An Efficient Algorithm for the Discovery of Multiple Longest Common Subsequence

Anusha Koneru, Gowtham D, Navya Sree Y & Venkata Rao J

Department of CSE, K L University
E-mail: anusha.koneru13@gmail.com, gowtham.deva1283@gmail.com,
navya.yarramsetti@gmail.com, venkat2all@gmail.com

***Abstract* – In many applications of bioinformatics and computational genomics, the discovery of the longest common subsequence (LCS) of multiple strings is an NP-hard problem which cannot be solved in polynomial time. As the rapid growth of size and complexity of biological data increasing, significant efforts have been made and enhancements are made for the efficiency to an arbitrary number of strings. In order to address these issues, we present an algorithm for the multiple LCS (or MLCS) problem, on discovering an LCS of any number of strings, and its parallel realization. This is based on the dominant point approach and employs a fast divide-and conquer technique to compute the dominant points. Our algorithm demonstrates the same performance as the fastest existing MLCS algorithm designed for a case of three strings. On the other hand for 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. Also it reveals that a near-linear speedup has occurring with respect to the sequential algorithm on applying to random and biological sequences.**

Keywords – longest common subsequence (LCS), multiple longest common subsequence (MLCS), dynamic programming, dominant point method, divide and conquer, multithreading.

I. 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, cannot be solved in polynomial time.1 When presented with a NP-hard problem, we can take one of three possible strategies: 1) Run a super-polynomial algorithm anyway, 2) Assume that the input is random, and find an algorithm that will perform well in the average case, 3) Settle for a suboptimal solution (an approximation) that can be found in polynomial time, and prove that this solution is "good enough". It addresses either the simplest case of MLCS of two strings, also known as the longest common subsequence (LCS) problem or the problem's special case of three strings . Although several methods have been proposed for the general case of any given number of strings 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. 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. 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. In our method, we implement a divide-and-conquer technique to construct dominant point sets efficiently. Unlike other MLCS algorithms, including FAST-LCS and parMLCS 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.

II. MLCS PROBLEM FORMULATION AND EXISTING METHODS

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

A. Problem Definition

Definition 1. Let a be a sequence of length n over a finite alphabet $\sum$: a = $s_1 s_2 \ldots s_n$; $s_i \in \sum$. Sequence b=$s_{i1} s_{i2} \ldots s_{ik}$ is called a subsequence of 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	$\{A, C, G, U\}$	$\sim$10–10^4 [16]
Genome	$\{gene_1, gene_2, \ldots, gene_k\}$	$\sim$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, $1 \le r < t \le k$:

$i_r < i_t$:

Definition 2. Let S={a_1,a_2, . . .,a_d}be a set of sequences over a finite alphabet $\sum$. The MLCS for set S is a sequence b such that:

1. b is a subsequence of a_i for each i;
2. b is the longest among all sequence satisfying point 1.

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

a1=i n f o r m a t i c s;

a2=p r o t e o m i c s;

a3=a r i t h m e t i c s:

one of the multiple longest common subsequences is b1=r m i c s and another is b2=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).

B. Dynamic Programming Methods

Classical methods for the MLCS problem are based on dynamic programming. 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 an $n_1 x n_2$ score matrix L in which L[i,j],$0 \le i \le n_1$,$0 \le j < n_2$,is the length of an LCS between two prefixes a_1[1,…,i] and a_2[1,…,j].specifically, the score matrix L is defined as follows:

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

The definition of the matrix L can be naturally generalized to a case of N sequences: for each position L[i_1, i_2, . . . , 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 a1 =GATTACA and a2 =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 a1 = GATTACA and a2 = 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. Various approaches have been introduced to reduce the complexity of dynamic programming. Unfortunately, these approaches primarily address a special case of two sequences.

C. Dominant Point Methods

Let L be the score matrix for a set of d sequences {a_1, a_2, . . . ,a_d}over a finite alphabet $\sum$. A point p in matrix L is denoted as p =[p_1, p_2, . . . , p_d], where each pi is a coordinate of p for the corresponding string 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, . . . , p_d] in L is called a match if a1[p_1] =a2[p_2] =…….=ad[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 $p_i \leq q_i$, for all i=1, 2, . . . , d. If d dominates q, we denote this relation as $p \leq q$.

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

A point p=[p1,p2,…,pd] does not dominate a point $q=[q_1,q_2,\ldots,q_d]$(denoted as $p \not\leq q$),if i, $1 \leq i \leq d$,$q_i \leq p_i$.note that $p \not\leq q$ does not necessarily imply $q \leq p$, i.e., for some points p and q ,$p \leq q$ and $q \not\leq p$ 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; $q \neq p$, satisfying 1 dominates p($q \leq p$).

