Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2012, Article ID 310328, 17 pages
doi:10.1155/2012/310328

Research Article

FACC: A Novel Finite Automaton Based on Cloud Computing for the Multiple Longest Common Subsequences Search

Yanni Li,[1,2] Yuping Wang,[1] and Liang Bao[2]

[1] *School of Computer Science and Technology, Xidian University, Xi'an 710071, China*
[2] *School of Software, Xidian University, Xi'an 710071, China*

Correspondence should be addressed to Yanni Li, yannili@mail.xidian.edu.cn

Received 14 April 2012; Accepted 30 August 2012

Academic Editor: Hailin Liu

Searching for the multiple longest common subsequences (MLCS) has significant applications in the areas of bioinformatics, information processing, and data mining, and so forth, Although a few parallel MLCS algorithms have been proposed, the efficiency and effectiveness of the algorithms are not satisfactory with the increasing complexity and size of biologic data. To overcome the shortcomings of the existing MLCS algorithms, and considering that MapReduce parallel framework of cloud computing being a promising technology for cost-effective high performance parallel computing, a novel finite automaton (FA) based on cloud computing called FACC is proposed under MapReduce parallel framework, so as to exploit a more efficient and effective general parallel MLCS algorithm. FACC adopts the ideas of matched pairs and finite automaton by preprocessing sequences, constructing successor tables, and common subsequences finite automaton to search for MLCS. Simulation experiments on a set of benchmarks from both real DNA and amino acid sequences have been conducted and the results show that the proposed FACC algorithm outperforms the current leading parallel MLCS algorithm FAST-MLCS.

1. Introduction

Searching for MLCS is a classic dynamic programming problem. Let Σ be a set of the finite or infinite alphabet, and $X = x_1, x_2, \ldots, x_m$ be a finite sequence of symbols drawn from Σ, that is, $x_i \in \Sigma$, $i = 1 \sim m$. A sequence $Z = z_1, z_2, \ldots, z_k$ is called a subsequence of X if it satisfies $z_j = x_{i_j}$, $j = 1 \sim k$ and $1 \le i_1 < i_2 < \cdots < i_k \le m$, that is, $Z = x_{i_1}, x_{i_2}, \ldots, x_{i_k}$. For two given sequences X and Y, Z is called a common subsequence (CS) of X and Y if and only if Z is simultaneously a subsequence of both X and Y. When no other common subsequence is longer than Z, Z is named the longest common subsequence (LCS) of X and Y. Similarly,

given a set of sequences $\{S_1, S_2, \ldots, S_k\}$ drawn from Σ, a subsequence is called their multiple longest common subsequence (MLCS) if it is a subsequence for all of them (this subsequence is called a common subsequence), and no other common subsequence is longer than it.

Searching for the MLCS is significant for a number of applications in the areas of bioinformatics, information processing, data mining, and pattern recognition [1] and so forth. In the above-mentioned application areas, information can be usually represented as a sequence over a finite or infinite alphabet. For instance, a protein can be expressed as a sequence of twenty different symbols (amino acid) in biology, and a DNA sequence can be described as a sequence of four symbols A, C, G, and T [2]. A program source code can be represented as a sequence over alphabets, that is, ASCII or Unicode. In a specific domain, similarly, web information can be regarded as a word sequence of the domain-ontology. We can explore and discover valuable information by making a comparison and analysis of the sequences. For example, in the biological field, sequence comparison has been successfully used to establish a link between cancer-causing genes and normal genes. By finding the similarity between gene sequences, we can obtain valuable information on genetic diseases [3–5], too. In web information search and data mining, by determining the MLCS of word sequences of the domain-ontology, we can not only increase the accuracy rate of information retrieval, but also mine considerably valuable information. In the programming analysis, by discovering the MLCS among program source codes, we can acquire their redundancy and similarity and eliminate the redundancy or detect clone codes.

Searching for the LCS of two or more given sequences is a classic NP-hard problem [6]. Wagner and Fischer [7] first introduced a classic dynamic programming algorithm (DP) for solving the LCS problem of two sequences, and both its time and space complexity are $O(mn)$, where m and n are the lengths of the two sequences, respectively (the same notations are used in what follows). However, its main drawbacks are as follows: it has a higher time and space complexity, and it can only find the LCS of two relatively short sequences, but can hardly deal efficiently with the LCS problem of the two longer sequences. Hirschberg [8] presented a new LCS algorithm based on the idea of the "divide and conquer" approach, which reduces space complexity to $O(m + n)$ and gives a better solution to the problem of the longer sequences. Using a decision tree model, Ullman et al. [9] (1976) devised a better LCS algorithm with lower bound $O(mn)$ of the time complexity. Based on the effective detection of all the major matched points between compared sequences, Hunt and Szymansi [10] designed a new LCS algorithm with the time complexity of $O(n \log \log n)$, where n is the maximum length of the compared sequences. The algorithm was later simplified by Bespamyatnikh and Segal [11]. Masek and Paterson [12] put forward an improved dynamic programming algorithm by using a fast algorithm to compute the sequence editing distance, which reduces the time complexity to $O(n^2 / \log n)$.

To further improve the time complexity of LCS algorithms, researchers have begun to study the parallel LCS algorithm. Based on a CREW-PRAM model, Aggarwal and Park [13] and Apostolico et al. [14] proposed parallel LCS algorithms with the time complexity of $O(\log m \log n)$ by using $mn/\log m$ processors, respectively. Freschi and Bogliolo [15] presented another new parallel LCS algorithm by using some packed arrays and $m+n$ processors based on the run-length-encoded (RLE) string. The time complexity of the algorithm is $O(m'n + mn' - m'n')$, where m' and n' are the numbers of runs in their RLE representation, respectively. Liu and Chen [16] presented a specific parallel MLCS algorithm over alphabet {A, C, G, T}, FAST-LCS, based on proposed pruning rules which is more efficient than the previous works. The space and time complexity of the algorithm are $\max\{4*(n+1)+4*(m+1), L\}$ and $O(|\mathrm{LCS}(X, Y)|)$, respectively, where L is the number of identical character pairs and

$|\mathrm{LCS}(X, Y)|$ is the length of the LCS of X and Y. Wang et al. [17–19] developed the efficient MLCS algorithms parMLCS and Quick-DP, respectively, based on dominant points approach, which have reached a near-liner speedup for large number of sequences. It is worth mentioning that Yang et al. [20], as a new attempt, develop an efficient parallel algorithm on GPUs for the LCS problem. But regretfully, the algorithm is not suitable for the general MLCS problem.

