Efficient Dominant Point Algorithms for the Multiple Longest Common Subsequence (MLCS) Problem

Qingguo Wang, Dmitry Korkin and Yi Shang
Department of Computer Science
University of Missouri
qwp4b@mizzou.edu, korkin@korkinlab.org, shangy@missouri.edu

Abstract

Finding the longest common subsequence of multiple strings is a classical computer science problem and has many applications in the areas of bioinformatics and computational genomics. In this paper, we present a new sequential algorithm for the general case of MLCS problem, 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 find a MLCS of 3 strings, our general algorithm is shown to exhibit the same performance as the best existing MLCS algorithm by Hakata and Imai, designed specifically for the case of 3 strings. Moreover, we show that for a general case of more than 3 strings, the algorithm is significantly faster than the best existing sequential approaches, reaching up to 2-3 orders of magnitude faster on the large-size problems. Finally, we propose a parallel implementation of the algorithm. Evaluating the parallel algorithm on a benchmark set of both random and biological sequences reveals a near-linear speed-up with respect to the sequential algorithm.

Keywords: Search, Dynamic Programming, Computational Biology

1 Introduction

The multiple longest common subsequence problem (MLCS) is to find the longest subsequence shared between two or more sequences. It is a classical computer science problem with important applications in many fields such as information retrieval and computational biology [Masek and Paterson, 1980; Smith and Waterman, 1981]. For over 30 years, significant efforts have been made to find an efficient algorithm for the MLCS problem. The most significant contribution has been done to study the simplest case of MLCS of two or three sequences [Hirschberg, 1977; Hakata and Imai, 1998]. However, while several attempts towards finding an efficient algorithm for a general case of more than 3 sequences [Hakata and Imai, 1998; Chen *et al.*, 2006], it is yet to be developed. A general case of MLCS is of a tremendous value to computational biology and computational genomics that deal with biological sequences [Korkin *et al.*, 2008]. With the increasing volume of biological data and prevalent usage of computational sequence analysis tools, an efficient MLCS algorithm applicable to many sequences will have a significant impact on computational biology and applications.

In this paper, we present an efficient algorithm for the MLCS problem of three and more sequences. The new method is based on the dominant point approach. Dominant points are minimal points in a multidimensional search space. Knowing those points allows to reduce the search space size by orders of magnitude, hence significantly the computation time. Our algorithm performs a new divide-and-conquer technique to construct dominant point sets efficiently. Unlike FAST-LCS [Chen *et al.*, 2006], a MLCS algorithm that works with the whole dominant point set, our method takes advantages of the structure relationships among dominant points and partitions them into independent subsets, where the divide-and-conquer technique is applied. Compared to existing state-of-the-art MLCS algorithms, our dominant-point algorithm is significantly faster on multiple sequences longer than 1000. We have also developed an efficient parallel version of the algorithm. By dividing the problem into smaller sub-problems and solving the sub-problems in parallel, we have achieved a near linear speedup.

The paper is organized as follows. In the next section, we briefly review state-of-the-art methods for MLCS. Then, we present the basics of the dominant point method in Section 3 and the new sequential algorithm in Section 4. The new parallel algorithm is presented in Section 5. In Section 6, we show the experimental results. Finally, in Section 7, we summarize the paper.

2 Related work

Classical methods for the MLCS problem are based on dynamic programming [Sankoff, 1972; Smith and Waterman, 1981]. In its simplest case, given two sequences a_1 and a_2 of length n_1 and n_2 respectively, a dynamic programming algorithm iteratively builds a $n_1 \times n_2$ score matrix L, in which $L[i, j], 0 \le i \le n_1, 0 \le j \le n_2$ is the length of a LCS between two prefixes $a_1[1, \ldots, i]$ and $a_2[1, \ldots, j]$.

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

In a straightforward implementation of dynamic programming, we calculates all entries in L. The resulting algorithm has time and space complexity of $O(n^d)$ for d sequences of length n. Various approaches have been introduced to reduce the complexity of dynamic programming [Hirschberg, 1977; Masek and Paterson, 1980; Hsu and Du, 1984; Apostolico *et al.*, 1992; Rick, 1994]. Unfortunately, these approaches primarily address the special case of 2 sequences.

