A Fast Multiple Longest Common Subsequence (MLCS) Algorithm

Qingguo Wang, Dmitry Korkin, and Yi Shang, *Senior Member*, *IEEE*

Abstract—Finding the longest common subsequence (LCS) of multiple strings is an NP-hard problem, with many applications in the areas of bioinformatics and computational genomics. Although significant efforts have been made to address the problem and its special cases, the increasing complexity and size of biological data require more efficient methods applicable to an arbitrary number of strings. In this paper, we present a new algorithm for the general case of multiple LCS (or MLCS) problem, i.e., finding an LCS of any number of strings, and its parallel realization. The algorithm is based on the dominant point approach and employs a fast divide-and-conquer technique to compute the dominant points. When applied to a case of three strings, our algorithm demonstrates the same performance as the fastest existing MLCS algorithm designed for that specific case. When applied to more than three strings, our algorithm is significantly faster than the best existing sequential methods, reaching up to 2-3 orders of magnitude faster speed on large-size problems. Finally, we present an efficient parallel implementation of the algorithm. Evaluating the parallel algorithm on a benchmark set of both random and biological sequences reveals a near-linear speedup with respect to the sequential algorithm.

Index Terms—Longest common subsequence (LCS), multiple longest common subsequence (MLCS), dynamic programming, dominant point method, divide and conquer, parallel processing, multithreading.

1 INTRODUCTION

The multiple longest common subsequence problem (MLCS) is to find the longest subsequence shared between two or more strings. It is an NP-hard problem [35], with important applications in many fields, such as information retrieval and computational biology [3], [9], [42], [44]. For over 30 years, significant efforts have been made to find efficient algorithms for the MLCS problem. Many of them, however, address either the simplest case of MLCS of two strings, also known as the longest common subsequence (LCS) problem [24], [36], [39], [45], or the problem's special case of three strings [21], [22]. Although several methods have been proposed for the general case of any given number of strings [11], [22], [27], [29], they could benefit greatly from improving their computation times. A method that solves the general MLCS problem efficiently can be applied to many computational biology and computational genomics problems that deal with biological sequences [8], [28], [29], [43]. With the increasing volume of biological data and prevalent usage of computational sequence analysis tools, we expect that the general MLCS algorithm will have a significant impact on computational biology methods and their applications.

In this paper, we present a fast algorithm for the MLCS problem for any given number of sequences. The new method is based on the dominant point approach [21], [22], [29]. Methods that solve MLCS using dominant points are found to be more efficient, reducing the size of search space by orders of magnitude, compared to classical dynamic programming methods [22]. In our method, we implement a divide-and-conquer technique to construct dominant point sets efficiently. Unlike other MLCS algorithms, including FAST-LCS [11] and parMLCS [29] that minimize entire dominant point set, our method partitions them into independent subsets, where an efficient divide-and-conquer technique is applied. Compared to the existing state-of-the-art MLCS algorithms, our dominant-point algorithm is significantly faster on the larger size problems, for instance, for a set of strings of 1,000 letters each and longer. We have also developed an efficient parallel version of the algorithm, achieving a near-linear speedup.

The paper is organized as follows: In the next section, we formally define the MLCS problem and review the related work of dynamic programming and dominant-point-based approaches and their parallelization. In Section 3, we present a new dominant point algorithm and analyze its complexity. In Section 4, we introduce a new parallel algorithm and its complexity analysis. In Section 5, we compare the performance of the new sequential and parallel algorithms with some of the best existing algorithms on a comprehensive benchmark set. Finally, in Section 6, we discuss the results and suggest future research directions.

- *Q. Wang and Y. Shang are with the Department of Computer Science, University of Missouri, Columbia, MO 65211. E-mail: qwp4b@mail.missouri.edu, shangy@missouri.edu.*
- *D. Korkin is with the Informatics Institute and Department of Computer Science, University of Missouri, 207 Engineering Building West, Columbia, MO 65211. E-mail: korkin@korkinlab.org.*

Manuscript received 7 May 2009; revised 23 Sept. 2009; accepted 2 Nov. 2009; published online 26 July 2010.
Recommended for acceptance by J. Zobel.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2009-05-0411.
Digital Object Identifier no. 10.1109/TKDE.2010.123.

2 MLCS PROBLEM FORMULATION AND EXISTING METHODS

In this section, we define the MLCS problem and review existing sequential and parallel methods for solving it.

2.1 Problem Definition

Definition 1. *Let* $\mathbf{a}$ *be a sequence of length* n *over a finite alphabet* Σ: $\mathbf{a} = s_1\ s_2\ \dots\ s_n\ ,\ \ s_i \in \Sigma$. *Sequence* $\mathbf{b} = s_{i_1}\ s_{i_2}\ \dots\ s_{i_k}$ *is called a* ***subsequence*** *of* $\mathbf{a}$, *if* $\forall j,\ \ 1 \le j \le k$:

TABLE 1
Common Structures in Computational Biology and Their Approximate Size Ranges

Biological data	**Alphabet, Σ**	**Sequence length**
Protein	$\{A, C, \ldots, W\}, \lvert\Sigma\rvert = 20$	$\sim 10^2$–10^4 [10], [51]
RNA	$\{A, C, G, U\}$	~ 10–10^4 [16]
Genome	$\{gene_1, gene_2, \ldots, gene_k\}$	~ 10–10^4 [7], [38]
Genome	$\{A, C, G, T\}$	$\sim 10^4$–10^{11}[20], [32]

$$1 \le i_j \le n,$$

and for all r and t, $\quad 1 \le r < t \le k$:

$$i_r < i_t.$$

Definition 2. *Let $S = \{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_d\}$ be a set of sequences over a finite alphabet Σ. The **MLCS** for set S is a sequence $\mathbf{b}$ such that:*

1. *$\mathbf{b}$ is a subsequence of $\mathbf{a}_i$ for each i;*
2. *$\mathbf{b}$ is the longest among all sequence satisfying point 1.*

In general, there may be more than one MLCS. For example, given three sequences,

$$\mathbf{a}_1 = i\ n\ f\ o\ r\ m\ a\ t\ i\ c\ s,$$
$$\mathbf{a}_2 = p\ r\ o\ t\ e\ o\ m\ i\ c\ s,$$
$$\mathbf{a}_3 = a\ r\ i\ t\ h\ m\ e\ t\ i\ c\ s.$$

one of the multiple longest common subsequences is $\mathbf{b}_1 = r\ m\ i\ c\ s$ and another is $\mathbf{b}_2 = r\ t\ i\ c\ s$. The LCS problem is a special case of MLCS for two sequences. Several important applications in computational biology can be formulated as MLCS problems, and the alphabet sizes and sequence lengths vary significantly, depending on the biological domain (Table 1).

2.2 Dynamic Programming Methods

Classical methods for the MLCS problem are based on dynamic programming [41], [45]. In its simplest case, given two sequences $\mathbf{a}_1$ and $\mathbf{a}_2$ of length n_1 and n_2, respectively, a dynamic programming algorithm iteratively builds an $n_1 \times n_2$ score matrix L in which $L[i, j], 0 \le i \le n_1, 0 \le j \le n_2$, is the length of an LCS between two prefixes $\mathbf{a}_1[1, \ldots, i]$ and $\mathbf{a}_2[1, \ldots, j]$. Specifically, the score matrix L is defined as follows:

$$L[i,j] = \begin{cases} 0, & \text{if i or j} = 0, \\ L[i-1, j-1] + 1, & \text{if } a_1[i] = a_2[j], \\ \max(L[i, j-1], L[i-1, j]), & \text{if } a_1[i] \ne a_2[j]. \end{cases} \quad (1)$$

The definition of the matrix L can be naturally generalized to a case of N sequences: for each position $L[i_1, i_2, \ldots, i_N]$, its value is defined through the immediately preceding positions.

Once the score matrix L is computed, we can extract MLCS by tracing back from the end point $[n_1, n_2]$ to the starting point $[0, 0]$. Let $|MLCS|$ be the length of an MLCS. It can be inferred that $|MLCS|$ is equal to the maximal value in L. Fig. 1 shows an example of score matrix L for two sequences $\mathbf{a}_1 = GATTACA$ and $\mathbf{a}_2 = GTAATCTAAC$.

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

Fig. 1. The matrix L computed using (1) for two sequences $\mathbf{a}_1 = GATTACA$ and $\mathbf{a}_2 = GTAATCTAAC$. The regions of the same entry values are bounded by contours; the corner points of contours are dominant points and are circled.

It is straightforward to calculate all entries in L using dynamic programming. The resulting algorithm has time and space complexity of $O(n^d)$ for d sequences of length n [25]. Various approaches have been introduced to reduce the complexity of dynamic programming [2], [24], [36], [39]. Unfortunately, these approaches primarily address a special case of two sequences.

2.3 Dominant Point Methods

Let L be the score matrix for a set of d sequences $\{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_d\}$ over a finite alphabet Σ. A point p in matrix L is denoted as $p = [p_1, p_2, \ldots, p_d]$, where each p_i is a coordinate of p for the corresponding string $\mathbf{a}_i$. The value at position p of the matrix L is denoted as $L[p]$.

Definition 3. *A point $p = [p_1, p_2, \ldots, p_d]$ in L is called a match if*

$$\mathbf{a}_1[p_1] = \mathbf{a}_2[p_2] = \cdots = \mathbf{a}_d[p_d].$$

For example, for matrix L in Fig. 1, points $[0, 0]$ and $[1, 2]$ are two matches, corresponding to symbols G and A, respectively.

Definition 4. *A point $p = [p_1, p_2, \ldots, p_d]$ dominates another point $q = [q_1, q_2, \ldots, q_d]$, if $p_i \le q_i$, for all $i = 1, 2, \ldots, d$. If p dominates q, we denote this relation as $p \le q$.*

Similarly, a point $p = [p_1, p_2, \ldots, p_d]$ strongly dominates another point $q = [q_1, q_2, \ldots, q_d]$, if $p_i < q_i$, for all $i = 1, 2, \ldots, d$. We denote this relation as $p < q$.

