortTop* performed afterwards. Secondly, as the number of subsections increases, not only the number of deleted edges increases but also the effect of concurrent execution amplifies, both of which lead to a significant decrease in running time. Notably, although the increase in the number of subsections may lead to additional serialization and deserialization operations, the running time of these operations is in fact negligible comparing to that of *ForwardSortTop* and *BackwardSortTop* for the massive dominants, which has been saved due to Strategy 4.

Strategy 5: *The Multiple Concurrent Execution.* Since the optimal subgraph $NCSG_i$ of the aligned sequences always contributes to most of the *MLCSs*, ranging from 75% to 100%, we can concurrently construct the non-optimal subgraph $NCSG_j$ of the non-optimal dominants with the reuse of intermediate results of the optimal subgraph $NCSG_i$ available (called *multiple concurrent execution*) so as to further improve the efficiency of the proposed algorithm.

4.2 A Novel Parallel MLCS Algorithm

For large-scale and big sequences alignments, based on the above strategies, we present a novel efficient parallel *MLCS* algorithm *RLP_MLCS*, which is shown in Algorithm 1. Notably, on one hand, all the primary procedures of our algorithm, the construction of *NCSG* of aligned sequences, a forward and a backward topological sorting to the *NCSG*, are run in parallel. On the other hand, the parallel efficiency of our algorithm with Strategies 4 and 5 is further enhanced and enlarged. To the best of our knowledge, we are the first to present a parallel *MLCS* algorithm in this manner. For lack of space, we give the framework of our algorithm.

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

1: Build *Successor Tables* $\{ST_1, ST_2, ..., ST_d\}$ of sequences set $T = \{S_1, S_2, ..., S_d\}$ in parallel /* $|S_i| = n$ */
2: **if** *StepLength* $> n$ **then**:
3: Construct *NCSG* of the T with *OptCalReusing1*() in parallel
4: Output all the *MLCS*s of the T
5: **end if**
6: **return**
7: Split the sequences into N subsections in *StepLength*
8: **for** $i \leftarrow 1$ to *N-1* **do**:
9: Construct the subgraph *Sub-NCSGs*$_i$ of the T with *OptCalReusing2*()[6]and Strategy 5
10: Serialize *Sub-NCSGs*$_i$ to the disk
11: **end for**
12: Construct the subgraph *Sub-NCSGs*$_N$ of the T with *OptCalReusing2*() and Strategy 5
13: Backward sort *Sub-NCSGs*$_N$ with Algorithm *BackwardSortTop* in parallel to obtain its optimized *Sub-NCSGs*$_N$
14: **for** $i \leftarrow N-1$ to 1 **do**:
15: Deserialize *Sub-NCSGs*$_i$
16: *Sub-NCSGs*$_i$ $\leftarrow$ *Sub-NCSGs*$_i$ $\cup$ the optimized *NCSGs*$_{i+1}$
17: Backward sort *Sub-NCSGs*$_i$ by Algorithm *BackwardSortTop* in parallel to get its optimized *Sub-NCSGs*$_i$
18: **end for**
19: Output all of the *MLCS*s of the T
20: **return**

4.3 Analysis of Time and Space Complexities

The time complexity of our algorithm in every stage is given first, followed by the total time complexity.

For each sequence S_l of T over alphabet Σ with length n, $O(|\Sigma|n)$ time is required for constructing its successor table ST_l by Eq. 3. Therefore, the time complexity of serially constructing d sequences is $O(d|\Sigma|n)$. The main operations of serially constructing the *NCSG* consist of establishing the predecessor-successor relationship among dominants and computing the in-degree of each point of the *NCSG*. Therefore, the time complexity of serially constructing the *NCSG* should be $O(|E|)$, where $|E|$ is the number of edges in the *NCSG*. Given the points set V of the *NCSG*, since both forward and backward topological sorting need to traverse every point in V, both the forward and backward topological sorting take the time $O(|V|)$.