In contrast to dynamic programming, the dominant point approach limits its search to exploring a smaller set of dominant points rather than the whole set of positions in L. The initial idea of dominant points as special points in a matrix was introduced by Hirschberg [1977]. The dominant-point approach has been successfully applied to the two-sequences cases [Hirschberg, 1977; Chin and Poon, 1990; Apostolico *et al.*, 1992]. In [Hakata and Imai, 1998], several dominant-point algorithms for more than 2 sequences were proposed. One of the algorithms, called Algorithm A, which was designed specifically for finding a LCS of 3 strings, is overwhelmingly faster than dynamic programming for 3 sequences. However, Algorithm A minimizes dominant point sets by enumerating points of the same coordinate values in each dimension. As a result, its complexity increases rapidly for more sequences. The other algorithm, Hakata and Imai's C algorithm [1998], works for arbitrary number of strings. It is similar to another MLCS algorithm published recently, FAST-LCS [Chen *et al.*, 2006], in that they both use pairwise comparison algorithm to compute dominant points.

Parallel processing of the MLCS algorithms have been developed to further speed up the computation of LCS. Similar to sequential algorithms, early parallel algorithms were designed mostly for the special case of 2 sequences and cannot be easily generalized for three and more sequences [Babu and Saxena, 1997; Yap *et al.*, 1998; Xu *et al.*, 2005]. Korkin [2001] attempted to tackle the general MLCS problem but failed to achieve a near-linear speedup. In the most recent approaches, FAST-LCS [Chen *et al.*, 2006] and parMLCS [Korkin *et al.*, 2008], a near-linear speedup was reached for a large number of sequences. parMLCS is a parallel version of Hakata and Imai's C algorithm.

3 Fundamentals of Dominant-Point Method

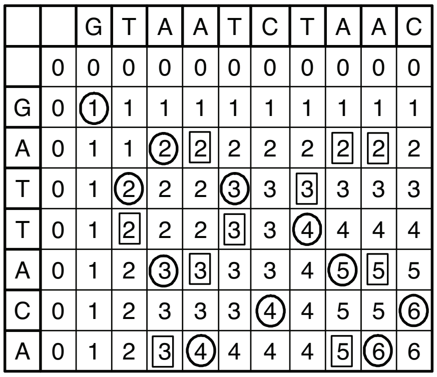
In this section, we introduce the basics and main ideas of the new dominant-point method.

Assume sequences are strings of characters defined over a finite alphabet Σ. Let $\mathbf{a}$ and $\mathbf{b}$ be two sequences of lengths n and k correspondingly. For sequence $\mathbf{a} = s_1 s_2 \dots s_n$, sequence $\mathbf{b} = s_{i_1} s_{i_2} \dots s_{i_k}$ is called a **subsequence** of $\mathbf{a}$ if $1 \le i_j \le n$, for $1 \le j \le k$, and $i_s < i_t$, for $1 \le s < t \le k$. Let $S = \{\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_d\}$ be a set of sequences over alphabet Σ. A **multiple longest common subsequence** (MLCS) for set S is a sequence $\mathbf{b}$ such that (i) $\mathbf{b}$ is a subsequence of $\mathbf{a}_i$, $1 \le i \le d$, and (ii) $\mathbf{b}$ is the longest one satisfying (i).

Let L be the score matrix for a set of d sequences, $\mathbf{a}_1$, $\mathbf{a}_2$, $\dots$, $\mathbf{a}_d$, as defined in Eq. (1). A point p in matrix L is denoted as $p = [p_1, p_2, \dots, p_d]$, 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 $L[p]$. A point $p = [p_1, p_2, \dots, p_d]$ in L is called a **match** if $\mathbf{a}_1[p_1] = \mathbf{a}_2[p_2] = \dots = \mathbf{a}_d[p_d]$. If a match p corresponds to character $s \in \Sigma$, i.e., $\mathbf{a}_i[p_i] = s$, it is denoted as $p(s)$.