A point $p = [p_1, p_2, \ldots, p_d]$ *does not dominate* a point $q = [q_1, q_2, \ldots, q_d]$ (denoted as $p \not\le q$), if $\exists i, \ 1 \le i \le d, \ q_i < p_i$. Note that $p \not\le q$ does not necessarily imply $q \le p$, i.e., for some points p and q, $p \not\le q$ and $q \not\le p$ may be true at the same time.

Definition 5. *A match p is called a k-dominant point, or simply k-dominant, or dominant at level k, if*

1. *$L[p] = k$;*
2. *there is no other match $q, q \ne p$, satisfying 1 dominates p $(q \le p)$.*

The set of all k-dominants is denoted as D^k. The set of all dominant points (that is, k-dominants for all k) is denoted as D.

In Fig. 1, regions of the same entry values are bounded by contours. Corner points of these contours are dominant

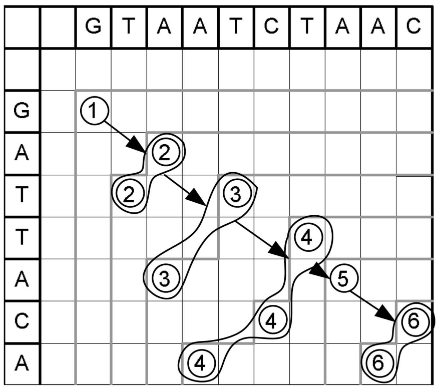

Fig. 2. The process of dominant point approach for two sequences $\mathbf{a}_1 = GATTACA$ and $\mathbf{a}_2 = GTAATCTAAC$. Broken lines are contours or boundary of different level of points. Dominant points at the same level are encircled together. Arrows in the figure point from the set of k-dominants to the set of $(k+1)$-dominants, $1 \le k \le |MLCS| - 1$.

points. They are the critical points sufficient to define the overall contour shape.

The main idea behind the dominant point approach is to identify exclusively the dominant point values instead of identifying values of all positions in matrix L [6], [12], [24]. Specifically, the dominant point approach first computes the set of all 1-dominants. As a general iteration step, a set of (k+1)-dominants is calculated from a set of k-dominants, $1 \le k \le |MLCS| - 1$. Thus, by advancing from one contour to another, all dominant points can be obtained. A basic pseudocode for the dominant point algorithm is provided in Appendix A.

Fig. 2 illustrates the process of dominant point approach for the above two sequences $\mathbf{a}_1$ and $\mathbf{a}_2$. In Fig. 2, the dominant points of the same level are encircled together. One dominant point $[1,1]$ of level 1 is found at the first step; at the second step, two dominant points $[2,3]$ and $[3,2]$ of level 2 are detected based on the previous dominant point set $\{[1,1]\}$, and so on. Arrows in the figure point from the set of k-dominants to the set of $(k+1)$-dominants, $1 \le k \le |MLCS| - 1$.

The dominant point approach has been successfully applied to a case of two sequences [2], [12], [26]. In [22], three dominant point algorithms for three or more sequences were proposed. One of the algorithms, Algorithm A [22], which was designed specifically for MLCS problems of three sequences, is overwhelmingly faster than traditional dynamic programming algorithms for three sequences. However, Algorithm A finds dominant point sets by enumerating points of the same coordinate values in each dimension. As a result, its complexity increases rapidly with growing number of sequences. The other algorithm, Hakata and Imai's C algorithm [22], works for arbitrary number of strings. It is similar to another MLCS algorithm published recently, FAST-LCS [11], in that they both use a pairwise comparison algorithm to compute the dominant points.

2.4 Parallel MLCS Methods

Many parallel MLCS algorithms have been developed to speed up the computation [49], [50]. The majority of them are designed for LCS problem of two sequences. On CREW-PRAM model, Apostolico et al. [1] designed an $O(\log m \log n)$ time algorithm with $O(mn/\log m)$ processors, where m and n are lengths of two input strings and $m \le n$. Lu and Lin [33] proposed two parallel algorithms: one is an $O(\log^2 m + \log n)$ time algorithm with $mn/\log m$ processors and the other is an $O(\log n)$ time algorithm with $mn/\log n$ processors when $\log^2 m \log\log m \le \log n$. Babu and Saxena [4] improved these algorithms, proposing an $O(\log m)$ time algorithm with mn processors and an $O(\log^2 n)$ time algorithm. Some parallel algorithms use systolic arrays. Luce and Myoupo [34] derived a $n + 3m + p$ time algorithm with an array of $m(m+1)/2$ cells. Freschi and Bogliolo [19] computed the LCS between run-length-encoded (RLE) strings. Their algorithm executed in $O(M+N)$ steps on a systolic array of $m + n$ units, where M and N are the number of runs in their RLE representation. Optical bus has also been used by some parallel algorithms. On LARPBS, Xu et al. [49] presented an algorithm that used p processors and takes $O(mn/p)$ time, where $1 \le p \le \max(m,n)$.

Only a few parallel algorithms have been proposed for MLCS problems of three or more sequences. The first parallel approach to tackle the general MLCS problem [27] could not achieve a consistent speedup on the test set of multiple sequences. In the most recent approaches, FAST-LCS [11] and parMLCS [29], a near-linear speedup was reached for a large number of sequences. parMLCS is a parallel version of Hakata and Imai's C algorithm [22].

3 A New Fast MLCS Algorithm, Quick-DP

In this section, we present a new dominant point algorithm for MLCS problem of any number of sequences. For convenience, we assume below that $\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_d$ are sequences over alphabet Σ, and the lengths of all sequences are equal to n.

3.1 Sequential Algorithm Design

Definition 6. *A match p is called an s-parent (a parent with respect to the s symbol $s \in \Sigma$) of a point q, if $q < p$ and there is no other match r of s such that $q < r < p$. We denote the s-parent of q as $q(s)$. The set of s-parent for q is denoted as $Par(q,s)$, i.e., $Par(q,s) = \{q(s)\}$. The set of all s-parents for a set of points A is denoted as $Par(A,s)$. The set of all parents, $\cup_{s\in\Sigma} Par(A,s)$, for A is denoted as $Par(A,\Sigma)$.*

Definition 7. *A point p in a set of points A is called a minimal element of A, if for all $q \in A - \{p\} : q \not\le p$. If $|A| = 1$, then its single element is defined to be a minimal element of A. The set of minimal elements is called the minima of A.*

It has been proven in [22] that $(k+1)$-dominants, $D^{k+1}, 0 \le k \le |MLCS| - 1$, constitute exactly the minima of the parent set $Par(D^k, \Sigma)$ of k-dominants D^k, i.e.,

$$D^{(k+1)} = Minima(Par(D^k, \Sigma)),$$

where $Minima()$ is an algorithm that returns the minima of a set of points.

The pseudocode of our sequential dominant point algorithm Quick-DP is presented in Fig. 3.

Quick-DP consists of two parts. In the first part, the set of all dominants is calculated iteratively, starting from 0-dominant set (containing one element). The set of $(k+1)$-dominants $D^{(k+1)}$ is obtained based on the set of k-dominants D^k. In the second part, a path corresponding to an MLCS is

```
Algorithm Quick-DP ({$\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_d$}, $\Sigma$)
Computing dominant points
Preprocessing;
$D^0 = \{[-1,-1,\ldots,-1]\}$; $k = 0$; $Par_s = \emptyset$ for all $s \in \Sigma$;
while $D^k$ not empty do {
  for $q \in D^k$ do {
    $B$ = Minima(Par($q, \Sigma$));
    for $s \in \Sigma$ do{
      $Par_s = Par_s \cup \{q(s) \mid q(s) \in B\}$; }}
  $D^{k+1} = \cup_{s\in\Sigma}$ Minima ($Par_s$) ;
  $k = k + 1$; }
Finding a MLCS
pick a point $p = [p_1, p_2, \ldots, p_d] \in D^{k-1}$;
while $k - 1 > 0$ do {
  current LCS position = $\mathbf{a}_1[p_1]$;
  pick a point $q$ such that $p \in Par(q, \Sigma)$ ;
  $p = q$;
  $k = k - 1$; }
```

Fig. 3. The pseudocode of our sequential algorithm Quick-DP.

found by tracing back through sets of dominant points obtained in the first part of the algorithm, starting with an element from the last dominant set. If needed, all MLCS can be enumerated systematically.

Quick-DP follows a two-step procedure to compute the minima of the parent set $Par(D^k, \Sigma)$: 1) For each dominant point $q \in D^k$, compute $Minima(Par(q, \Sigma))$ and take a union of all such sets,

$$Par_s = \{q(s) \mid q(s) \in Minima(Par(q, \Sigma)), q \in D^k\}, s \in \Sigma.$$

2) Compute the union of nonoverlapping sets $Minima(Par_s), s \in \Sigma$.

Theorem 1 below demonstrates that this two-step procedure correctly construct D^{k+1} from D^k by proving that $Minima(Par(D^k, \Sigma)) = \cup_{s\in\Sigma} Minima(Par_s), s \in \Sigma$.

Lemma 1. *Assume that p and q are k-dominants. If $p(s_i) < q(s_j)$ for $i \neq j$, then $p(s_j) \leq q(s_j)$.*

The proof of Lemma 1 can be found in [22].