To meet the needs of practical applications, some researchers have also studied some variations of the LCS problems, such as the longest common increasing subsequence (LCIS) problems, the longest increasing subsequence (LIS) problems, and the common increasing subsequence (CIS) problems, and so forth. Fredman [21] proposed an algorithm for LIS problems. The optimal time complexity of the algorithm is $O(n \log n)$ when the average length of sequences equals n. By combining LCS with LIS, Yang et al. [22] defined a common increasing subsequence (CIS) and designed a dynamic programming algorithm for two sequences CIS problems. The space complexity of the algorithm is $O(mn)$. Brodal et al. [23] present an algorithm for finding a LCIS of two or more input sequences. For two sequences of lengths m and n, where $m \geq n$, the time complexity and space complexity of the algorithm are $O((m + nl) \log \log \sigma + \mathrm{Sort})$ and $O(m)$, respectively, where l is the length of an LCIS, σ is the size of the alphabet, and Sortis the time to sort each input sequence.

Nevertheless, the aforementioned algorithms have the following disadvantages: (1) most of them are inapplicable to the problems with more than two sequences (especially a considerable number of sequences, a large alphabet, and a long average length of the sequences); (2) the efficiency and effectiveness of a few parallel algorithms remain to be improved; (3) the parallel implementations of the algorithms are of a certain difficulty due to their complicated concurrency, synchronization, and mutual exclusion, that is, none of the existing algorithms employed simple and cost-effective high performance parallel computing framework such as MapReduce for implementing their algorithms; (4) most of the algorithms did not provide an abstract and formal description to reveal the inherent properties of the MLCS problem. To overcome these shortcomings, a novel finite automaton based on cloud computing for the MLCS problem was proposed in this paper. The main contributions of this paper are as follows.

(1) All common subsequences (CS) of the n sequences are abstracted as the language over their common alphabet Σ, that is, every CS of n sequences is a sequence over the Σ.

(2) Based on the ideas of the matched pair and finite automaton, a novel finite automaton which can recognize/accept all of the CSs of n sequences is presented.

(3) A formal definition of the finite automaton was introduced, and its properties were verified.

(4) A novel parallel algorithm FACC was proposed. FACC is by abstracting the MLCS problem as one of searching for the longest path on the finite automaton, and was implemented based on the new parallel framework MapReduce of cloud computing and a variety of optimization techniques.

(5) A quantitative analysis of the time and space complexity of the FACC was conducted.

(6) The algorithm FACC was validated on the DNA and amino acid sample sequences from ncbi and dip databases. Then, the comparison of the time performance between FACC and the leading algorithm: FAST-LCS [16] was made. The experimental results show that the proposed FACC outperforms FAST-LCS.

Table 1: Notations and their meanings.

Notations	Meaning
Σ	The common alphabet of N sequences
Σ_i	The set of alphabets in the ith sequence
S_k $(k = 1, 2, \ldots, d)$	The kth sequence, with d being the number of sequences drawn from Σ
T	A set of sequences drawn from Σ
$P = [p_1, p_2, \ldots, p_d]$	A matched pair
$Q = \{Q_0, Q_1, \ldots, Q_n\}$	The set of the FA states
δ	The FA's transition function
$Q_0 = (0, 0, \ldots, 0)$	The FA's initial state
F	The set of the FA's final states
Atm	The FA recognizing/accepting all of the CSs of sequences from T
Σ^*	Kleene closure of Σ, $\Sigma^* = \{x \mid x$ is a sequence from Σ with the length of zero or nonzero$\}$
$L(Atm)$	A language recognized/accepted by the Atm

The rest of this paper is organized as follows. Section 2 introduces some notations and concepts in this paper for convenience discussion. Section 3 presents the finite automaton Atm for recognizing common subsequence and its basic properties. Section 4 proposes a new algorithm called finite automaton based on cloud computing (FACC) and describes its implementation in detail. Section 5 explains the analysis of the time and space complexity of FACC. The experiments are made and the analysis results are explained in Section 6. Finally, Section 7 concludes the research.

2. Notations and Basic Concepts

For convenience, the following notations are adopted in Table 1.

Note that a sequence over some alphabet is a finite sequence of symbols drawn from that alphabet. According to the formal language and automaton theory, we can view the common subsequences of all sequences on set T as a language L of the common alphabet Σ, and then regard the MLCS of the over sequences as the one or several longest statements of the language L. Based on this idea, a novel finite automaton Atm which can recognize/accept the L is designed, and the Atm for MLCS was constructed quickly based on a new constructing-searching algorithm and the MapReduce parallel framework of cloud computing proposed in this paper, meanwhile the MLCS can be easily achieved.

For easy understanding, some basic concepts in the following are introduced, and the properties of the Atm are discussed.

Definition 2.1. Suppose $S_k \in T$ is a sequence over Σ for $k = 1 \sim d$. Let $S_k[pk] \in \Sigma$ denotes the p_kth $(p_k = 1, 2, \ldots, |S_k|)$ character in sequence S_k. For the sequences $S_1, S_2, \ldots, S_d$ from T, vector $p = [p_1, p_2, \ldots, p_d]$ is called a matched pair of the sequences, if and only if $S_1[p_1] = S_2[p_2] = \cdots = S_i[p_i] = \cdots = S_d[p_d] = \text{ch}$ $(i = 1, 2, \ldots, d, \text{ch} \in \Sigma)$, where ch is the character corresponding to the matched pair p, denoted as $\text{ch}(p)$.