For two points $p = [p_1, p_2, \dots, p_d]$ and $q = [q_1, q_2, \dots, q_d]$, we say that p **dominates** q if $p_i \le q_i$, for $1 \le i \le d$. If p dominates q, we denote this relation as $p \le q$. Similarly, p **strongly dominates** q if $p_i < q_i, 1 \le i \le d$. We denote this relation as $p < q$. p **does not dominate** q (and denote this as $p \not\le q$), if there is an i, $1 \le i \le d$, such that $q_i < p_i$. Note that $p \not\le q$ does not necessarily imply $q \le p$. For some points p and q, both $p \not\le q$ and $q \not\le p$ can be true. A match $p(s)$ is called a k**-dominant point**, or k**-dominant**, or **dominant at level** k, if (i) $L[p] = k$ and (ii) there is no other match $q(s)$ satisfying (i) and $q \le p$, i.e., p is not dominated by another point q with the same value k and for the same character s. The set of all k-dominants is denoted as D^k. The set of all dominant points is denoted as D.

An example of the score matrix L, the set of dominant points and rest of the matches for two sequences are shown in Figure 1.



		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	$\boxed{2}$	2	2	2	$\boxed{2}$	$\boxed{2}$	2
T	0	1	②	2	2	③	3	$\boxed{3}$	3	3	3
T	0	1	$\boxed{2}$	2	2	$\boxed{3}$	3	④	4	4	4
A	0	1	2	③	$\boxed{3}$	3	3	4	⑤	$\boxed{5}$	5
C	0	1	2	3	3	3	④	4	5	5	⑥
A	0	1	2	$\boxed{3}$	④	4	4	4	$\boxed{5}$	⑥	6

Figure 1: The dominant points and matches for two sequences, $\mathbf{a}_1 = GTAATCTAAC$ and $\mathbf{a}_2 = GATTACA$. The dominant positions are circled, while the remaining matches, which are not dominant, are squared.

A match $p(s)$ is called an s**-parent** for a point q, if $q < p$ and there is no other match $r(s)$ such that $q < r < p$. The set of all s-parents for q is denoted as $Par(q, s)$. 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 $Par(A, \Sigma)$. A point p in a set of points A is called a **minimal element** of A, if $q \not\le p$ for all $q \in A - \{p\}$. If $|A| = 1$, then its single element is defined to be a minimal element of A. The set of minimal elements is called **minima** of A.

It has been proven that $(k+1)$-dominants, D^{k+1}, is exactly the minima of the parent set, $Par(D^k, \Sigma)$, of k-dominants, D^k [Hakata and Imai, 1998]. It is easy to find the minima of $Par(D^k, \Sigma)$ directly [Chen *et al.*, 2006; Korkin *et al.*, 2008]. However, Theorem 1 below provides a more efficient way.

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

$Minima(Par(D^k, \Sigma)) = \cup_{s \in \Sigma} Minima(\{p(s) \,|\, p(s) \in Minima(Par(p, \Sigma)), p \in D^k\})$*, for* $k = 1, \ldots, K$*, where* K *is the length of the longest common subsequence.*

Proof of the theorem is omitted due to the size limitations of the paper. This theorem leads to a two-step procedure to compute the minima of the parent set $Par(D^k, \Sigma)$ as follows,

1. Minimize $Par(p, \Sigma)$ for each point $p \in D^k$,
2. Minimize each s-parent set of D^k, $s \in \Sigma$.

Both $Par(p, \Sigma)$ of p, $p \in D^k$, and s-parent set of D^k, $s \in \Sigma$, are smaller than $Par(D^k, \Sigma)$. By avoiding minimizing entire $Par(D^k, \Sigma)$, computation time is saved.

We have developed a fast divide-and-conquer algorithm for the procedure $Minima()$ mentioned above. It is based on the following Theorem 2.

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

Proof of the theorem can be derived from [Kung *et al.*, 1975; Bentley, 1980; Hakata and Imai, 1998]. Here we just give the central idea. Consider a divide-and-conquer algorithm on a set of N dominant points in d-dimensional space. The algorithm first evenly partitions these points into two subsets R and Q with respect to the d-dimensional coordinate so that the d-dimensional coordinates of points in R are greater than those of points in Q. Then, the algorithm recursively minimizes R and Q, respectively. Finally, after getting the minima of both R and Q, the algorithm removes points in R that are dominated by points in Q.