Theorem 1. *Let $Minima()$ be an algorithm that returns the minima of a set of points. Then,*

$$Minima(Par(D^k, \Sigma)) = \cup_{s\in\Sigma} Minima(Par_s),$$

where

$$0 \leq k \leq |MLCS|, s \in \Sigma,$$
$$Par_s = \{q(s) \mid q(s) \in Minima(Par(q, \Sigma)), q \in D^k\}.$$

Proof. First, we prove that

$$Minima(Par(D^k, \Sigma)) \subseteq \cup_{s\in\Sigma} Minima(Par_s).$$

For any q, $q \in D^k$, if $q(s) \in Minima(Par(D^k, \Sigma))$, then $q(s) \in Minima(Par_s)$, since $q(s)$ is a global minima in $Par(D^k, \Sigma)$. Second, we prove by contradiction that $Minima(Par(D^k, \Sigma)) \supseteq \cup_{s\in\Sigma} Minima(Par_s)$. For $q \in D^k$, $s_j \in \Sigma$, assume that $q(s_j) \in \cup_{s\in\Sigma} Minima(Par_s)$ and $q(s_j) \notin Minima(Par(D^k, \Sigma))$. Then, there has to be $p(s_i), p(s_i) \in Minima(Par(D^k, \Sigma))$, dominating $q(s_j)$, i.e., $p(s_i) \leq q(s_j)$. p and q cannot be the same; otherwise, $Minima()$ would remove $q(s_j)$ from $Par(q, \Sigma)$, thus eliminating $q(s_j)$ from $\cup_{s\in\Sigma} Minima(Par_s)$. i is different from j; otherwise, $Minima()$ would remove $q(s_j)$ from Par_{s_j}. Hence, $p(s_i) < q(s_j)$, since $p(s_i)$ and $q(s_j)$ are different matches. From Lemma 1, it can then be inferred that $p(s_j) \leq q(s_j)$. It means that $q(s_j)$ cannot be in Par_{s_j}, contradicting the assumption that $q(s_j) \in \cup_{s\in\Sigma} Minima(Par_s)$. □

Both $Par(q, \Sigma)$ of q, $q \in D^k$, and Par_s, $s \in \Sigma$, are significantly smaller than $Par(D^k, \Sigma)$. By avoiding the minimization of entire $Par(D^k, \Sigma)$, computation time is saved. More importantly, by breaking the minimization problem $Minima(Par(D^k, \Sigma))$ into independent parts, i.e., $Minima(Par(q, \Sigma)), q \in D^k$, and $Minima(Par_s), s \in \Sigma$, it is possible to use multiple processors simultaneously to solve the minimization problem, as illustrated in Section 4.

3.2 Implementation Details and Complexity Analysis

We add a preprocessing step in the beginning of the algorithm to efficiently find all parents of each dominant point. We calculate a preprocessing matrix $\mathbf{T} = \{T[s, j, i]\}$, $s \in \Sigma, 0 \leq j \leq max_{1\leq k\leq d}\{|\mathbf{a}_k|\}$, $1 \leq i \leq d$, where each element $T[s, j, i]$ specifies the position of the first occurrence of character s in the ith sequence, starting from the $(j + 1)$st position in that sequence. If s does not occur any more in the ith sequence, the value of $T[s, j, i]$ is equal to $1 + max_{1\leq k\leq d}\{|\mathbf{a}_k|\}$. With the matrix $\mathbf{T}$, the s-parent $p = [p_1, p_2, \ldots, p_d]$ of a point $q = [q_1, q_2, \ldots, q_d]$ can be calculated in $O(d)$ time, using the formula $p_i = T(s, q_i, i), 1 \leq i \leq d$. The calculation of this preprocessing matrix $\mathbf{T}$ takes $O(n\,|\Sigma|\,d)$ time, where $|\Sigma|$ is the size of alphabet Σ.

The pseudocode of the procedure $Minima()$ mentioned above is provided in Appendix B. $Minima()$ implements a fast divide-and-conquer algorithm, which is based on the following theorem:

Lemma 2. *For $d = 2$, the minima of N points in the d-dimensional space requires time $O(N)$ if the point set is sorted.*

The proof of Lemma 2 is given in [5], [30].

Theorem 2. *For $d \geq 3$, the minima of N points in the d-dimensional space can be computed in $O(dN \log^{d-2} N)$ time by a divide-and-conquer algorithm. The computation time is $O(dN \log^{d-2} n)$ if the sequence length $n \leq N$.*

Proof. The proof can be derived from [5], [22], [30]. Let $T(N, d)$ denote the computation time of a divide-and-conquer algorithm on a set of N points in d-dimensional space. $T(N, d)$ consists of three parts: 1) time for dividing N points into two subsets R and Q each of $N/2$ points in a way that the d-dimensional coordinates of points in R are greater than those of points in Q. This step can be accomplished in $O(N)$ time using any fast median algorithm; 2) $2T(N/2, d)$ for minimizing R and Q individually; and 3) $T(N, d - 1)$ time for removing points in R that are dominated by points in Q. By combining these, we have the following recurrence formula:

$$T(N, d) = O(N) + 2T(N/2, d) + T(N, d - 1). \qquad (2)$$

Lemma 2 indicates that $T(N,2) = O(N)$ for N sorted points. The sorting of points takes $\Omega(N \log N)$ time. If we presort the points at the beginning of the algorithm and maintain the order of points later in each step of the algorithm, then we don't have to sort points each time. This gives an $O(N)$ algorithm to find the minima of N points for $d = 2$. Using $d = 2$ as basis for induction on d, we can solve (2) and establish that

$$T(N,d) = O(dN \log^{d-2} N). \quad (3)$$

Note that the coordinates of points are integers between 1 and n. If the sequence length n is less than the number of points N, i.e., $n \le N$, after at most $O(\log n)$ recursions instead of $O(\log N)$ points in subproblems come to have the same coordinate. Therefore, we conclude the proof by deriving the following equation:

$$T(N,d) = O(dNlog^{d-2}n). \quad (4)$$

□

Now, we estimate the computational complexity of Quick-DP. Let $|D|$ be the size of dominant point set D. From Theorem 2, we can derive that it takes $O(|\Sigma|\ d\ \log^{d-2}|\Sigma|)$ time to compute the minima of each parent set $Par(q,\Sigma)$, $q \in D$, and $O(|D|\ d \log^{d-2} n)$ time to compute the minima of each s-parent set Par_s, $s \in \Sigma$. Hence, the time complexity $\dot{T}_{seq}$ of Quick-DP is

$$\dot{T}_{seq} = O(\ n\ |\Sigma|\ d + |D|\ |\Sigma|\ d\ (\ \log^{d-2} n + \log^{d-2}|\Sigma|)). \quad (5)$$

Based on experimental evaluation of $|D|$ and the estimated complexities of Hakata and Imai's C algorithm [22], FAST-LCS [11] and parMLCS [29], our algorithm is significantly faster than C algorithm, FAST-LCS and parMLCS, when the sequences are long, i.e., large n.

The space complexity of the algorithm can be easily estimated as $O(|D|d + n|\Sigma|d)$. While an accurate estimation of the dominant set size remains an open question, the following section shows that the size of D is orders of magnitude smaller than the number of points in L.

4 A New Parallel MLCS Algorithm

4.1 Parallel Algorithm Design

As shown in the previous section, calculating the set of $(k+1)$-dominants D^{k+1} requires computing the minima of parent set $Par(q,\Sigma)$, for $q \in D^k$, and the minima of s-parent set Par_s, $s \in \Sigma$, of D^k. These sets can be calculated independently in parallel. Based on this observation, we propose the following parallelization of the sequential algorithm.

Given $N_p + 1$ processors, the parallel algorithm uses one as the master and N_p as slaves and performs the following seven steps:

1. The master processor computes D^0.
2. Every time the master processor computes a new set D^k of k-dominants ($k = 1,\ 2,\ 3,\ \ldots$), it distributes them evenly among all slave processors.
3. Each slave computes the set of parents and the corresponding minima of k-dominants that it has, and then, sends the result back to the master processor.
4. The master processor collects each s-parent set Par_s, $s \in \Sigma$, as the union of the parents from slave processors and distributes the resulting s-parent set among slaves.
5. Each slave processor i is assigned to find the minimal elements only of one s-parent set Par_s.
6. Each slave processor i computes the set D_i^{k+1} of $(k+1)$-dominants of Par_s and sends it to the master.
7. The master processor computes $D^{k+1} = D_1^{k+1} \cup D_2^{k+1} \cup \ldots \cup D_{N_p}^{k+1}$ and goes to step 2.

```
Algorithm Quick-DPPAR ($\{a_1, a_2, \ldots, a_d\}, \Sigma, N_p$)
Proc_0 : Preprocessing; $D^0 = \{[-1,-1,\ldots,-1]\}$; $k = 0$;
while $D^k$ not empty do {
  Proc_0 : distribute elements of $D^k$
  Each processor, Proc_i, $1 \le i \le N_p$, performs:
    get $D_i^k$ from Proc_0;
    for $q \in D_i^k$ do {
      B = Minima(Par(q, $\Sigma$));
      for s $\in \Sigma$ do{
        $Par_{is} = Par_{is} \cup \{q(s) \mid q(s) \in B\}$; }}
    Send $Par_{is}, s \in \Sigma$, to Proc_0;
  Proc_0 : calculate $Par_s = \cup_{1 \le i \le N_p} Par_{is}, s \in \Sigma$;
  Proc_0 : distribute $Par_s, s \in \Sigma$;
  Each processor, Proc_i, $1 \le i \le N_p$, performs:
    get $Par_s, s \in \Sigma$;
    $D_i^{k+1}$ = Minima($Par_s$);
    send $D_i^{k+1}$ to Proc_0;
  Proc_0 : defines $D^{k+1} = \cup_{1 \le i \le N_p} D_i^{k+1}$ ;
  k = k + 1; }
```

Fig. 4. The pseudocode of our parallel algorithm Quick-DPPAR.

The pseudocode of the parallel algorithm Quick-DPPAR is presented in Fig. 4.

In Quick-DPPAR, each s-parent set $Par_s, s \in \Sigma$, of D^k is assigned to a slave processor to compute the minima of Par_s using our divide-and-conquer method. So, as many as $|\Sigma|$ slave processors can work simultaneously in this step. To utilize more than $|\Sigma|$ processors, it is necessary to parallelize the divide-and-conquer algorithm. Two observations described previously in the proof of Theorem 2 are intrinsic for the parallel divide-and-conquer algorithm: 1) each set S is evenly divided into two subsets Q and R to be minimized independently; and 2) the minimization of Q and R is much more time-consuming than the dividing step (linear time).