For example, for two sequences $S_1 = abcfg$ and $S_2 = bacgf$, one can get the following matched pairs by Definition 2.1: [1,2], [3,3], [4,5], [2,1], and [5,4] with their corresponding ch's being a, c, f, b, and g, respectively.

Definition 2.2. For the sequences $S_1, S_2, \ldots, S_d$ from T, $p = [p_1, p_2, \ldots, p_d]$ and $q = [q_1, q_2, \ldots, q_d]$ are two matched pairs. one calls $p = q$ if and only if $p_i = q_i$ for $i = 1, 2, \ldots, d$. If $p \neq q$ and $q_i < p_i$ for $i = 1, 2, \ldots, d$, one calls p a *successive matched pair* of q, and denote it as $q < p$. If $q < p$ and there does not exist a matched pair $r = [r_1, r_2, \ldots, r_d]$ for $S_1, S_2, \ldots, S_d$ such that $q < r < p$, one calls p a *direct successor matched pair* of q, denoted as $q \rightarrow p$.

Definition 2.3. For the sequences $S_1, S_2, \ldots, S_d$, from T, let $q = [q_1, q_2, \ldots, q_d]$ be a matched pair. If there does not exist a matched pair $p = [p_1, p_2, \ldots, p_d]$ $(p \neq q)$ such that $p < q$, one terms q an initial matched pair. In general, there may be more than one initial matched pair from T.

Taking above sequences S_1 and S_2 as an example, we can see that the matched pairs [3,3], [4,5], and [5,4] are the successive matched pairs of matched pairs [1,2] and [2,1]. Moreover, the matched pair [3,3] is a direct successive matched pair of the matched pairs [1,2] and [2,1], wherein [1,2] and [2,1] are two initial matched pairs in total for sequences S_1 and S_2.

Based on above the definitions, the following conclusion can be easily inferred.

Lemma 2.4. *The total number of all possible initial matched pairs is less than or equal to* $|\Sigma|$ *regardless of* $|T|$.

3. Finite Automaton *Atm* for Recognizing Common Subsequence and Its Basic Properties

It can be seen from the above discussion that the characters in the MLCS from T must be the characters corresponding to their matched pairs. In the following, based on some ideas and concepts from finite automaton (*Atm*), we can see the matched pairs of T as states of the *Atm* and construct the *Atm* which can recognize/accept all the CSs of T by defining a specific transition function. The formal definition of the *Atm* is as follows.

Definition 3.1. The common subsequence finite automaton *Atm* is a 5-tuple, that is,

$$Atm = \{Q, \Sigma, \delta, Q_0, F\}, \tag{3.1}$$

where Q—$Q = \{Q_0, Q_1, \ldots, Q_n\}$ is a finite set of states of the *Atm*, where Q_i is the state corresponding to the ith matched pair of T for $i = 0 \sim n$; Σ—the common alphabet of sequences set T; δ—the transition function $\delta : Q \times \Sigma \rightarrow Q$. For for all $a \in \Sigma$ and Q_i, one defines the transition function δ as follows

$$\delta(Q_i, a) = \begin{cases} Q_j, & \text{if } Q_i \longrightarrow Q_j,\ \text{ch}(Q_j) = a \\ \phi, & \text{else.} \end{cases} \tag{3.2}$$

Q_0—Let $Q_0 = (0, 0, \ldots, 0)$, the initial state of the *Atm*. F—The set of final states of the *Atm*, that is, $F = \{Q_i \mid \text{if } \delta(Q_i, a) = \Phi\ \forall a \in \Sigma\}$.

It can be seen from Definition 3.1 that the *Atm* is a deterministic finite automaton (DFA), but it is different from the normal DFA. The *Atm* can be partial, that is, every state in the *Atm* can be initial or final state.

What follows are the formal definitions of $L(Atm)$ and MLCS recognized/accepted by the *Atm*.

Definition 3.2. For $Atm = \{Q, \Sigma, \delta, Q_0, F\}$ defined by Definition 3.1, a character sequence $x \in \sum^*$ is called to be recognized/accepted by the *Atm* if and only if for $\delta(Q_i, a) \in Q'$, where, $Q_i \in Q$, $Q' = Q - \{Q_0\} = \{Q_1, Q_2, \ldots, Q_n\}$. $L(Atm) = \{x \mid x \in \sum^*, \delta(Q_i, x) \in Q'\}$ is called a language recognized/accepted by *Atm*.

Based on Definition 3.2, we can easily deduce the following conclusion.

Lemma 3.3. *For $Atm = \{Q, \Sigma, \delta, Q_0, F\}$, if for all $x \in \Sigma^*$ and $\delta(Q_0, x) \in F$, then*

$$MLCS = \{x \mid x \in \Sigma^*, \delta(Q_0, x) \in F, |x| = \max(|x|)\}. \tag{3.3}$$

That is, MLCS is the set of the longest sequences recognized/accepted by $L(Atm)$.

With Definition 3.2, we can obtain the following properties.

Theorem 3.4. *For all $S \in MLCS$, let β be the corresponding matched pair of the ith ($i > 1$) character c_i and α be the corresponding matched pair of the $(i-1)$th character c_{i-1} in sequence S, then, β must be the direct successive matched pair of α. Furthermore, the first character of S must belong to the character of an initial matched pair.*

Proof. We give the proof by reduction in the following.

For all $S \in$ MLCS, assuming that β is not the direct successive matched pair of α, there will be a matched pair γ (γ corresponds to character c) such that $\alpha < \gamma < \beta$. So we can insert c between c_{i-1} and c_i to get a longer common subsequence, denoted as MLCS', which is contradiction to the fact that S is the longest common subsequence.