4 A new sequential algorithm for MLCS

In this section, we present a new dominant-point algorithm for MLCS 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 that the lengths of all sequences are equal to n. Our algorithm is based upon the following ideas:

1. Each position in a MLCS corresponds to a match in the score matrix L. Therefore, the search can be restricted to the set of all matches in L.
2. The number of dominant levels is equal to the size of MLCS. Moreover, there exists at least one dominant point at each level k, $k = 1, 2, \ldots, K$, which corresponds to the k-th position in the MLCS. Therefore, the search can be further restricted to all dominant points.
3. The set of dominant points in L can be computed recursively, where $D^{(k+1)}$ is computed, based solely on the dominant points of the previous level, D^k. The next two ideas explain the computation in more detail.
4. The set of $(k + 1)$-dominants, $D^{(k+1)}$, is the minima of the set of all parents for k-dominants, D^k, i.e. $D^{(k+1)} = Minima(Par(D^k, \Sigma))$.
5. $Minima(\,Par(D^k, \Sigma))$ can be computed in two steps as indicated in Theorem 1: (i) For each dominant point $p \in D^k$, compute $Minima(Par(p, \Sigma))$ and take a union of all such sets, $Par_s = \{p(s) \,|\, p(s) \in Minima(Par(p, \Sigma)), p \in D^k\}, s \in \Sigma$; (ii) D^{k+1} is a union of non-overlapping sets, $Minima(Par_s)$, i.e., $D^{k+1} = \cup_{s \in \Sigma} Minima(Par_s)$, for each $s \in \Sigma$.
6. Finally, $Minima()$ can be implemented using a fast divide-and-conquer approach as indicated in Theorem 2.

Based on these ideas, the sequential dominant-point and divide-and-conquer (Quick-DP) algorithm is as follows:

```
Algorithm Quick-DP($\{\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_d\}$, $\Sigma$)
  Calculation of dominant points
Preprocessing; $D^0 = \{[0,0,\ldots,0]\}$;  $k = 0$;
while $D^k$ not empty do {
  for $p \in D^k$ do {
    $B = Minima(Par(p,\Sigma))$;
    for $s \in \Sigma$ do{
       $Par_s = Par_s \cup \{p(s)|p(s) \in B\}$; }}
  $D^{k+1} = \cup_{s \in \Sigma} Minima(Par_s)$;
  $k = k + 1$; }
  Calculation of MLCS-optimal path
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 $p \in Par(q,\Sigma)$;
  $p = q$;
  $k = k - 1$; }
```

Quick-DP consists of two parts. In the first part, the set of all dominants is calculated iteratively, starting from a 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 MLCS-optimal path, corresponding to a MLCS, is calculated, 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. All MLCS can be enumerated systematically as well.

To efficiently enumerate 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 i-th sequence, starting from the $(j + 1)$-st position in that sequence. If s does not occur any more in the i-th sequence, the value of $T[s, j, i]$ is equal to $1 + max_{1 \leq k \leq d}\{|\mathbf{a}_k|\}$. The calculation of this preprocessing matrix $\mathbf{T}$ takes $O(\,n\,|\Sigma|\,d)$ time.

Let $|\Sigma|$ be the size of alphabet Σ and $|D|$ 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 $Par(p, \Sigma)$, $p \in D$, 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 of Quick-DP is

$$O(\,n\,|\Sigma|\,d + |D|\,|\Sigma|\,d\,(\,\log^{d-2} n + \log^{d-2}|\Sigma|\,)\,)$$

Based on the experimental evaluation of $|D|$ and the estimated complexities of Hakata and Imai's C algorithm [1998] and FAST-LCS [2006], we expect our approach to be significantly faster than C algorithm and FAST-LCS for large n.

5 A new parallel algorithm for MLCS

As shown in the previous section, calculating the set of $(k+1)$-dominants, D^{k+1}, requires computing the minima of parent set $Par(p,\Sigma)$, for $p \in D^k$, and the minima of s-parent Par_s, $s \in \Sigma$, of D^k. These sets can be calculated independently. 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 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 processor 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 processor.
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.