Based on these observations, we developed a parallel version of the divide-and-conquer algorithm. The main idea of the algorithm is as follows: Let $Proc_0$ be the master processor that starts the divide-and-conquer algorithm. $Proc_0$ split the set of N dominant points evenly into two subsets Q and R and assign them to two children processors, say $Proc_q$ and $Proc_r$, respectively, for the computation of their minima. $Proc_q$ and $Proc_r$ can have their own children too. Thus, during the recursive execution of the program, a binary tree is formed based on this parent-children relationship, with processors as tree nodes. Given m processors, the depth of the tree is at most $\log m$. The leaf processors of the tree run sequential divide-and-conquer algorithm. The pseudocode of the parallel divide-and-conquer algorithm is provided in Appendix C.

4.2 Time Complexity Analysis

We first evaluate the theoretical time complexity of the parallel divide-and-conquer algorithm.

Lemma 3. *For $d \geq 3$, the minima of N dominant points in d-dimensional space can be computed by the parallel divide-and-conquer algorithm using m processors in $T_m(N,d) = \frac{1}{m}T(N,d)$ time, if $mn \leq N$, where $T(N,d)$ is the running time of the sequential divide-and-conquer algorithm.*

Proof. From the description of the parallel divide-and-conquer algorithm above, we can derive the following recurrence formula:

$$T_m(N,d) = \begin{cases} O(N) + T_{m/2}(N/2,d) + T_m(N,d-1), & m>1, \\ O(N) + 2T(N/2,d) + T(N,d-1), & m=1. \end{cases} \tag{6}$$

Using (3) as basis for induction on m, we can solve (6) and establish that

$$T_m(N,d) = O\left(\frac{dN}{m} log^{d-2} \frac{N}{m}\right). \tag{7}$$

Because the coordinates of points are integers between 1 and n and given $n \leq \frac{N}{m}$, the number of recursion steps is at most $\log n$ instead of $\log \frac{N}{m}$. Hence, we have

$$T_m(N,d) = \frac{1}{m}T(N,d). \tag{8}$$

Based on Lemma 3, we can estimate the theoretical time complexity of the main part of the parallel algorithm (without its preprocessing and postprocessing stages). □

Theorem 3. *Assume $|\Sigma| \leq n \leq |D|$ and $nN_p \leq |D^k|$ for each level k. Let $\dot{T}_{seq}$ be the time complexity of the main part of sequential algorithm Quick-DP as in (5). Then, the time complexity $\dot{T}_{par}$ of the main part of parallel algorithm is*

$$\dot{T}_{par} = \dot{T}^{comp}_{par} + \dot{T}^{comm}_{par},$$

where

$$\dot{T}^{comp}_{par} \leq \frac{1}{N_p}\left(1 + \frac{2N_p}{|\Sigma| \ \log^{d-2} n}\right) \dot{T}_{seq},$$
$$\dot{T}^{comm}_{par} \leq 4d|D| \ (t_{startup} + t_{data}),$$

and $t_{startup}$ is the startup time or message latency and t_{data} is the communication time to send a standard unit of data.

Proof.

1. The communication steps in Quick-DPPAR are 3, 4, 9, 11, 12, and 14. Thus, the overall communication time $\dot{T}^{comm}_{par}$ can be obtained directly from the analysis of communication time for each of the above steps.
2. The computation time $\dot{T}^{comp}_{par}$ can be expressed as

$$\dot{T}^{comp}_{par} = \hat{T}^{comp}_{common} + \hat{T}^{comp}_{par}.$$

In this formula, $\hat{T}^{comp}_{common}$ is the computation time of those steps of Quick-DPPAR that are also performed in the sequential algorithm Quick-DP, and $\hat{T}^{comp}_{par}$ is the computation time of the steps performed exclusively by Quick-DPPAR algorithm (and thus, specifying the extra amount of work done by the parallel processors, when running Quick-DPPAR).

3. $\hat{T}^{comp}_{common}$ is comprised out of the running times for steps 5, 6, 7, 8, and 13 and can be estimated as[1]

$$\hat{T}^{comp}_{common} \leq \frac{1}{N_p} \dot{T}_{seq}.$$

It is important to note that the amount of work done by N_p parallel processors during those steps in Quick-DPPAR is equal to the amount of work which is done by a single processor, when running Quick-DP algorithm.

4. $\hat{T}^{comp}_{par}$ is comprised out of the running times for steps 10 and 15 and can be estimated as

$$\hat{T}^{comp}_{par} \leq 2d|D|.$$

5. On the other hand, the analysis of sequential algorithm Quick-DP can give us the following estimation of its computation time $\dot{T}_{seq}$:[2]

$$c_1 \ |D||\Sigma|d \log^{d-2} n \leq \dot{T}_{seq} \leq c_2|D||\Sigma|d \log^{d-2} n, c_2>c_1>1.$$

6. Therefore, the ratio of computation times for Quick-DPPAR and Quick-DP is

$$\frac{\dot{T}^{comp}_{par}}{\dot{T}_{seq}} \leq \frac{1}{N_p}\left(1 + \frac{2N_p}{|\Sigma| \ \log^{d-2} n}\right).$$

To estimate the term $\frac{2N_p}{|\Sigma| \ \log^{d-2} n}$ in the theorem, we consider a case of $d = 8$, $|\Sigma| = 4$, and $n = 512$. $|\Sigma| \ \log^{d-2} n \approx 2 \times 10^6$, which is far greater than 122,400, the total number of computing cores in today's fastest supercomputer, IBM Roadrunner [46]. Therefore, it is practical to assume that $N_p \ll |\Sigma| \ \log^{d-2} n$, i.e., $\frac{2N_p}{|\Sigma| \ \log^{d-2} n} \approx 0$. □

From this analysis, it is not difficult to see that the total amount of work that is not parallelized and the work that is done during the communication steps is significantly smaller, compared to the amount of the parallelized work. From Theorem 3, an estimation of the running time of the parallel algorithm that reflects the algorithm's efficiency is as follows:

Corollary 1. *If n is great enough, the running time $\dot{T}_{Par}$ of the parallel algorithm is*

$$\dot{T}_{Par} = \frac{1}{N_p}(1 + \alpha(n))\dot{T}_{seq}, \quad \text{where} \quad \lim_{n\to\infty} \alpha(n) = 0.$$

We note that the number of dominant points directly affects the efficiency of Quick-DPPAR. Table 2 and Fig. 5 show the total number of dominant points of multiple random DNA sequences of length 100. It indicates that the number of dominant points grows exponentially as the number of sequences increases. This result, on the one hand,

1. We remind that in our estimation, we assume $nN_p \leq |D^k|$.
2. Remind that we assume $|\Sigma| \leq n \leq |D|$.

TABLE 2
The Total Number of Dominant Points and the Number of Positions in the Corresponding Score Matrix L of Multiple Random DNA Sequences of Length 100

Number of sequences	Total number of dominant points	Number of positions in the score matrix L
3	3,432	10^6
5	26,931	10^{10}
7	438,764	10^{14}
9	4,154,064	10^{18}
11	32,964,082	10^{22}

shows the reasonableness of our assumption that $|\Sigma| \leq n \leq |D|$ and $nN_p \leq |D^k|$ in Theorem 3. On the other hand, it reveals that the capability of our method Quick-DPPAR is limited by computer memory to store dominant points.

Fig. 5 also shows that the size of the set of all dominants, due to its nature, is significantly smaller than the number of positions in L. This is the great advantage of this approach, in contrast to classical dynamic programming approaches. The former has the capability to solve biological problems as illustrated by our experimental results in the next section, while the latter are beyond the scope of biological applications.

5 Experimental Results

In our experiments, the algorithms were run on RedHat Enterprise Linux Server release 5.3 (Tikanga) Kernel with 8 Intel Xeon(R) CPUs (2.826 GHz) and 16 GB memory. The programming environment is GNU C++. The algorithms were tested on a set of strings of lengths ranging between 100 and 4,000, over alphabets of size 4 (e.g., nucleotide sequences) and 20 (e.g., protein sequences).

5.1 Sequential Algorithm Quick-DP

The sequential algorithm Quick-DP is compared with Hakata and Imai's A and C algorithms [22]. The A algorithm is specifically designed for three strings and is among the fastest algorithms for three-sequence MLCS problems. The C algorithm can work with any number of

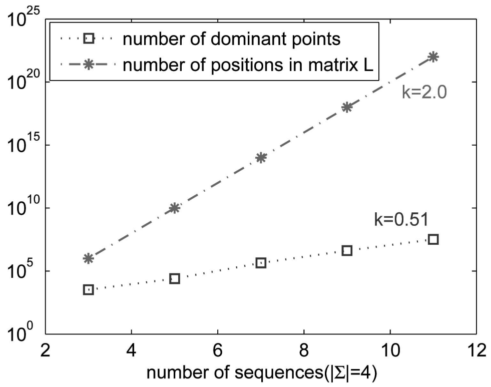

Fig. 5. The total number of dominant points and the number of positions in the corresponding score matrix L of multiple random DNA sequences of length 100; k_1 and k_2 in the figure are the slopes (between the number of sequences and the logarithm of the number of points) calculated for the best fitted lines; the data are taken from Table 6.

TABLE 3
Comparison of the Running Time (in Seconds) of Our Implementation of Hakata and Imai's A Algorithm to Hakata and Imai's Implementation on Three Random DNA Sequences

Sequence length	Hakata and Imai[3]	Our implementation	Speedup[4]
100	0.6	0.0	N/A
200	4.2	0.1	42
300	13.0	0.3	43.3
400	33.7	0.8	42.1
500	62.4	1.3	48
600	107.2	2.4	44.7
700	174.2	3.6	48.4

1. Hakata and Imai's result was taken from their paper [22].
2. speedup is the ratio of the running time of Hakata and Imai's implementation of A algorithm to the running time of our implementation of A algorithm.

sequences. We implemented both Hakata and Imai's A algorithm and C algorithm according to their paper.

Since the MLCS method can be applied to many areas of bioinformatics and computational genomics as well outside the biological domain, the sequence representations of the objects from each application domain as well as the distributions of the letters in the sequences can be drastically different. Therefore, to make an unbiased assessment of our algorithms, we first used as test set a set of strings randomly and independently generated from the alphabet.