Then, let us consider the matched pair λ which corresponds to the first character of the S. If λ is not an initial matched pair, according to Definition 2.3, there must exist a matched pair ω, and $\omega < \lambda$. So we can get a longer MLCS' by inserting the character corresponding to ω into the header of the S which contradicts to the fact that S is the longest common subsequence. □

Theorem 3.5. *The Atm is a directed acyclic graph (DAG).*

Proof. The theorem will be proved by reduction.

Suppose that there was a series of states Q_i, Q_j, and Q_k forming a cycle in *Atm*, that is, state Q_i is the successive state of state Q_i, Q_k is the successive state of Q_j, and Q_i is the successive state of Q_k. Due to the fact that matched pairs p_i, p_j, and p_k correspond to the states Q_i, Q_j, and Q_k, we can get the results, $p_i < p_j$, $p_j < p_k$ and $p_k < p_i$, which contradicts to Definition 2.2. Therefore, *Atm* is a directed acyclic graph. □

Theorem 3.6. $\sum_{a \in \Sigma} \prod_{i=1}^{d} |S_i'|_a + 1$ *is an upper bound of the number of Atm's states, where $|S_i'|_a$ represents the occurrence number of a in sequence S_i'.*

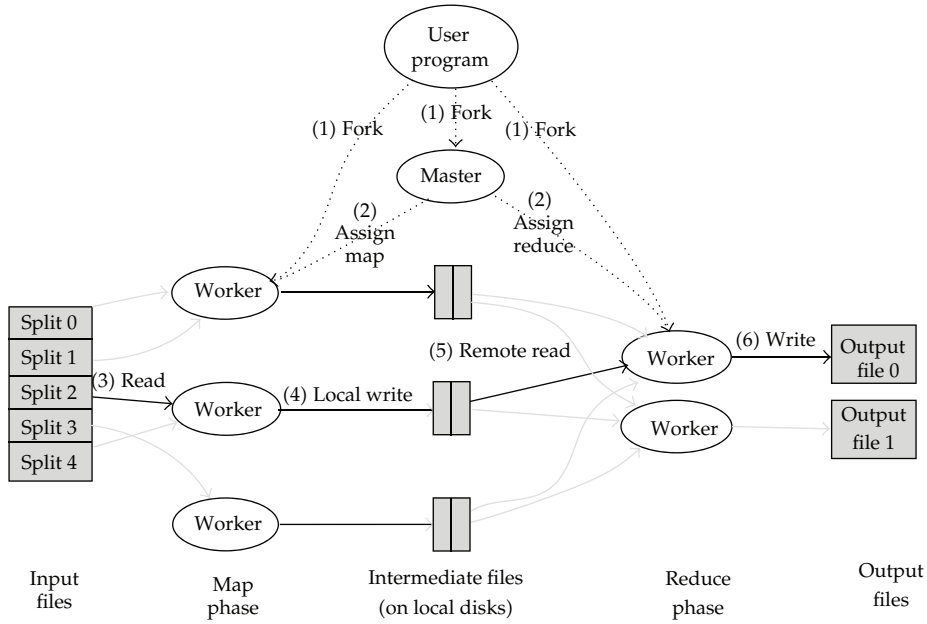

Figure 1: The execution framework of MapReduce.

Proof. Because the number of times the character a appears in the sequence S_i' is less than or equal to $|S_i'|_a$, $\prod_{i=1}^{d}|S_i'|_a$ is an upper bound for the number of the Atm's states for the character a. Thus, $\sum_{a\in\Sigma}\prod_{i=1}^{d}|S_i'|_a$ is an upper bound for all elements in Σ. By considering the initial state of the Atm, we get an upper bound $\sum_{a\in\Sigma}\prod_{i=1}^{d}|S_i'|_a + 1$ of the number of the Atm's states. □

4. FACC Algorithm Based on MapReduce Parallel Framework of Cloud Computing

4.1. The Overview of the MapReduce Parallel Framework of Cloud Computing

For the convenience, we first briefly overview the MapReduce parallel framework of cloud computing.

Cloud computing is a new computing model [24], which is a development of distributed computing, parallel computing and grid computing. MapReduce is a parallel framework of cloud computing and an associated implementation for processing and generating large datasets. It is amenable to a broad variety of real-world tasks and has been widely recognized as a cost-effective high performance parallel computing model. In the model, based on "divide and conquer" technology, users only specify the computation in terms of a *map* and a *reduce* function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules intermachine communication to make efficient use of the network and disks [25, 26]. Figure 1 shows the execution framework of MapReduce [26].

4.2. The Proposed Algorithm: FACC

In this subsection, before describing the details of the proposed FACC, we first give its framework as follows.

(1) *Preprocessing.* Determine the common alphabet Σ of sequence set T, and preprocess every sequence of T as follows. The redundant characters in each sequence are filtered, that is, removing the characters which do not appear in at least one sequence in T. This process will ensure a quick searching later for MLCS of T.

(2) *Construction of successor tables.* Based on the MapReduce parallel framework of cloud computing, successor tables (see Section 4.2 Definition 4.1) for every preprocessed sequence of T are parallel constructed. Since any character in the MLCS should be a character corresponding to a matched pair, we can construct the successor table for each sequence so that all possible matched pairs of T can be found quickly.

(3) *Construction of the finite automaton Atm for recognizing/accepting common subsequence.* Based on the MapReduce parallel framework, and using the matched pairs as the states and adding an initial state to the *Atm*, we can construct the *Atm* which can recognize/accept all the CS of sequences of the T according to transition function of the *Atm*, wherein each state holding all of the states of its parent nodes during the construction of the *Atm*.

(4) *Traversal of the Atm and output all of the MLCS*. Search for the MLCS by traversing the *Atm* through the depth-first method.

In the following, the proposed FACC and its implementation based on MapReduce will be described step by step in detail.

4.2.1. Preprocessing

Recall that the MLCS of T should be the sequences over their common alphabet Σ. The goal of the preprocessing is to reduce the searching time by filtering the redundant characters in each sequence which does not appear in Σ. After preprocessing, we will obtain the specific sequences which only reserve the characters in Σ. Since preprocessing leads to some time and space cost, the proposed FACC adopts this procedure only in the situations of the large or unknown alphabet Σ.