The pseudocode of the parallel algorithm Quick-DPPAR is as follows.

```
Algorithm Quick−DPPAR($\{a_1, a_2, \ldots, a_d\}, \Sigma, N_p$)
Proc$_0$ : Preprocessing; $D^0 = \{[0,0,\ldots,0]\}$; $k = 0$;
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 $p \in D_i^k$ do {
      $B$ = Minima(Parents($p$));
      for $s \in \Sigma$ do{
        $Par_{si} = Par_{si} \cup \{p(s) | p(s) \in B\}$; }}
    Send $Par_{si}, s \in \Sigma$, to Proc$_0$;
  Proc$_0$ : calculate $Par_s = \cup_{1 \le i \le N_p} Par_{si}, s \in \Sigma$;
  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$_0$ : $D^{k+1} = \cup_{1 \le i \le N_p} D_i^{k+1}$;
  $k = k + 1$; }
```

Quick-DPPAR assigns each s-parent set $Par_s, s \in \Sigma$, of D^k 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 of minimization. To utilize more than $|\Sigma|$ processors, we further parallelize the divide-and-conquer algorithm. Our parallel algorithm generates a binary tree during the execution of $Minima(Par_s)$. The root node of the tree contains the entire data set Par_s. The data set of each internal node of the tree is equally split into two subsets which correspond to the children of the node. The subsets are distributed among processors and the tree is built in parallel. Finally, the minima of each subset is solved recursively and the results are combined in the parent node.

6 Experimental Results

In our experiments, the algorithms were run on a Linux Server with 8 Intel Xeon(R) CPUs ($2.826GHz$) and $16GB$ memory. The programming environment is GNU C++. The algorithms were tested on random strings of length 100 to 4000 over alphabets of size 4 (e.g., nucleotide sequences) and 20 (e.g., protein sequences).

6.1 Sequential algorithm Quick-DP

The sequential method Quick-DP is compared with Hakata and Imai's A and C algorithms [Hakata and Imai, 1998]. The A algorithm is specifically designed for 3 strings and is the most efficient one so far for 3 strings. The C algorithm can work with any number of strings. We implemented Hakata and Imai's A algorithm and C algorithm according to their paper. The data structures of them were carefully designed to reduce their computation time.

For each string length of an alphabet, we generated 10 sets of 3 random strings. Quick-DP and Hakata and Imai's A and C algorithms were tested on the same data sets and their average running times for each length of sequence are shown in Fig. 2. Fig. 2 shows that Quick-DP is comparable and slightly faster than Hakata and Imai's A algorithm on 3 strings. We note that A was designed specially for the case of 3 strings, while Quick-DP works for an arbitrary number of sequences. Fig. 2 also shows that Hakata and Imai's C algorithm is significantly slower.

Next, we compared Quick-DP with another state-of-the-art sequential algorithm FAST-LCS [Chen *et al.*, 2006] and Hakata and Imai's C algorithm on test sets consisting of more than 3 sequences. The results in Fig. 3 shows that Quick-DP is several orders of magnitude faster than FAST-LCS and C algorithm. For instance, for random DNA sequences, Quick-DP is over 1,000 times faster than FAST-LCS on sequences of length 140 and over 1,000 times faster than C algorithm on problems of length 200.

6.2 Parallel algorithm Quick-DP$_{PAR}$

The parallel algorithm Quick-DPPAR was implemented using multithreading. Following the parallelization scheme in Section 5, our implementation consists of 1 master thread and multiple slave threads. The master thread allocates dominant points to slaves to perform time-consuming computation.

We first evaluated the speedup of our parallel algorithm Quick-DPPAR over sequential algorithm Quick-DP. Speedup is defined as the ratio of sequential time and parallel time. Similar to previous test data, we generated 10 sets of 5 random strings each for alphabet 4 and 20, respectively. We ran Quick-DPPAR using different number of slave threads. Fig. 4 shows the results for sequence lengths 1000, 1500, and 2000. Near-linear speedups were achieved for all these cases.