Table 3 and Fig. 6 compared our implementation of Hakata and Imai's A algorithm to Hakata and Imai's own implementation of A algorithm, using test sets consisting of three random DNA sequences of various lengths, as described in [22]. The running times of Hakata and Imai's implementation of A algorithm were taken directly from their published paper [22]. From Table 3 and Fig. 6, we can see that our implementation of Hakata and Imai's A algorithm is comparable to Hakata and Imai's own implementation. The speedup of our implementation over Hakata and Imai's implementation is consistent and is a result of hardware improvement.

In the experiments on three-sequence MLCS problems, we generated 10 sets of three random strings for each string length. Quick-DP and Hakata and Imai's A and C algorithms were tested on the same data sets and their

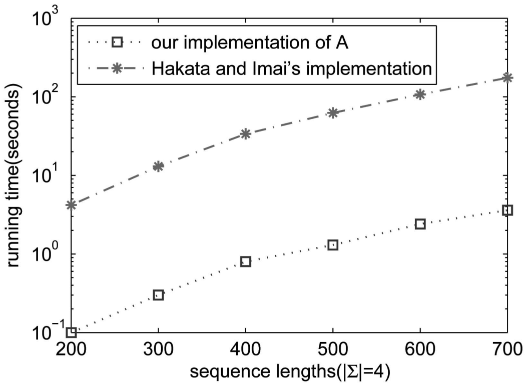

Fig. 6. Comparison of the running time of our implementation of Hakata and Imai's A algorithm to Hakata and Imai's implementation on three random DNA sequences.

TABLE 4
The Average Running Time (in Seconds) of Quick-DP and Hakata and Imai's A and C Algorithms for Random Three-Sequence MLCS Problems of Different Lengths

Sequence	$\|\Sigma\| = 4$			$\|\Sigma\| = 20$		
length	Quick-DP	A	C	Quick-DP	A	C
100	0	0	0	0	0	0
200	0	0.1	0.7	0	0.1	0.1
300	0.3	0.3	1.5	0.1	0.2	0.2
400	0.7	0.8	6.3	0.4	0.6	0.6
500	1.1	1.3	18.5	0.8	1.2	1.3
600	2	2.4	44	1.3	2.2	3.2
700	3.1	3.6	94	2.1	3.6	6.8
800	4.6	5.3	174	3.3	5.4	12
900	7	7.8	315	4.7	7.8	20.6
1000	9.7	10.4	502	6.7	10.7	33.8

average running times for each length of sequence are shown in Table 4 and Fig. 7.

Fig. 7 shows that Quick-DP is slightly faster than Hakata and Imai's A algorithm on three strings, even though A was designed specially for the case of three strings, while Quick-DP is a general algorithm designed to work with any number of sequences. Hakata and Imai's C algorithm for a general MLCS problem is significantly slower and has much worse computational complexity.

In the benchmark tests of the MLCS algorithms for more than three sequences, we compared Quick-DP with Hakata and Imai's C algorithm and another MLCS algorithm FAST-LCS [11]. FAST-LCS is designed specifically for MLCS problems of alphabet size 4. Table 5 and Fig. 8 show the result. The results show that Quick-DP is much faster than FAST-LCS and the C algorithm, achieving several orders of magnitude higher speed on long sequences. For instance, on the $\Sigma = 4$ test cases, Quick-DP is over 10^4 times faster than FAST-LCS and about 400 times faster than the C algorithm on problems of length 200.

5.2 Parallel Algorithm Quick-DP$_{PAR}$

The parallel algorithm Quick-DPPAR was implemented using multithreading in GCC and compiled using the command line option "g++ -pthread." The reason for choosing multithreading is that it provides fine-grained computation and efficient performance.

Following the parallelization scheme in Section 4, our implementation consists of one master thread and N_p slave threads. The master thread distributes a set of dominant points D^k evenly among slaves to calculate the parents and the corresponding minima. After all slave threads finish calculating their subsets of parents, they copy these subsets back to the memory of the master thread. Then, the master thread assigns each slave to find the minimal elements of s-parents, $s \in \Sigma$. The set of minima is then assigned to be the $(k+1)$st dominant set. The master thread reiterates this process until an empty parent set is obtained.

We first evaluated the speedup of our parallel algorithm Quick-DPPAR over sequential algorithm Quick-DP. Speedup is defined here as the ratio of the execution time of the sequential algorithm over that one of the parallel algorithm. Similar to previous experiments, we generated 10 sets of five random sequences for alphabet size 4 and 20, respectively. We ran Quick-DPPAR using different number of slave threads. Table 6 and Fig. 9 show the results for sequence

TABLE 5
The Average Running Time (in Seconds) of Quick-DP, Hakata and Imai's C Algorithm, and FAST-LCS (FAST-LCS Code Only Works on $\Sigma = 4$ Problems) on MLCS Problems of Five Random Sequences of Different Lengths

(a)$|\Sigma| = 4$

Sequence length	Quick-DP	C	FAST-LCS
100	0.2	3.6	46.8
120	0.6	15.8	266.9
140	0.9	54.9	1,430.5
160	1.4	149.9	4,801.3
180	2.2	426.0	17,143.7
200	2.6	996.4	40,262.5

(b)$|\Sigma| = 20$

Sequence length	Quick-DP	C	FAST-LCS
100	0.0	1.7	N/A
120	0.1	6.6	N/A
140	0.4	26.0	N/A
160	0.5	71.5	N/A
180	0.8	203.3	N/A
200	1.1	560.0	N/A

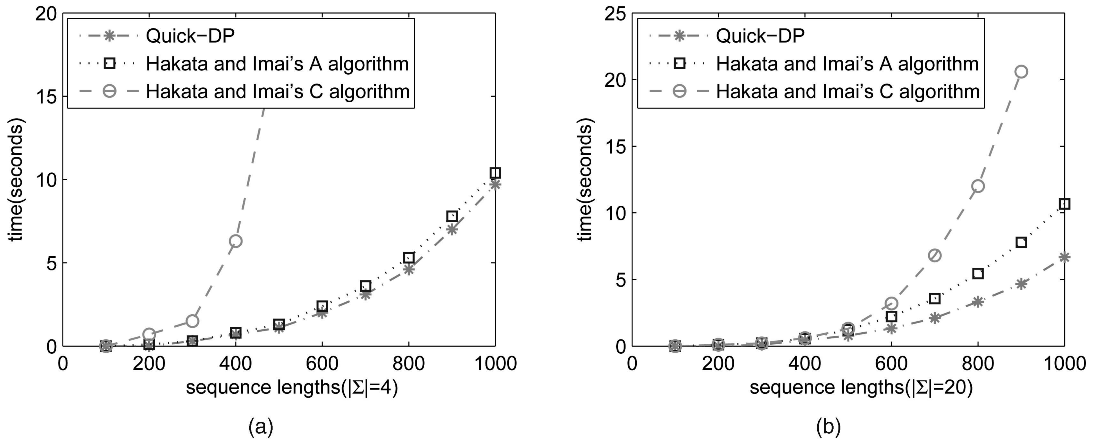

Fig. 7. The average running time (in seconds) of Quick-DP and Hakata and Imai's A and C algorithms for random three-sequence MLCS problems of different lengths. The data are taken from Table 4.

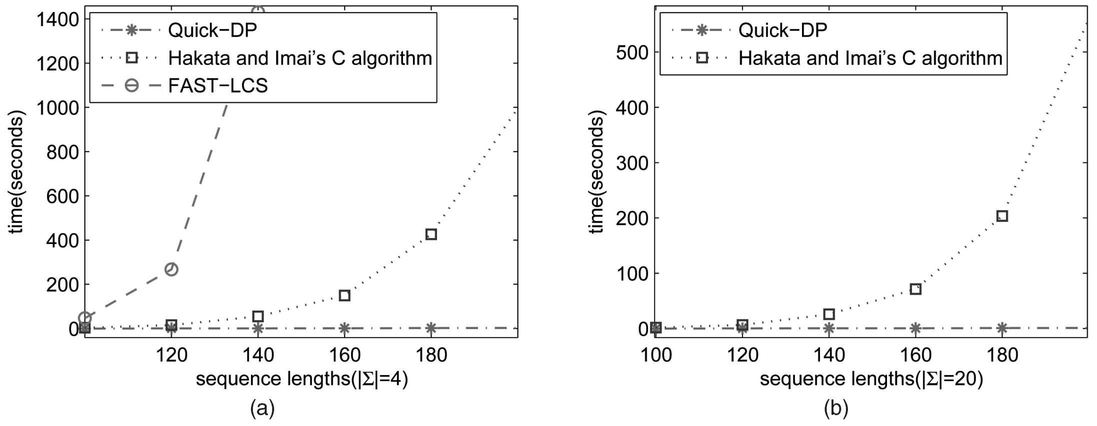

Fig. 8. The average running time (in seconds) of Quick-DP, Hakata and Imai's C algorithm, and FAST-LCS on MLCS problems of five random strings of different lengths. The data are taken from Table 5.

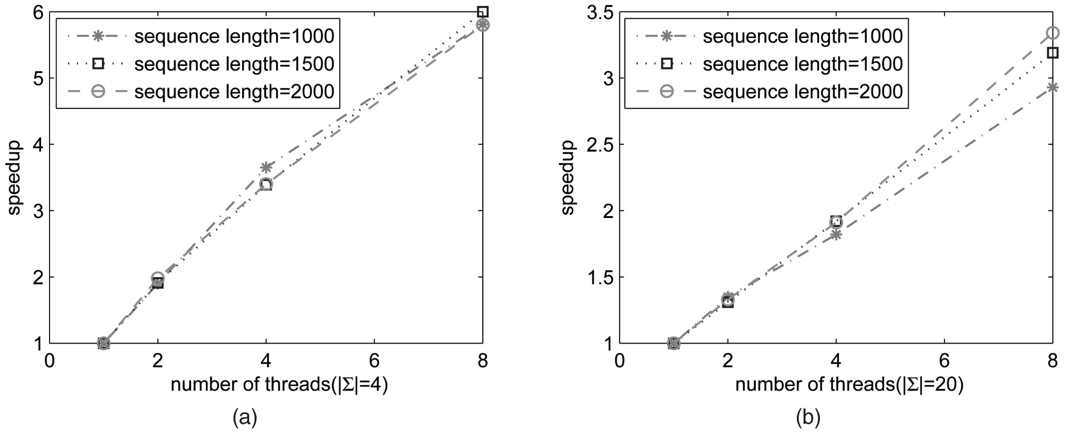

Fig. 9. The speedup of our parallel method Quick-DPPAR over sequential algorithm Quick-DP on MLCS problems of five random strings. The data are taken from Table 6.

lengths 1,000, 1,500, and 2,000. Near-linear speedups were achieved for all these cases.