The idea of the preprocessing is that a *Key-Value* table is designed (that is a data structure Map), where the *Key* represents a character α and the *Value* is the total number of the sequences containing character α. For N sequences, the *Value* corresponding to *Key* α in the Map equals k if and only if k sequences contain the character α. Obviously, $k \leq N$. In this situation, we call the value of *Key* α is k. According to the definition of the value k of *Key* α, we can see that all characters with value N consist of the alphabet Σ. Then, all sequences are filtered in parallel using the MapReduce of cloud computing and only the characters in Σ are reserved. The ith resulted sequence obtained from S_i after the filter is denoted as $S_i'$for $i = 1 \sim N$.

Algorithm 1 shows the pseudocode of the preprocessing algorithm.

For example, sequences $S_1 = abecadbca$ and $S_2 = acbafcgba$, $\Sigma_1 = \{a, b, c, d, e\}$ and $\Sigma_2 = \{a, b, c, f, g\}$ are the alphabets of these two sequences, respectively. After preprocessing

```
Algorithm Pre-processing (InitStringSet)
Input: InitStringSet: a initial string set.
Output: Σ :common alphabet over InitStringSet
        StringSet: the result of pre-processed InitStringSet
(1)  for each string S_i in InitStringSet
(2)     tempSet = ∅
(3)     for each character S_i[j] in S_i
(4)        if (S_i[j] ∉ tempSet)
(5)           tempSet = tempSet ∪ {S_i[j]}
(6)           if (S_i[j] ∈ map)
(7)              map (S_i[j]) = map (S_i[j]) + 1
(8)           else
(9)              map (S_i[j]) = 1
(10)    endfor
(11) endfor
(12) Σ = ∅
(13) for each character char[i] in map
(14)    if (map(char[i]) == size (InitStringSet))
(15)          Σ = Σ ∪ {char[i]}
(16) endfor
(17) StringSet = InitStringSet
(18) for each string S_i in StringSet
(19)    for each character S_i[j] in S_i
(20)          if (S_i[j] ∉ Σ)
(21)             delete (S_i[j])
(22)    endfor
(23) endfor
(24) return Σ and StringSet
```

Algorithm 1: The pseudocode of the preprocessing algorithm.

to S_1 and S_2, we can get $\Sigma = \{a, b, c\}$, and S_1 is converted into $S_1' = abcabca$, and S_2 is converted into $S_2' = acbacba$.

4.2.2. Successor Table

Let Tab_k denote the successor table of the sequence S_k and its definition is as follows.

Definition 4.1. For a sequence $S_k = x_1, x_2, x_3, \ldots, x_n$ drawn from alphabet $\Sigma_k = (\sigma_1, \sigma_2, \ldots, \sigma_t)$, the successor table Tab_k of the sequence S_k is an irregular two-dimensional table, where the element of ith row and jth column of the table Tab_k is denoted as $\mathrm{Tab}_k[i, j]$, which is defined as follow.

$$\mathrm{Tab}_k[i, j] = \min\{p \mid x_p = \sigma_i, p \geq 1, p \geq j, p < n, 1 \leq i \leq |\Sigma_k|, 1 \leq j \leq n\}. \tag{4.1}$$

The value of $\mathrm{Tab}_k[i, j]$ indicates the minimal subscript position p of the sequence $S_k = x_1, x_2, x_3, \ldots, x_n$ according to σ_i after position j when $x_p = \sigma_i$.

For the two sequences $S_1' = abcabca$ and $S_2' = acbacba$, Tables 2 and 3 show the successor tables of them.

Table 2: Successor table Tab_1 of S'_1.

a	1	4	4	4	7	7	7
b	2	2	5	5	5		
c	3	3	3	6	6	6	

Table 3: Successor table Tab_2 of S'_2.

a	1	4	4	4	7	7	7
b	3	3	3	6	6	6	
c	2	2	5	5	5		

To construct successor tables for T, We dispatch $|T|$ Map functions to construct successor table for each sequence of T in parallel, and then employ a Reduce function to aggregate the successor tables of the sequences of T.

Because the irregular successor tables only store the useful information and are constructed in parallel, it can considerably reduce the time and space complexity of searching for the MLCS.

With the constructed successor table, a direct successive matched pair of a matched pair can be gotten quickly. Take Tab_1 and Tab_2 as examples. When searching for the successive matched pairs of matched pair $[i, j]$, all we need to do is searching for the matched pairs of the $[\text{Tab}_1(i), \text{Tab}_2(j)]$, (where $\text{Tab}_1(i)$ and $\text{Tab}_2(j)$ stand for all the elements of the ith and jth columns in the Tab_1 and Tab_2, respectively), and then, removes all of the matched pairs which are not direct successive matched pairs. For example, by checking Tab_1 and Tab_2 based on Definition 4.1, we can get the following successive matched pairs $[4,4]$, $[2,3]$, and $[3,2]$ of matched pair $[1,1]$. Due to $[2,3] < [4,4]$ and $[3,2] < [4,4]$, we remove indirect successive matched pair $[4,4]$, and finally, get the expected direct successive matched pairs $[2,3]$ and $[3,2]$ of the matched pair $[1,1]$.

4.2.3. Constructing the Common Subsequence Atm of T

By the aforementioned Definition 3.1 and its properties of the *Atm*, we can build the common subsequence *Atm* of the sequence set T in parallel by MapReduce. The algorithm Build-*Atm* is shown in Algorithm 2.

With the algorithm shown in Algorithm 2, take two sequences S'_1 and S'_2 for example, the main construction process of the *Atm* can be shown as follows (the process also applies to multiple sequences of T). (1) Construct a virtual initial state (0,0) corresponding to the matched pair [0,0] with character ε. (2) Determine all of the direct successive matched pairs of the matched pair [0,0]. (3) Use the deep first search (DFS) method to construct the *Atm*.