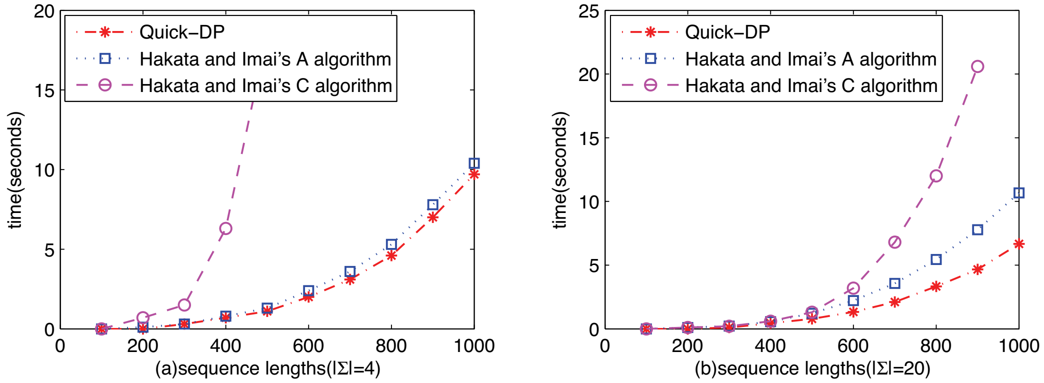


Figure 2: The average running time of Quick-DP and Hakata and Imai's A and C algorithms for 3 random strings.

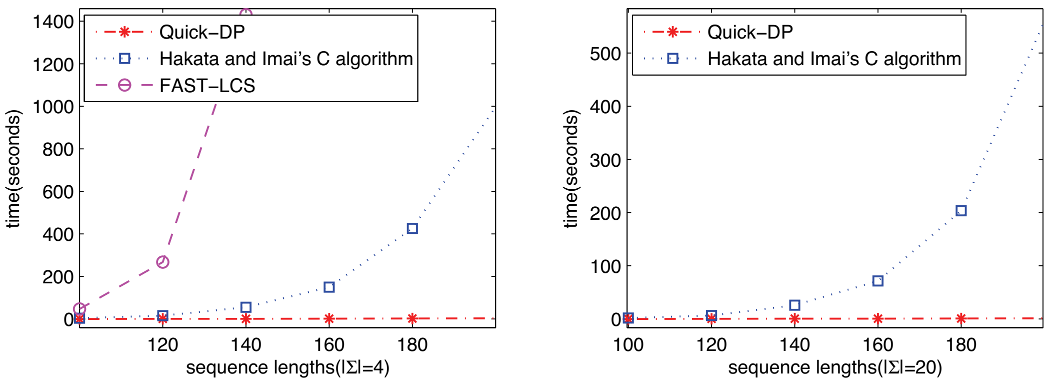


Figure 3: The average running time of Quick-DP, Hakata and Imai's C algorithm, and FAST-LCS on 5 random strings of different lengths. FAST-LCS does not work on strings of large alphabet, such as 20.

Then, we measured the running time of Quick-DPPAR on strings of various lengths. We ran Quick-DPPAR using 8 slave threads on test sets of 5 random DNA sequences. The results in Fig. 5 demonstrate the efficiency of Quick-DPPAR. We have also applied Quick-DPPAR to determine a MLCS for 11 proteins from the family of melanin-concentrating hormone receptors (MCHR). The lengths of the protein sequences range from 296 to 423 amino acids. As a result, it took Quick-DP 939 seconds to detect a MLCS. For Quick-DPPAR, the time was reduced to 185 seconds.

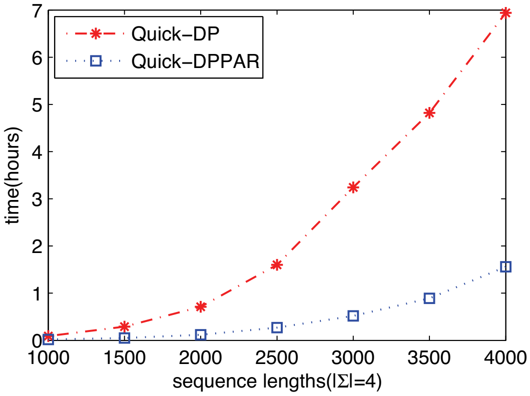


Figure 5: The average running times of our parallel Quick-DPPAR and sequential Quick-DP algorithms on MLCS problems of 5 random sequences.