Then, we measured the running time of Quick-DPPAR on strings of various lengths. We ran Quick-DPPAR using eight slave threads on test sets of five random DNA sequences. The results in Fig. 10 demonstrate the efficiency of Quick-DPPAR.

Next, Quick-DPPAR was compared with parMLCS [29], a parallel version of Hakata and Imai's C algorithm [22], on multiple random sequences. Fig. 11 shows the computation times of both algorithms using eight slave threads. Here, the length of sequences was fixed at 100 and the number of sequences was changed in each test case. The result shows that Quick-DPPAR is significantly more efficient than parMLCS.

TABLE 6
Speedup of the Parallel Method Quick-DPPAR over Sequential Algorithm Quick-DP on MLCS Problems of Five Random Strings

(a)$|\Sigma| = 4$

Sequence length	1 CPU	2 CPU	4 CPU	8 CPU
1000	1.00	1.90	3.65	5.81
1500	1.00	1.91	3.39	6.00
2000	1.00	1.98	3.40	5.80

(b)$|\Sigma| = 20$

Sequence length	1 CPU	2 CPU	4 CPU	8 CPU
1000	1.00	1.35	1.82	2.93
1500	1.00	1.31	1.92	3.19
2000	1.00	1.33	1.91	3.34

We also tested our algorithms on real biological sequences by applying our algorithms to find MLCS of various number of protein sequences from the family of melanin-concentrating hormone receptors (MCHRs), proteins that are linked to the regulation of energy balance and body weight [40]. The lengths of the protein sequences range from 296 to 423 amino acids. The results in Fig. 12 demonstrate that Quick-DPPAR (ran on eight processors) is more efficient than parMLCS (on eight processors) and the sequential algorithm Quick-DP on real protein sequences.

Finally, we compared Quick-DPPAR with current multiple sequence alignment programs used in practice, ClustalW (version 2) [31] and MUSCLE (version 4) [14], [15]. Both ClustalW and MUSCLE were run on the same Linux Server as Quick-DPPAR. Two basic command line options, "-input" and "-log," were used for the execution of MUSCLE.

As test data, we chose eight protein domain families from the Pfam database [17], [18], a collection of protein families that includes their annotations and multiple sequence

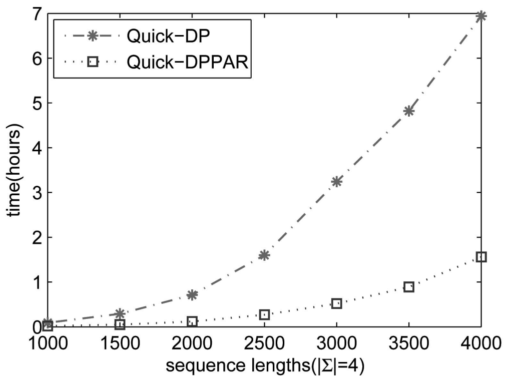

Fig. 10. The average running times (in hours) of our parallel Quick-DPPAR (on eight processors) and sequential Quick-DP algorithms on MLCS problems of five random sequences.

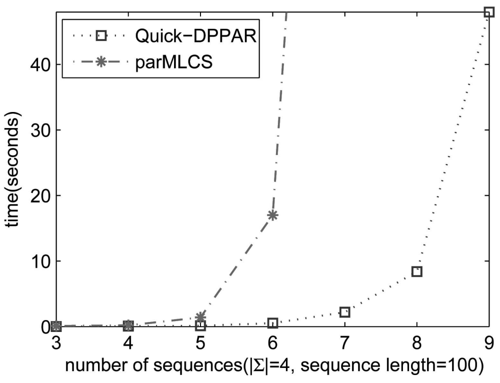

Fig. 11. The average running time of the parallel methods Quick-DPPAR and parMLCS on multiple random sequences of length 100.

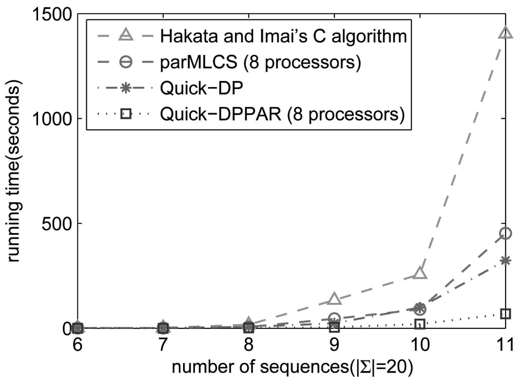

Fig. 12. The running time of the parallel methods Quick-DPPAR (on eight processors) and parMLCS (on eight processors) and the sequential methods Quick-DP and Hakata and Imai's C algorithm on multiple protein sequences from the family of MCHRs.

alignments. Since neither Quick-DPPAR nor ClustalW and MUSCLE are able to handle within reasonable time protein families as large as AP_endonuc_2 (accession number: PF01261) that consists of 2,966 protein sequences, we selected eight sequences of roughly the same length, i.e., around 200 amino acids, from each family. The protein families as well as the selected sequences are provided in Table 7, sorted in increasing order of the last column, average pairwise sequence identities, which were computed using MUSCLE.

Because MLCS provides the optimal solution and an alignment does not, it is very difficult to compare the computation time of Quick-DPPAR with those ones of ClustalW and MUSCLE. For the protein families in Table 7, it took Quick-DPPAR 8.1 seconds, on average, to compute the longest common subsequences for a family, while it

TABLE 7
Protein Domain Families and Sequences Selected from the Families for the Comparison of Quick-DPPAR with ClustalW and MUSCLE

Family ID (accession number)	Sequences	Average length of sequences	Pairwise sequence identity
AP_endonuc_2 (PF01261)	A0AMF1_LISW6/19-219, A0B837_METTP/20-236, A0AF79_LISW6/85-300, A0JS78_ARTS2/26-224, A0LNH5_SYNFM/22-217, A0K062_ARTS2/47-259, A0JT54_ARTS2/20-238, A0NI89_OENOE/24-207	205	19.6%
DUF2077 (PF09850)	A0GJN6_9BURK/66-274, A0GQV3_9BURK/29-231, A0VH27_COMAC/49-254, A0J3W8_9GAMM/40-245, A0KJC3_AERHH/15-220, A0Y0Q2_9GAMM/8-212, A0NQ34_9RHOB/140-345,A0Q3Z7_FRATN/3-202	205	25.2%
NikM (PF10670)	A0RME2_CAMFF/19-227, A1IET5_9DELT/22-232, A7DA90_METEX/17-217, A3PRQ8_RHOS1/21-229, A5NWV1_9RHIZ/24-234, A7H0G9_9PROT/15-223, A5VRB3_BRUO2/18-226, A6X6M1_OCHA4/22-219	207	35.7%
Nop25 (PF09805)	A1CDA9_ASPCL/6-213, A1DC87_NEOFI/6-215, Q75EF5_ASHGO/66-261, Q0CPT2_ASPTN/6-208, Q1DQ51_COCIM/6-209, , RRP17_YEAST/17-235, Q2UT94_ASPOR/6-210, Q6CY97_KLULA/17-243	211	45.9%
Exon_PolB (PF10108)	A1ISK6_NEIMA/47-263, A1WT12_HALHL/45-254, A4SYE6_9BURK/47-269, Q0A8Y5_ALHEH/46-256, Q0K954_RALEH/47-265, Q0VP25_ALCBS/45-254, Q2L012_BORA1/47-256, Q5WWT9_LEGPL/45-255	213	56.5%
Frag1 (PF10277)	A3M0G2_PICST/28-238,A5DAL2_PICGU/21-231,A5DS39_LODEL/32-242, A6ZTJ7_YEAS7/5-207, A7TRI4_VANPO/5-205,Q6CX54_KLULA/5-205, Q6FKX5_CANGA/5-205, Q757Q5_ASHGO/5-206	205	65.1%
G6PD_bact (PF10786)	A1ISU8_NEIMA/1-204,A1KUW8_NEIMF/1-204,A3MYB3_ACTP2/1-202, A4N6J4_HAEIN/3-206,A4NBD8_HAEIN/3-206,A4NGP0_HAEIN/3-206, A4NMH6_HAEIN/3-206,A4NQJ0_HAEIN/3-206	203	74.9%
Adeno_hexon_C (PF03678)	A0MK56_9ADEN/603-830,A2I909_ADE03/1-180, A2I915_ADE03/1-180, A2TJK9_9ADEN/588-815,A4ZKK1_9ADEN/641-867,A4ZKL6_9ADEN/646-872, A4ZZ78_ADEC2/588-815,A5JQJ4_9ADEN/588-815	215	85.2%

TABLE 8
The Lengths of Common Subsequences Extracted by ClustalW, MUSCLE, and Quick-DPPAR from the Families of Sequences Described in Table 7

Family ID	ClustalW	MUSCLE	Quick-DPPAR
AP_endonuc_2	0	0	30
DUF2077	7	7	35
NikM	10	10	40
Nop25	30	30	67
Exon_PolB	63	63	76
Frag1	87	87	93
G6PD_bact	100	100	105
Adeno_hexon_C	133	133	136

took MUSCLE only 0.8 seconds to align sequences of a family. However, the big advantage of Quick-DPPAR over ClustalW and MUSCLE is that Quick-DPPAR guarantees to find optimal solution. Table 8 shows the length of the common subsequences extracted by the three methods from the given sequences (the common subsequences were retrieved from alignment by counting the number of residues that are in common among all sequences in the alignment). It indicates that the lengths of the common subsequences computed by ClustalW are the same as those produced by MUSCLE on the given test data. It also demonstrates that the common subsequences calculated by Quick-DPPAR are consistently longer than those extracted by ClustalW and MUSCLE. More importantly, in the cases such as protein family AP_endonuc_2 when the pairwise identity among sequences is poor, Quick-DPPAR still worked well while both ClustalW and MUSCLE failed to find residues that are in common among all sequences.