As a result, the total *serial* time complexity of *RLP_MLCS* is $O(d|\Sigma|n) + O(|E|) + 2O(|V|)$. 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 of the same order as that of $O(|V|)$ and $|E|$ is at most several times of $|V|$, the time complexity of *RLP_MLCS* is $\frac{1}{N_p}O(|V|) + T_{com}$, where T_{com} is the communication overhead of the parallel execution algorithm *RLP_MLCS* and N_p is the number of threads. From the above discussion, we can see that the time complexity of *RLP_MLCS* is linear in $|V|$. It is important to note that $|V|$ should be replaced with $|K| + |V_{si}|$ with subsection calculation of our proposed algorithm, where $|K|$ is the total number of the key points on the *NCSG*, and $|V_{si}|$ is the total number of the dominants on a subsection subgraph *Sub-*

[6] D^1 in Step 2) are the dominants with the minimal coordinates in each subsection, such as the left-most dominants on the right part of dashed line shown in Fig. 8(a); in Step 4) after the forward sorting, we need to update the number of levels for dominants by adding those of their immediate predecessors.

$NCSGs_i$ of the aligned sequences (as the above discussion, we can see that $|K| + |V_{si}| \ll |V|$).

Similarly, the storage space of successor tables is $O(d|\Sigma|n)$, the storage space of the *NCSG* is $O(d|V| + |E|)$, and the space complexity of *RLP_MLCS* is $O(d|\Sigma|n + d|V| + |E|) \approx O(|V| + |E|) = O(|V|)$ (given d aligned sequences, d is a constant). Similar to the analysis in time complexity, with subsection calculation, the space complexity of *RLP_MLCS* should be $O(|K| + |V_{si}|)$.

Notably, the state-of-the-art dominant-point-based *MLCS* algorithms, *FAST_MLCS* [2] with effective pruning techniques and *Quick-DPPAR* [16] with a fast divide-and-conquer technique in the calculation of the dominants, were claimed to be efficient. 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 3). The claimed linear time complexity $O(|MLCS|)$ without considering the time of computing dominants 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 *NCSG*. Thus the complexities of time and space for our proposed algorithm are much lower than those of *FAST_MLCS* and *Quick-DPPAR*.

5. EXPERIMENTAL RESULTS

In experiments, all the algorithms (*FAST_LCS, Quick-DPPAR* and the proposed *RLP_MLCS*) were run on Inspur Corporation K1 800 high-performance key host (Intel Xeon E7-8870, 4 chip, 4 cores/chip, 2 threads/core, 2.80 GHz and 1TB RAM) in the High Performance Computing Center of Xidian University, written in Java with JDK 1.7 and tested on the benchmark datasets provided by real biological sequences datasets ncbi[7] and dip[8], as well as 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 the datasets, respectively, and each group consists of 5 or d sequences. We tested the algorithms 10 times on the above 20 groups of benchmark datasets. Notably, the result *MLCS*s are consistent over all the algorithms as long as the algorithms finish without exception[9]. Therefore, we focus on the comparison in efficiency and report the average running times over 10 runs in milliseconds (ms).

Firstly, we test the performance of our parallel algorithm *RLP_MLCS* and compare it with the state-of-the-art dominant-point-based parallel *FAST_LCS* [2] and *Quick-DPPAR* [16] algorithms (these algorithms were reported to have been implemented in corresponding literature and run on the same hardware platform). Due to space limit, only parts of the experimental results are shown in Table 2 and Table 3, respectively, where '+' stands for the memory overflow such that the algorithm fails in the corresponding case.

Table 2 shows that the time performance of the proposed parallel *RLP_MLCS* algorithm is superior to that of the other two competitors, reaching up to $2-3$ orders of magnitude faster for long sequences. For example, the running time of our algorithm *RLP_MLCS* shown in Table 2 is in average 58 times / 47 times shorter than that of the algorithm *FAST_LCS* (ranging from $6-127$ times / $4-115$ times), and 113 times / 84 times shorter than that of the algorithm *Quick-DPPAR* (ranging from $2-374$ times / $2-257$ times) in different $|\Sigma|$ individually. Moreover, with the increasing length of aligned sequences, the advantage of *RLP_MLCS* in time performance is even more obvious compared with algorithms *FAST_LCS* and *Quick-DPPAR*. Obviously, Table 2 shows that our algorithm *RLP_MLCS* is not only vastly superior to the algorithms *FAST_LCS* and *Quick-DPPAR*, but also is more practical to longer sequences alignments.