Notice that each state in the *Atm* must remember all of the states of its parent nodes during the construction of the *Atm*. For example, two sequences S'_1 and S'_2 illustrated in Figure 2 have a direct successive matched pair [1,1] of the matched pair [0,0]. Because matched pair [1,1] corresponds to character a, we can get a state transition $\delta((0,0), a) = (1,1)$. By the algorithm shown in Algorithm 2. The final common subsequence *Atm* constructed in the example of Figure 2 is shown in Figure 3, where the states (0,0) and (7,7) are the initial and final states of the *Atm* of the sequences S'_1 and S'_2, respectively.

Algorithm **Build-Atm**(*Pos*, *dsucSet*)
Input: Q_0: the initial state of the *Atm* $(0, 0, \ldots, 0)$
tabSet: the set of successor tables for T drawn from Σ
Output: *Atm*: a *Atm* of T for MLCS
(1) *Atm* = NULL, $Q = \{Q_0\}$
(2) Build-Atm(Q, *tabSet*)
(3) dispatch $|Q|$ map functions in parallel do
(4) for each $Q_i \in Q$, for $\forall a \in \Sigma$ dispatch a Map function
(5) $Q_j = \delta(Q_i, a)$
(6) if $Q_j \notin Q$ and $Q_j \neq \Phi$
(7) $Q = Q \cup Q_j$
(8) end for
(9) reduce all the results of the Map functions
(10) reduce all the results of the Map functions
(11) Build-Atm(Q, *tabSet*)
(12) return *Atm*
(13) end

Algorithm 2: The pseudocode for constructing *Atm* in parallel by MapReduce.

	1	2	3	4	5	6	7
S_1'	a	b	c	a	b	c	a
S_2'	a	c	b	a	c	b	a

Figure 2: Sequences S_1' and S_2'.

4.2.4. Traversing Atm and Finding the MCLS

By the finite automata theory and Definitions 3.1 and 3.2, we can get a character sequence, corresponding to a path from Q_0 to a state of the F of *Atm*, which is a candidate longest common subsequence of T, named LCS$'$. Hence, all the longest character sequences of all the candidate sequences are exactly the elements of MLCS from T. We first design a specific set named *resultSet* to store the expected MLCS. Then, By the depth first search method, the *Atm* is traversed from Q_0 to every state of the F in parallel by MapReduce schemes. Once we get a candidate LCS$'$, we make following operations: if *resultSet* is not empty and all string length of elements in *resultSet* is longer than that of the LCS$'$, ignore the LCS$'$; otherwise clear *resultSet*, and then insert LCS$'$ into the *resultSet*. Because *resultSet* is a set, it can eliminate redundant elements automatically, hence, we can acquire the MLCS from the *resultSet* eventually. In the example for the sequences S_1' and S_2' the all MLCSs are $\{abcba, ababa, acaca, acbca, acaba, abaca\}$.

5. The Time and Space Complexity Analysis of FACC

5.1. Time Complexity

In what follows, we first give the time complexity of the FACC in every stage, and then the total time complexity.

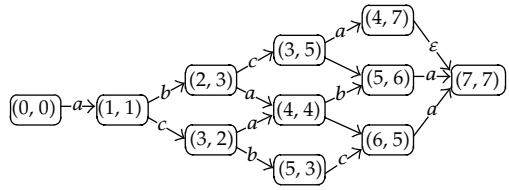

Figure 3: The common subsequence *Atm* of the sequences S_1' and S_2'.

Preprocessing

It is necessary to traverse every sequence of T once in order to find its common alphabet, therefore the step is $O(\sum_{i=1}^{d} |S_i|)$ time, where $|S_i|$ is the length of a sequence S_i and d is the number of total sequences of T, that is, $d = |T|$. It is also necessary to traverse every sequence of T to filter the characters not in Σ, which also requires $O(\sum_{i=1}^{d} |S_i|)$ time complexity. Assuming all the sequences in T with the same length n and the number of Map functions is d, the total time complexity is $O(n)$ in the stage based on MapReduce schemes.

Constructing the Successor Table

To build successor tables for all of the sequences pre-processed, it is necessary to traverse these sequences once again. It turns out that the same time complexity is required as the above preprocessing stage based on MapReduce schemes.

Constructing Atm

According to Theorem 3.6, it is known that the upper bound for the number of the *Atm*'s states is $\sum_{a\in\Sigma} \prod_{i=1}^{d} |S_i'|_a + 1$, where $|S_i'|_a$ stands for the number of times character a appears in the sequence S_i'. Because the time complexity for constructing the *Atm* is proportional to the number of the *Atm*'s states $|Q|$, the time complexity for constructing the *Atm* is $O(|Q|)$ in this stage based on MapReduce schemes.

Thus, the total time complexity of FACC is equal to $O(n) + O(n) + O(|Q|) = \max\{O(n), O(|Q|)\}$.

5.2. Space Complexity

Because the storage space of sequences and successor tables is static and proportional to the size of T, the space complexity of their storage is $O(\sum_{i=1}^{d} |S_i|)$. For building the *Atm*, the storage space is proportional to the number of states, hence the space complexity for building the *Atm* is $O(\sum_{a\in\Sigma} \prod_{i=1}^{d} |S_i'|_a)$. On the other hand, in recursively constructing the *Atm*, on the average, the recursion depth is $\log(\sum_{a\in\Sigma} \prod_{i=1}^{d} |S_i'|_a)$, and then the space complexity required temporary space is $O(\log(\sum_{a\in\Sigma} \prod_{i=1}^{d} |S_i'|_a))$ for constructing and traversing the *Atm*. It happens that the space complexity of FACC is

$$O\left(\sum_{i=1}^{d} |S_i|\right) + O\left(\sum_{a\in\Sigma} \prod_{i=1}^{d} |S_i'|_a\right) + O\left(\log\left(\sum_{a\in\Sigma} \prod_{i=1}^{d} |S_i'|_a\right)\right) + O\left(\log\left(\sum_{a\in\Sigma} \prod_{i=1}^{d} |S_i'|_a\right)\right)$$
$$= O\left(\sum_{i=1}^{d} |S_i| + \sum_{a\in\Sigma} \prod_{i=1}^{d} |S_i'|_a\right). \tag{5.1}$$

Table 4: Performance comparison between FAST-LCS and FACC on 20 DNA sequences ($|\Sigma| = 4$) with lengths from 25 to 180.