6 Summary

The first main contribution of this paper is the design of a new efficient algorithm, Quick-DP, for solving a general case of MLCS problem. The comparison with the best current methods using a comprehensive benchmark test allows one to suggest that Quick-DP is currently the fastest sequential general MLCS algorithm, whose speed is orders of magnitude higher than the existing methods. When ran on real protein sequences, Quick-DP is even faster than the parallel algorithm parMLCS ran on eight processors. Our second contribution of this paper is to design an efficient parallelization of Quick-DP. Our theoretic analysis of the parallel algorithm predicted that speedup is asymptotically linear. The comprehensive benchmark assessment fully confirmed this prediction. In addition, the results of comparative benchmarking of our parallel implementation and the top current parallel approaches allow to suggest that our algorithm is currently the fastest among any other parallel implementations of a general MLCS algorithm.

Analysis of protein and genome sequences is one of the principle application areas for the MLCS methods [11], [12], [13], [29]. Therefore, the directions of further improvement and development of our algorithms in this area will be guided by the needs of the bioinformatics and computational genomics community. An important bioinformatics problem is finding a protein motif in a family of homologous proteins. Homologous proteins usually share sequence identity of 30 and higher [43]. Often, an identical substring among the homologous protein sequences corresponds to a functionally important, and thus, evolutionary conserved region of a protein. The set of all such conserved regions for a group of proteins or protein domains constitutes a protein motif. While a simple sequence motif usually belongs to one region of a protein sequence, a protein motif often corresponds to a structurally common feature, which may not necessarily be localized sequentially. Applying the MLCS problem thus allows to determine such type of protein motif together with the positions of the corresponding conserved regions on the protein sequences. An average protein domain contains 100-200 residues [47]; thus, our method has to deal with the sequences of 100-200 letters, on average. Even in its current implementation, the method is readily applicable to detecting protein motifs of a family of more than 10 proteins on a small server of 8 cpus. Our next steps will be toward improving the method's efficiency to handle larger protein families. In particular, we will aim to limit the algorithm's search to a small subset of the dominant set.

Appendix A

Pseudocode of the Dominant Point Approach

Pseudocode of a simplified dominant point algorithm is as follows:

```
 Algorithm MCLS ($\{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_d\}$, $\Sigma$)
 Calculation of dominant points
01 Preprocessing; $D^0 = \{[-1, -1, \ldots, -1]\}$;  $k = 0$;
02 while $D^k$ not empty do {$A = \emptyset$
03   for p $\in D^k$ do {
04     Par(p, $\Sigma$) = Parents(p);
05     A = A $\cup$ Par(p, $\Sigma$) ; }
06   $D^{k+1}$ = Minima (A)
07   $k = k + 1$; }
 Calculation of MLCS-optimal path
08 pick a point $p = [p_1, p_2, \ldots, p_d] \in D^{k-1}$;
09 while $k - 1 > 0$ do {
10   current LCS position = $\mathbf{a}_1[p_1]$;
11   pick a point q such that p $\in$ Par(q, $\Sigma$) ;
12   p = q;
13   k = k - 1; }
```

The algorithm consists of two main parts. In the first part, the set of all dominants is calculated iteratively, starting from a 0-dominant set (containing one element), where the set of $(k+1)$-dominants $D^{(k+1)}$ is obtained, based on the set of k-dominants D^k as follows: We first calculate the set of all parents of D^k. Then, from this set, we choose a subset of minimal elements, which will be exactly the set of all $(k+1)$-dominants. In the second part, a path corresponding to an MLCS is calculated by tracing back through set of dominant points obtained in the first part of the algorithm and starting with an element from the last dominant set.

Appendix B

Pseudocode of the Divide-and-Conquer Method

The following function $Minima(A)$ computes the minima for a multidimensional vector A using divide-and-conquer approach. The algorithm is based on the ideas presented in Section 3.

The function $Minima()$ initiates the recursive process. In the function $Divide()$, the point set A is equally partitioned into two sets Q and R with respect to the dth-dimensional coordinate so that the dth-dimensional coordinates of points in R are all greater than those of points in Q. Then, the minima of each partitioned subset are recursively computed. The points in the minima of R that are dominated by points in the minima of Q are removed by the function $Union()$. The union of the remaining points in the minima of R and the minima of Q is the output.

Appendix C

Parallelization of the Divide-and-Conquer Algorithm

The following function $MinimaPAR()$ is a parallel version of $Minima()$ (see Appendix B). $MinimaPAR()$ is based upon two observations on $Minima()$: 1) The sets Q' and R', which are returned by function $Divide()$ in lines 10 and 11, can be calculated independently; and 2) The sets Q' and R' in lines 21-22 in function $Union()$ are also independent from each. In recursive functions $Divide()$ and $Union()$, the computation of Q' and R' is much more expensive computationally than other linear time operations such as $Partition()$(lines 9 and 20) and two-dimensional minimization (lines 7 and 18). Therefore, two processors $Proc_q$ and $Proc_r$ are allocated to calculate Q' and R', respectively, so as to reach good speedup.

```
 Function Minima (A)
 Compute the minima of A
01 B = QuickSort(A);
02 A' = Divide(B, d);
03 return A';

 Function Divide(A, d)
 Termination condition
04 if(size(A) == 1){
05   return A;
06 }else if(d == 2){
07   remove non − minima from A;
08   return A; }
 Recursively minimize A
09 [Q, R] = Partition(A);
10 Q' = Divide(Q, d);
11 R' = Divide(R, d);
12 Label q ∈ Q' black and r ∈ R' white;
13 A' = Union(Q' ∪ R', d − 1);
14 return A';

 Function Union(A, d)
 Termination condition
15 if(size(A) == 1){
16   return A;
17 }else if(d == 2){
18   remove white points dominated by blacks from A;
19   return A; }
 Recursively minimize A
20 [Q, R] = Partition(A);
21 Q' = Union(Q, d);
22 R' = Union(R, d);
23 B = black points in Q' ∪ white points in R';
24 A' = Union(B, d − 1);
25 return Q' ∪ {black points in R'} ∪ A';

 Function MinimaPAR (A)
 Proc_0:
01 B = QuickSort(A);
02 A' = DividePAR(B, d);
03 return A';

 Function DividePAR(A, d)
 Proc_0:
04 if(size(A) == 1){
05   return A;
06 }else if(d == 2){
07   remove non − minima from A;
08   return A; }
09 [Q, R] = Partition(A);
10 if(exist(Proc_q and Proc_r)){
11 Proc_q: Q'= DividePAR(Q, d);
12 Proc_r: R'= DividePAR(R, d);
13 }else{
14   Q' = Divide(Q, d);
15   R' = Divide(R, d);}
 Proc_0:
16 Label q ∈ Q' black and r ∈ R' white;
17 A' = UnionPAR(Q' ∪ R', d − 1);
18 return A';

 Function UnionPAR(A, d)
 Proc_0:
19 if(size(A) == 1){
20   return A;
21 }else if(d == 2){
22   remove white points dominated by blacks from A;
23   return A; }
24 [Q, R] = Partition(A);
25 if(exist(Proc_q and Proc_r)){
26 Proc_q: Q' = UnionPAR(Q, d);
27 Proc_r: R' = UnionPAR(R, d);
28 }else{
29   Q' = Union(Q, d);
30   R' = Union(R, d); }
 Proc_0:
31 B = black points in Q' ∪ white points in R';
32 A'= UnionPAR(B, d − 1);
33 return Q' ∪ {black points in R'} ∪ A';
```

Acknowledgments

This research was supported in part by the Shumaker Endowment in Bioinformatics and NIH Grant 5R21GM078601-02. D. Korkin acknowledges funding from the US National Science Foundation (NSF) grant DBI-0845196. The authors are also grateful to the authors of FAST-LCS for sharing with them its code.

REFERENCES

[1] A. Apostolico, M. Atallah, L. Larmore, and S. Mcfaddin, "Efficient Parallel Algorithms for String Editing and Related Problems," *SIAM J. Computing,* vol. 19, pp. 968-988, 1990.

[2] 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.

[3] T.K. Attwood and J.B.C. Findlay, "Fingerprinting G Protein-Coupled Receptors," *Protein Eng.,* vol. 7, no. 2, pp. 195-203, 1994.

[4] K.N. Babu and S. Saxena, "Parallel Algorithms for the Longest Common Subsequence Problem," *Proc. Fourth Int'l Conf. High Performance Computing,* pp. 120-125, 1997.

[5] L.J. Bentley, "Multidimensional Divide-and-Conquer," *Comm. ACM,* vol. 23, no. 4, pp. 214-229, 1980.

[6] L. Bergroth, H. Hakonen, and T. Raita, "A Survey of Longest Common Subsequence Algorithms," *Proc. Int'l Symp. String Processing Information Retrieval (SPIRE '00),* pp. 39-48, 2000.

[7] M. Blanchette, T. Kunisawa, and D. Sankoff, "Gene Order Breakpoint Evidence in Animal Mitochondrial Phylogeny," *J. Molecular Evolution,* vol. 49, no. 2, pp. 193-203, 1999.

[8] P. Bork and E.V. Koonin, "Protein Sequence Motifs," *Current Opinion in Structural Biology,* vol. 6, pp. 366-376, 1996.

[9] G. Bourque and P.A. Pevzner, "Genome-Scale Evolution: Reconstructing Gene Orders in the Ancestral Species," *Genome Research,* vol. 12, pp. 26-36, 2002.