Table 2: The running times of *FAST_LCS* (A1), *Quick-DPPAR* (A2) and *RLP_MLCS* (A3) for 5 sequences with various lengths on 32 threads and step length 100.

	$\|\Sigma\| = 4$				$\|\Sigma\| = 20$		
$\|S_i\|$	A1(ms)	A2(ms)	A3(ms)	$\|S_i\|$	A1(ms)	A2(ms)	A3(ms)
50	353	241	56	50	46	37	23
75	1640	1323	143	100	1047	682	98
100	11107	8018	328	140	6749	2137	239
140	121450	92936	1109	180	28875	17886	547
180	1335962	1205232	10480	220	97021	59604	1769
220	+	+	23097	260	355323	296465	2839
300	+	+	52560	300	984187	865325	3796
400	+	+	85071	340	3739639	2567178	9987
500	+	+	167848	400	+	+	19844
1000	+	+	629676	600	+	+	73792
1500	+	+	1038030	800	+	+	107890
2000	+	+	5218596	1000	+	+	214610
2500	+	+	9126758	2000	+	+	571363
3000	+	+	13703431	3000	+	+	703337
3500	+	+	18158274	4000	+	+	931582
4000	+	+	22083068	5000	+	+	1008042
4500	+	+	27080701	6000	+	+	1138462

[1] $|S_i|$: length of sequences

Table 3: The running times of *FAST_LCS* (A1), *Quick-DPPAR* (A2) and *RLP_MLCS* (A3) for d sequences with lengths 100 and 250, on 32 threads with step lengths 100 and 250, respectively.

	$\|S_i\| = 100$ and $\|\Sigma\| = 4$				$\|S_i\| = 250$ and $\|\Sigma\| = 20$		
d	A1(ms)	A2(ms)	A3(ms)	d	A1(ms)	A2(ms)	A3(ms)
3	237	196	87	3	2042	1526	237
4	1818	2631	198	4	28745	21241	953
5	32394	30211	338	5	879434	624429	2753
6	985023	653235	3620	6	9573349	7451484	19688
7	+	+	10927	7	+	+	69440
8	+	+	31532	8	+	+	187688
9	+	+	54676	9	+	+	420122
10	+	+	103598	10	+	+	861038
50	+	+	189629	50	+	+	17925
100	+	+	62526	100	+	+	5536
200	+	+	17275	200	+	+	2132
400	+	+	13560	400	+	+	1074
600	+	+	11532	600	+	+	537
800	+	+	9645	800	+	+	398
1000	+	+	8092	1000	+	+	286

[1] $|S_i|$: length of sequences
[2] d: number of sequences

As discussed above, thanks to the well-designed strategies adopted in *RLP_MLCS*, we have eliminated redundant dominants, got rid of dimension by dimension comparisons of dominants, and saved much storage space compared with the state-of-the-art competitors. Table 3 reveals that these advantages of our algorithm *RLP_MLCS* get more obvious as the number of sequences alignments is increased, e.g., for the different $|\Sigma|$, when testing on 6 sequences with the lengths of 100 (resp., 250) individually, the running time of *RLP_MLCS* has already been up to 272 (resp., 486) times better than that of *FAST_MLCS* and 180 (resp., 378) times better than that of *Quick-DPPAR*. It is a remarkable fact that *FAST_MLCS* and *Quick-DPPAR* cannot work due to memory overflow when the number of aligned sequences is larger than 6. In comparison, our proposed algorithm *RLP_MLCS* can also run correctly and efficiently even though the number of aligned sequences reaches 1000. More-

[7] http://www.ncbi.nlm.nih.gov/nuccore/110645304?report=fasta

[8] http://dip.doe-mbi.ucla.edu/dip/Download.cgi

[9] As shown in Table 2 and 3, in many cases, *FAST_LCS, Quick-DPPAR* will run out of memory.

over, with the increasing number of aligned sequences, the number of dominants firstly increases and then usually decreases. Thus, the experimental results of our algorithm *RLP_MLCS* shown in Table 3 is reasonable, and the algorithm is very suitable for large-scale sequences alignments.