7 Conclusions

In this paper, we have presented fast sequential and parallel algorithms for MLCS problems of arbitrary number of strings. The algorithms improve existing dominant point methods through new efficient methods for computing dominant points. For problems of 3 strings, the sequential algorithm is as fast as the best existing MLCS algorithm designed for the 3-string case, Hakata and Imai's A algorithm. For problems of more than 3 strings, the sequential algorithm is much faster than the best existing algorithms, achieving more

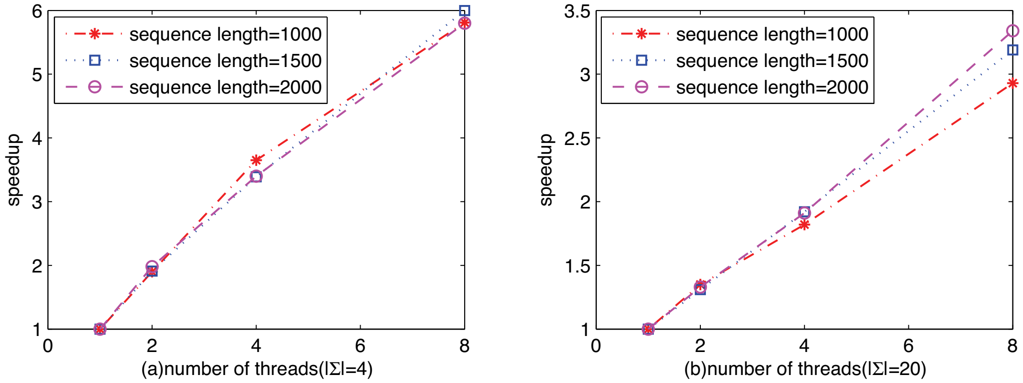


Figure 4: The speedup of our parallel method Quick-DPPAR over sequential algorithm Quick-DP on MLCS problems of 5 random strings.

than a thousand times faster performance on larger size problems. The parallel implementation is efficient and achieves a near-linear speedup over the sequential algorithm.

Acknowledgement

This research was supported in part by the Shumaker Endowment in Bioinformatics and NIH Grant R33GM078601.

References

[Apostolico *et al.*, 1992] A. Apostolico, S. Browne, and C. Guerra. Fast linear-space computations of longest common subsequences. *Theor. Comput. Sci.*, 92(1):3–17, 1992.

[Babu and Saxena, 1997] K. Nandan Babu and Sanjeev Saxena. Parallel algorithms for the longest common subsequence problem. In *Proc. 4th Intl. Conf. on High-Performance Computing*, pages 120–125, 1997.

[Bentley, 1980] Jon L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, 1980.

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

[Chin and Poon, 1990] Francis Y. L. Chin and Chung K. Poon. A fast algorithm for computing longest common subsequences of small alphabet size. *J. Inf. Process.*, 13(4):463–469, 1990.

[Hakata and Imai, 1998] Koji Hakata and Hiroshi Imai. Algorithms for the longest common subsequence problem for multiple strings based on geometric maxima. *Optimization Methods and Software*, 10:233–260, 1998.

[Hirschberg, 1977] Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.

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

[Korkin *et al.*, 2008] Dmitry Korkin, Qingguo Wang, and Yi Shang. An efficient parallel algorithm for the multiple longest common subsequence (mlcs) problem. In *ICPP '08: Proc. 37th Intl. Conf. on Parallel Processing*, pages 354–363, Washington, DC, USA, 2008. IEEE Computer Society.

[Korkin, 2001] Dmitry Korkin. A new dominant point-based parallel algorithm for multiple longest common subsequence problem. Technical report, Department of Computer Science, University of New Brunswick, N.B. Canada, 2001.

[Kung *et al.*, 1975] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.

[Masek and Paterson, 1980] W. J. Masek and M. S. Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, pages 18 – 31, 1980.

[Rick, 1994] Claus Rick. New algorithms for the longest common subsequence problem. Technical report, University of Bonn, 1994.

[Sankoff, 1972] David Sankoff. Matching sequences under deletion/insertion constraints. *Proc. Natl. Acad. Sci. USA*, 69(1):4 – 6, 1972.

[Smith and Waterman, 1981] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195 – 197, 1981.

[Xu *et al.*, 2005] Xiaohua Xu, Ling Chen, Yi Pan, and Ping He. *Computational Science and Its Applications - ICCSA 2005*, volume 3482, chapter Fast Parallel Algorithms for the Longest Common Subsequence Problem Using an Optical Bus, pages 338–348. Springer Berlin / Heidelberg, 2005.

[Yap *et al.*, 1998] Tieng K. Yap, Ophir Frieder, and Robert L. Martino. Parallel computation in biological sequence analysis. *IEEE Trans. Parallel Distrib. Syst.*, 9(3):283–294, 1998.