$\|S_i\|$	FAST-LCS algorithm				FACC algorithm				TPR
	AT (ms)	|MLCS|	N_{MLCS}	Precision	AT (ms)	|MLCS|	N_{MLCS}	Precision	
25	8	14	7	0.88	13	14	8	1.00	0.62
50	21	23	2	1.00	32	23	2	1.00	0.66
75	747	33	**8**	0.80	545	33	10	1.00	1.37
100	12952	42	2	1.00	6859	42	2	1.00	1.89
110	38341	50	4	1.00	19951	50	4	1.00	1.92
120	84738	54	9	1.00	39218	54	9	1.00	2.16
130	256214	59	**48**	0.98	81781	59	49	1.00	3.13
140	614583	64	**46**	0.96	164817	64	48	1.00	3.73
150	1188403	69	2	1.00	333625	69	2	1.00	3.56
160	2417387	76	4	1.00	650569	76	4	1.00	3.72
170	4828904	79	**10**	0.83	1268609	79	12	1.00	3.81
180	9811863	85	**2**	1.00	2473788	85	2	1.00	3.97

6. Experiments and Analysis of Experimental Results

6.1. Dataset and Experimental Results

In this paper, to test the time performance of FACC and FAST-LCS fairly, we run the two algorithms on the same hardware platform. Using the DNA and amino acid sample sequences dataset provided by ncbi [27] and dip [28], we tested the proposed algorithm FACC on the Hadoop cluster with 4 worker nodes, each of which contains 2 Intel CPUs (2.67 GHz) X5550, 8 GB of RAM, and 24 GB of local disk allocated to HDFS. In the cluster each node was running Hadoop version 0.20.0 on RedHat Enterprise Linux Server release 5.3 and connected by 100 M Ethernet to a commodity switch. The FAST-LCS algorithm was run on 4×2 Intel CPUs (2.67 GHz) X5550, and 8 GB of RAM, using the same datasets and operating system, and the programming environment of the algorithms is JDK 1.7. The comparisons of the time performance between FACC and FAST-LCS are shown in Tables 4 and 5 and Figures 4 and 5.

6.2. Discussion of Experimental Results

Tables 4 and 5 compares various performance indices of FAST-LCS and FACC on 20 DNA sequences ($|\Sigma| = 4$) and 20 amino acid sequences ($|\Sigma| = 20$) with different lengths of input sequences. It can be seen that various performance indices of FACC are superior to those of FAST-LCS, wherein the precision of FACC reaches 100% (shown in column *Precision*, which is the ratio of the number of found MLCSs N_{MLCS} to the total number of MLCSs, of Tables 4 and 5), while that of FAST-LCS reaches 95% due to its incorrect pruning operation 2. Moreover, Tables 4 and 5 and Figures 4 and 5 show that the time performance of FACC considerably outperforms that of FAST-LCS, and with the increasing lengths of input sequences, the advantage of FACC is growing significantly over FAST-LCS.

Figure 6 shows the time performance of the proposed FACC with preprocessing and without preprocessing. It can be seen from Figure 6 that the time performance for the case with preprocessing is obviously superior to that without preprocessing, especially for

Table 5: Performance comparison between FAST-LCS and FACC on the 20 amino acid sequences ($|\Sigma| = 20$) with lengths from 25 to 320.

$\|S_i\|$	FAST-LCS algorithm				FACC algorithm				TPR
	Time (ms)	|MLCS|	N_{MLCS}	Precision	T (ms)	|MLCS|	N_{MLCS}	Precision	
25	96	3	11	1.00	104	3	11	1.00	0.92
50	171	5	2	1.00	158	5	2	1.00	1.08
75	682	8	26	1.00	373	8	26	1.00	1.83
100	1177	14	1	1.00	601	14	1	1.00	1.96
120	2489	19	8	1.00	1210	19	8	1.00	2.06
140	4161	20	**109**	0.92	2006	20	119	1.00	2.07
160	14460	28	**190**	0.98	7192	28	194	1.00	2.01
180	32245	32	**62**	0.95	15807	32	65	1.00	2.04
200	69332	36	22	1.00	33803	36	22	1.00	2.05
220	140677	38	**534**	0.99	67889	38	539	1.00	2.07
240	269059	42	353	1.00	127409	42	353	1.00	2.11
260	497177	47	**637**	0.98	217605	47	650	1.00	2.28
280	1116905	49	**8689**	1.00	478707	49	8728	1.00	2.33
300	1617748	53	**454**	0.98	658212	53	462	1.00	2.46
320	2951500	60	**201**	0.96	1129275	60	210	1.00	2.61

Note: TPR indicates the ratio of the running time of FAST-LCS to that of FACC in Table 5.