The set of all k-dominants is denoted as Dk. 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 points. They are the critical points sufficient to define the overall contour shape.

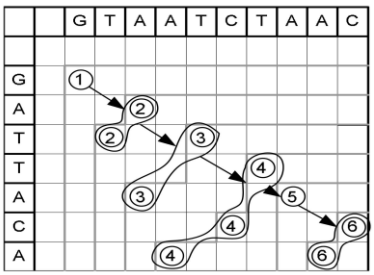

Fig. 2. The process of dominant point approach for two sequences a1 =GATTACA and a2 =GTAATCTAAC. Broken lines are contours or boundary of different level of points. Dominant points at the same level are encircled together. Arrows in the figure point from the set of k-dominants to the set of(k+1)-dominants, $1 \leq k \leq |MLCS| \leq 1$.

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. 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, $1 \leq k \leq |MLCS| \leq 1$. 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 a1 and a2. 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, $1 \leq k \leq |MLCS|-1$.

The dominant point approach has been successfully applied to a case of two sequences. In, three dominant point algorithms for three or more sequences were proposed. One of the algorithms, Algorithm A, 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, works for arbitrary number of strings. It is similar to another MLCS algorithm published recently, FAST-LCS , in that they both use a pair wise comparison algorithm to compute the dominant points.

D. Parallel MLCS Methods

Many parallel MLCS algorithms have been developed to speed up the computation .The majority of them are designed for LCS problem of two sequences. On CREWPRAM model, Apostolico et al. 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 $m<n$. Lu and Lin 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 $\text{Log}^2 m \log\log m \leq \log n$. Babu and Saxena 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 derived a n+3m+p time algorithm with an array of M(m+1)=2 cells. Freschi and Bogliolo 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 $1 \leq p \leq \max(m,n)$.

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 could not achieve a consistent speedup on the test set of multiple sequences. In the most recent approaches, FASTLCS and parMLCS, a near-linear speedup was reached for a large number of sequences.

parMLCS is a parallel version of Hakata and Imai's C algorithm.

III. 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 a1,a2, . . . ,ad are sequences over alphabet ∑, and the lengths of all sequences are equal to n.

A. Sequential Algorithm Design

Definition 6. A match p is called an s-parent (a parent with respect to the s symbol s Є∑) 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, U_s Є∑Par(A,s), for A is denoted as Par(A, ∑).

Definition 7. A point p in a set of points A, is called a minimal element of A, if for all q ∑A-{p} : q ≤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 that (k+1)-dominants, D^{k+1}, 0 ≤ k≤|MLCS|-1, constitute exactly the minima of the parent set Par(D^k, Σ)of k-dominants D^k, i.e., D(k+1) =Minima(Par(D^k, ∑)); 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 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,∑):

1) For each dominant point q∑D^k, compute Minima(Par(q, ∑)) and take a union of all such sets,

   Pars = {q(s) |q(s)ЄMinima(Par(q,∑)); q ЄD^k},sЄ∑:

2) Compute the union of nonoverlapping sets Minima(Pars,sЄ∑.

```
    Algorithm Quick−DP ({a1, a2, . . . , ad}, Σ)
    Computing dominant points
    Preprocessing;
01  D^0 = {[−1,−1, . . . ,−1]}; k = 0; Par_s = ∅ for all s ∈ Σ;
02  while D^k not empty do {
03    for q ∈ D^k do {
04      B = Minima(Par(q, Σ));
05      for s ∈ Σ do{
06        Par_s = Par_s ∪ {q(s) | q(s) ∈ B}; }}
07    D^{k+1} = ∪_{s∈Σ} Minima (Par_s) ;
08    k = k + 1; }
    Finding a MLCS
09  pick a point p = [p1, p2, . . . , pd] ∈ D^{k−1};
10  while k − 1 > 0 do {
11    current LCS position = a1[p1];
12    pick a point q such that p ∈ Par(q, Σ) ;
13    p = q;
14    k = k − 1; }
```

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

B. Implementation Details and Complexity Analysis

We add a preprocessing step in the beginning of the algorithm to efficiently find all parents of each ominant point. We calculate a preprocessing matrix T={T[s,j,i]}; sЄ∑, 0≤j ≤max1≤k≤dfjakjg; 1≤ i≤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+max1≤k≤d{|a_k|}. With the matrix T, the s-parent p=[p_1,p_2, . . . ,p_d] of a point q=[q1, q2, . . . ,qd] can be calculated in O(d) time, using the formula pi = T(s,qi, i),1≤id. The calculation of this preprocessing matrix T takes O(n|∑| d) time, where |∑| is the size of alphabet ∑.