[10] L. Brocchieri and S. Karlin, "Protein Length in Eukaryotic and Prokaryotic Proteomes," *Nucleic Acids Research,* vol. 33, no. 10, pp. 3390-3400, 2005.

[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, p. S4, 2006.

[12] F.Y. Chin and C.K. Poon, "A Fast Algorithm for Computing Longest Common Subsequences of Small Alphabet Size," *J. Information Processing,* vol. 13, no. 4, pp. 463-469, 1990.

[13] M.O. Dayhoff, "Computer Analysis of Protein Evolution," *Scientific Am.,* vol. 221, no. 1, pp. 86-95, 1969.

[14] R.C. Edgar, "MUSCLE: Multiple Sequence Alignment with High Accuracy and High Throughput," *Nucleic Acids Research,* vol. 32, no. 5, pp. 1792-1797, 2004.

[15] R.C. Edgar, "MUSCLE: A Multiple Sequence Alignment Method with Reduced Time and Space Complexity," *BMC Bioinformatics,* vol. 5, no. 1, p. 113, 2004.

[16] S.M. Elbashir, J. Harborth, W. Lendeckel, A. Yalcin, K. Weber, and T. Tuschl, "Duplexes of 21-Nucleotide RNAs Mediate RNA Interference in Cultured Mammalian Cells," *Nature,* vol. 411, no. 6836, pp. 494-498, 2001.

[17] R.D. Finn, J. Tate, J. Mistry, P.C. Coggill, J.S. Sammut, H.R. Hotz, G. Ceric, K. Forslund, S.R. Eddy, E.L. Sonnhammer, and A. Bateman, "The Pfam Protein Families Database," *Nucleic Acids Research,* vol. 36, pp. D281-D288, 2008.

[18] R.D. Finn, J. Mistry, B. Schuster-Böckler, S. Griffiths-Jones, V. Hollich, T. Lassmann, S. Moxon, M. Marshall, A. Khanna, R. Durbin, S.R. Eddy, E.L.L. Sonnhammer, and A. Bateman, "Pfam: Clans, Web Tools and Services," *Nucleic Acids Research,* vol. 34, pp. D247-D251, 2006.

[19] V. Freschi and A. Bogliolo, "Longest Common Subsequence between Run-Length-Encoded Strings: A New Algorithm with Improved Parallelism," *Information Processing Letters,* vol. 90, no. 4, pp. 167-173, 2004.

[20] T.R. Gregory, Animal Genome Size Database, http://www.genomesize.com, 2005.

[21] K. Hakata and H. Imai, "Algorithms for the Longest Common Subsequence Problem," *Proc. Genome Informatics Workshop III,* pp. 53-56, 1992.

[22] K. Hakata and H. Imai, "Algorithms for the Longest Common Subsequence Problem for Multiple Strings Based on Geometric Maxima," *Optimization Methods and Software,* vol. 10, pp. 233-260, 1998.

[23] K.F. Han and D. Baker, "Recurring Local Sequence Motifs in Proteins," *J. Molecular Biology,* vol. 251, no. 1, pp. 176-187, 1995.

[24] D.S. Hirschberg, "Algorithms for the Longest Common Subsequence Problem," *J. ACM,* vol. 24, pp. 664-675, 1977.

[25] W.J. Hsu and M.W. Du, "Computing a Longest Common Subsequence for a Set of Strings," *BIT Numerical Math.,* vol. 24, no. 1, pp. 45-59, 1984.

[26] J.W. Hunt and T.G. Szymanski, "A Fast Algorithm for Computing Longest Common Subsequences," *Comm. ACM,* vol. 20, no. 5, pp. 350-353, 1977.

[27] D. Korkin, "A New Dominant Point-Based Parallel Algorithm for Multiple Longest Common Subsequence Problem," Technical Report TR01-148, Univ. of New Brunswick, 2001.

[28] D. Korkin and L. Goldfarb, "Multiple Genome Rearrangement: A General Approach via the Evolutionary Genome Graph," *Bioinformatics,* vol. 18, pp. S303-S311, 2002.

[29] D. Korkin, Q. Wang, and Y. Shang, "An Efficient Parallel Algorithm for the Multiple Longest Common Subsequence (MLCS) Problem," *Proc. 37th Int'l Conf. Parallel Processing (ICPP '08),* pp. 354-363, 2008.

[30] H.T. Kung, F. Luccio, and F.P. Preparata, "On Finding the Maxima of a Set of Vectors," *J. ACM,* vol. 22, pp. 469-476, 1975.

[31] M.A. Larkin, G. Blackshields, N.P. Brown, R. Chenna, P.A. McGettigan, H. McWilliam, F. Valentin, I.M. Wallace, A. Wilm, R. Lopez, J.D. Thompson, T.J. Gibson, and D.G. Higgins, "Clustal W and Clustal X Version 2.0," *Bioinformatics,* vol. 23, pp. 2947-2948, 2007.

[32] H.F. Lodish, *Molecular Cell Biology.* WH Freeman, 2003.

[33] M. Lu and H. Lin, "Parallel Algorithms for the Longest Common Subsequence Problem," *IEEE Trans. Parallel and Distributed System,* vol. 5, no. 8, pp. 835-848, Aug. 1994.

[34] G. Luce and J.F. Myoupo, "Systolic-Based Parallel Architecture for the Longest Common Subsequences Problem," *VLSI J. Integration,* vol. 25, pp. 53-70, 1998.

[35] D. Maier, "The Complexity of Some Problems on Subsequences and Supersequences," *J. ACM,* vol. 25, pp. 322-336, 1978.

[36] W.J. Masek and M.S. Paterson, "A Faster Algorithm Computing String Edit Distances," *J. Computer and System Sciences,* vol. 20, pp. 18-31, 1980.

[37] J.F. Myoupo and D. Seme, "Time-Efficient Parallel Algorithms for the Longest Common Subsequence and Related Problems," *J. Parallel and Distributed Computing,* vol. 57, pp. 212-223, 1999.

[38] A. Nekrutenko and W.H. Li, "Transposable Elements Are Found in a Large Number of Human Protein-Coding Genes," *Trends in Genetics,* vol. 17, no. 11, pp. 619-621, 2001.

[39] C. Rick, "New Algorithms for the Longest Common Subsequence Problem," Technical Report No. 85123-CS, Computer Science Dept., Univ. of Bonn, Oct. 1994.

[40] Y. Saito, H.-P. Nothacker, Z. Wang, S.H.S. Lin, F. Leslie, and O. Civelli, "Molecular Characterization of the Melanin-Concentrating-Hormone Receptor," *Nature,* vol. 400, pp. 265-269, 1999.

[41] D. Sankoff, "Matching Sequences Under Deletion/Insertion Constraints," *Proc. Nat'l Academy of Sciences USA,* vol. 69, pp. 4-6, 1972.

[42] D. Sankoff and M. Blanchette, "Phylogenetic Invariants for Genome Rearrangements," *J. Computational Biology,* vol. 6, pp. 431-445, 1999.

[43] D. Sankhoff and J.B. Kruskal, *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wealey, 1983.

[44] R.P. Sheridan and R. Venkataraghavan, "A Systematic Search for Protein Signature Sequences," *Proteins,* vol. 14, no. 1, pp. 16-28, 1992.

[45] T.F. Smith and M.S. Waterman, "Identification of Common Molecular Subsequences," *J. Molecular Biology,* vol. 147, pp. 195-197, 1981.

[46] The Los Alamos National Laboratory Website, http://www.lanl.gov/roadrunner/index.shtml, 2009.

[47] E.N. Trifonov and I.N. Berezovsky, "Evolutionary Aspects of Protein Structure and Folding," *Current Opinion in Structural Biology,* vol. 13, no. 1, pp. 110-114, 2003.

[48] R.A. Wagner and M.J. Fischer, "The String to String Correction Problem," *J. ACM,* vol. 21, no. 1, pp. 168-173, 1974.

[49] X. Xu, L. Chen, Y. Pan, and P. He, "Fast Parallel Algorithms for the Longest Common Subsequence Problem Using an Optical Bus," *Lecture Notes in Computer Science,* pp. 338-348, Springer, 2005.

[50] T.K. Yap, O. Frieder, and R.L. Martino, "Parallel Computation in Biological Sequence Analysis," *IEEE Trans. Parallel and Distributed Systems,* vol. 9, no. 3, pp. 283-294, Mar. 1998.

[51] M.S. Zastrow, D.B. Flaherty, G.M. Benian, and K.L. Wilson, "Nuclear Titin Interacts with A-and B-Type Lamins In Vitro and In Vivo," *J. Cell Science,* vol. 119, no. 2, pp. 239-249, 2006.

Qingguo Wang is working on his PhD degree in the Department of Computer Science at the University of Missouri. He received the ME degree in computer technology from Shanghai Jiao Tong University, China, and the BE degree in nuclear technology from Chengdu University of Technology, China. His research interests include algorithm, artificial intelligence, machine learning, and bioinformatics. He is a recipient of the Shumaker Fellowship for 2008 and the Upsilon Pi Epsilon Scholarship Award for 2010.

Dmitry Korkin received the PhD degree from the University of New Brunswick, Canada. He is an assistant professor in the Informatics Institute and Department of Computer Science at the University of Missouri, Columbia. His interests are in structural bioinformatics and computational biology of protein assemblies and host-pathogen interactions as well as machine learning and pattern recognition. Before coming to MU, he was a postdoctoral researcher at the University of California at San Francisco and Rockefeller University. His research is supported by the University of Missouri and National Science Foundation.

Yi Shang received the PhD degree from the University of Illinois at Urbana-Champaign in 1997. He is a professor in the Department of Computer Science at the University of Missouri. He has published more than 100 technical papers in international journals and conferences and has six US patents. He worked as a researcher for the University of Illinois and the Palo Alto Research Center before coming to Missouri. His research interests include wireless sensor networks, bioinformatics, intelligent distributed systems, and nonlinear optimization. His research has been supported by the US National Science Foundation (NSF), NIH, US Defense Advanced Research Projects Agency (DARPA), Microsoft, and Raytheon. He is a senior member of the IEEE and ACM.