In addition, we further evaluated the speedup of our algorithm *RLP_MLCS* by varying the number of threads. Due to space limit, we briefly describe the high-level results. The results show that our algorithm *RLP_MLCS* achieves a n-early linear speedup, and that the larger d , n, and $|\Sigma|$, the better speedup is.

In a word, the time and space performances of *RLP_MLCS* are not only very superior to the state-of-the-art *FAST_LCS* and *Quick-DPPAR*, but also more practical to longer and large-scale or even big sequences alignments.

6. CONCLUSION

In order to overcome the disadvantages of the leading dominant-point-based *MLCS* algorithms and tackle the challenges of longer and large-scale aligned sequences, we first present a novel general problem-solving model *NCSG* and series of new strategies, *e.g.*, the parallel topological sorting, optimal calculating, reuse of intermediate results, subsection calculation, etc. Based on that, we present a novel real linear and parallel *MLCS* algorithm *RLP_MLCS*. In addition to the external communication overhead T_{com} of parallel execution with N_p threads, theoretical study show that the time and space complexities of the proposed algorithm are only linear to and even smaller than the number of the dominants from aligned sequences, *i.e.*, $\frac{1}{N_p}O(|V|) + T_{com}$ and $O(|V|)$, respectively. In particular, with subsection calculation of our algorithm, $|V|$ should be replaced by $|K| + |V_{si}|$, where $|K|$ is the total number of the key points on the *NCSG*, $|V_{si}|$ is the total number of the dominants on a subsection subgraph *Sub-NCSGs*$_i$, and $|K| + |V_{si}| \ll |V|$. Finally, our algorithm is evaluated by comprehensive experiments on datasets of both random synthetic and real biological sequences. The results show that the proposed problem-solving model *NCSG* and the strategies are efficient and effective, and that the proposed algorithm *RLP_MLCS* not only greatly outperforms the leading state-of-the-art dominant-point-based parallel *MLCS* algorithms available, but also is practical for longer and large-scale sequences alignments.

As part of our future work, we will further improve the efficiency of the proposed algorithm *RLP_MLCS* and explore applications in big sequence analysis.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (No.61472296, 61472297 and 61202179) and National High Technology Research and Development Program (863 Program) (Grant No. 2015AA016007).

7. REFERENCES

[1] A. Apostolico, S. Browne, and C. Guerra. Fast linear-space computations of longest common subsequences. *Theoretical Computer Science*, 92(1):3–17, 1992.

[2] Y. Chen, A. Wan, and W. Liu. A fast parallel algorithm for finding the longest common sequence of multiple biosequences. *BMC Bioinformatics*, 7(Suppl 4):S4, 2006.

[3] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.

[4] I. L. Hofacker, M. A. Huynen, P. F. Stadler, and P. E. Stolorz. Knowledge discovery in RNA sequence families of HIV using scalable computers. In *KDD*, pages 20–25, 1996.

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

[6] W. Hsu and M. Du. Computing a longest common subsequence for a set of strings. *BIT Numerical Mathematics*, 24(1):45–59, 1984.

[7] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences. *Communications of the ACM*, 20(5):350–353, 1977.

[8] G. Ifrim and C. Wiuf. Bounded coordinate-descent for biological sequence classification in high dimensional predictor space. In *KDD*, pages 708–716, 2011.

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

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

[11] 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*, 2012, 2012.

[12] M. Lu and H. Lin. Parallel algorithms for the longest common subsequence problem. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):835–848, 1994.

[13] D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM (JACM)*, 25(2):322–336, Apr. 1978.

[14] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[15] D. Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences*, 69(1):4–6, 1972.

[16] Q. Wang, D. Korkin, and Y. Shang. A fast multiple longest common subsequence (MLCS) algorithm. *Knowledge and Data Engineering, IEEE Transactions on*, 23(3):321–334, 2011.

[17] 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*, volume 1, pages 499–504, 2010.

[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. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):862–870, 2013.

[19] T. K. Yap, O. Frieder, and R. L. Martino. Parallel computation in biological sequence analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):283–294, 1998.

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