IV. A NEW PARALLEL MLCS ALGORITHM

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,∑), for qЄD^k, and the minima of s-parent set Pars, sЄ∑, 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 Np+1 processors, the parallel algorithm uses one as the master and Np 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, . . .), 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 Pars, s 2 _, as the union of the parents from slave processors and distributes the resulting s-parent set among slaves.

```
   Algorithm Quick−DPPAR ({a_1, a_2, ..., a_d}, Σ, N_p )
01 Proc_0 : Preprocessing; D^0 = {[−1,−1,...,−1]}; k = 0;
02 while D^k not empty do {
03   Proc_0 : distribute elements of D^k
     Each processor, Proc_i, 1 ≤ i ≤ N_p, performs:
04     get D_i^k from Proc_0;
05     for q ∈ D_i^k do {
06        B = Minima(Par(q, Σ));
07        for s ∈ Σ do{
08           Par_is = Par_is ∪ {q(s) | q(s) ∈ B}; }}
09     Send Par_is, s ∈ Σ, to Proc_0 ;
10   Proc_0 : calculate Par_s = ∪_{1≤i≤N_p} Par_is, s ∈ Σ;
11   Proc_0 : distribute Par_s, s ∈ Σ;
     Each processor, Proc_i, 1 ≤ i ≤ N_p, performs:
12     get Par_s, s ∈ Σ;
13     D_i^{k+1} = Minima(Par_s);
14     send D_i^{k+1} to Proc_0;
15   Proc_0 : defines D^{k+1} = ∪_{1≤i≤N_p} D_i^{k+1} ;
16   k = k + 1; }
```

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

5. Each slave processor i is assigned to find the minimal elements only of one s-parent set Pars.

6. Each slave processor i computes the set D_i^{k+1} of (k+1)-dominants of Pars and sends it to the master.

7. The master processor computes $D^{k+1} = D_1^{k+1} D_2^{k+1}$ U . . . UD^{k+1} Np and goes to step 2.

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

In Quick-DPPAR, each s-parent set Pars; sЄ∑, of Dk is assigned to a slave processor to compute the minima of Pars using our divide-and-conquer method. So, as many as|∑|slave processors can work simultaneously in this step. To utilize more than |∑| 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 Procr can have their own children too. Thus, during the recursive execution of the program, a binary tree is formed based on this parentchildren 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.

V. CONCLUSION

We have mainly considered based on two contributions: a new algorithm and parallelization of that algorithm. Our theoretic analysis of the parallel algorithm predicted that speedup is asymptotically linear. 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. 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.

REFERENCES

[1] A pointing method of an object in real space using dominant eye's view field, First IEEE Technical Exhibition Based Conference, Robotics and Automation, 2004. TExCRA '04. Mitsudo, Y. Graduate Sch. of Information Syst., Miyazaki, E. ; Idesawa, M.

[2] Finite Automata Inspired Model for Dominant Point Detection: A Non-Parametric Approach, International Conference on Computing: Theory and Applications, 2007. ICCTA '07. Dinesh, R. Dept. of Studies in Comput. Sci., Mysore Univ., Karnataka

[3] New dominant point detection for image recognition, Circuits and Systems, 1999. ISCAS '99. Proceedings of the 1999 IEEE International Symposium, Chun-Pong Chau.

[4] Fast subspace-based tensor data filtering, 16th IEEE International Conference on Image Processing (ICIP), 2009,
MaroJ. Ecole Centrale Marseille-Inst. Fresnel, Marseille, France

[5] Approximate dynamic programming of continuous annealing process IEEE International Conference on Automation and Logistics, 2009. ICAL'09. Chao Guo ; Xue.

[6] A novel service recovery method based upon Bellman dynamic programming, International Conference on Computer and Information Application (ICCIA), 2010 Coll. of Comput. Sci. & Technol., Harbin Eng. Univ., Harbin, China Wang Hui-qiang; Guangsheng Feng

[7] A Survey of Approximate Dynamic Programming, International Conference on Intelligent Human-Machine Systems and Cybernetics, 2009. IHMSC '09. Coll. of Mechatron. Eng. & Autom., Nat. Univ. of Defense Technol., Changsha, China Peng Hui ; Zhu Hua-yong ; Shen Lin-cheng

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

◈◈◈