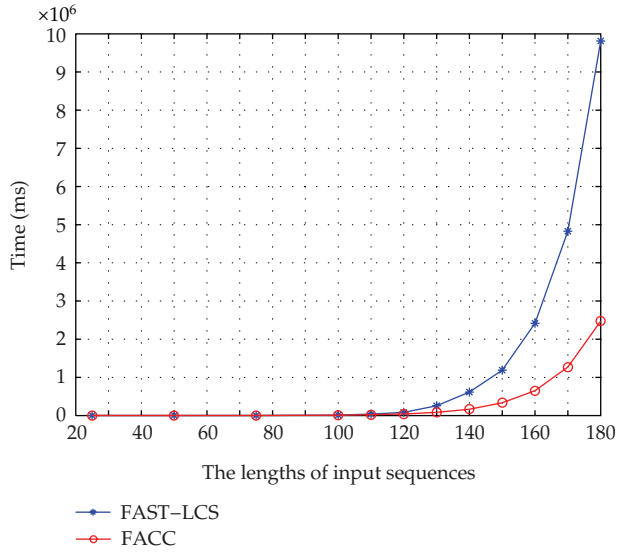

Figure 4: The time performance comparison between FACC and FAST-LCS on 20 DNA sequences ($|\Sigma| = 4$) with lengths from 25 to 180.

the cases of a large set of alphabet and a long average length of sequences. Furthermore, the more the number of input sequences, the more efficient the proposed FACC.

In summary, the time performance of the proposed algorithm FACC is much better than that of FAST-LCS.

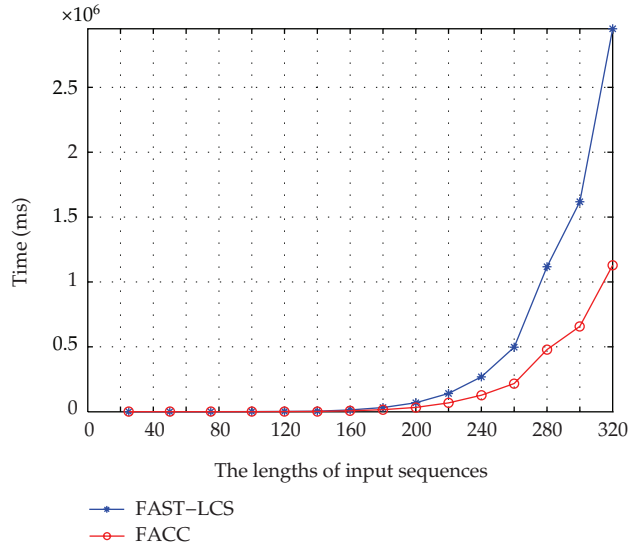

Figure 5: The time performance comparison between FACC and FAST-LCS on 20 amino acid sequences ($|\Sigma| = 20$) with lengths from 25 to 320.

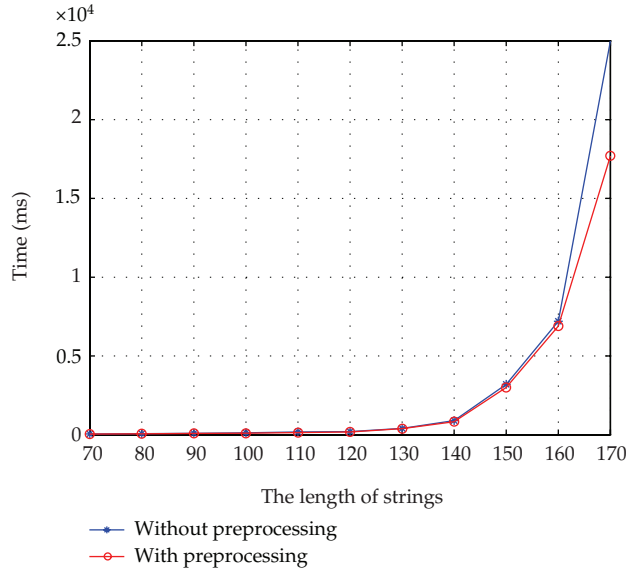

Figure 6: Performance of FACC with/without preprocessing on 20 amino acid sequences ($|\Sigma| = 20$) with lengths from 70 to 170.

7. Conclusions

Considering that the efficiency and effectiveness of the existing parallel algorithms for searching for MLCS are not satisfactory with the increasing complexity and size of biologic data and do not give an abstract and formal description of the MLCS problem and adopt complicated parallel schemes, we propose a novel finite automaton based on MapReduce

parallel framework of cloud computing called FACC to overcome the existing algorithms' shortcomings. The proposed algorithm is based on MapReduce parallel programming framework of cloud computing, the matched pair, and finite automaton (FA) by using some efficient techniques such as preprocessing, constructing the efficient successor table and common subsequence Atm, and looking for MLCS, and so forth. The theoretical analysis to the proposed algorithm shows that the time and space complexity are linear, that is, they are $\max\{O(n), O(|Q|)\}$ and $O(\sum_{i=1}^{d}|S_i| + \sum_{a\in\Sigma}\prod_{i=1}^{d}|S_i'|_a)$, respectively, which are superior to the leading parallel MLCS algorithms. Moreover, the simulation experiments of the proposed algorithm on some real DNA and amino acid sequence sample datasets are made, and their performance is compared with that of one of the current leading algorithms: FAST-LCS. The experimental results show that the proposed algorithm is very efficient and effective, and its performance is much better than that of FAST-LCS, especially for the cases of a large alphabet, a considerable number and a long average length of sequences. Meanwhile, experimental results also verify the correctness of our theoretical analysis.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (no. 61272119).

References

[1] W. S. Chen, P. C. Yuen, and X. Xie, "Kernel machine-based rank-lifting regularized discriminant analysis method for face recognition," *Neurocomputing*, vol. 74, no. 17, pp. 2953–2960, 2011.
[2] A. Cherkasov, "Bioinformatics: a practical guide to the analysis of genes and proteins," *American Journal of Human Biology*, vol. 17, no. 3, pp. 387–389, 2005.
[3] E. W. Edmiston, N. G. Core, J. H. Saltz, and R. M. Smith, "Parallel processing of biological sequence comparison algorithms," *International Journal of Parallel Programming*, vol. 17, no. 3, pp. 259–275, 1988.
[4] E. Lander, "Protein sequence comparison on a data parallel computer," in *Proceedings of the International Conference on Parallel Processing (ICPP '88)*, pp. 257–263, The Pennsylvania State University, University Park, PA, USA, 1988.
[5] A. Galper and D. L. Brutlag, "Parallel similarity search and alignment with the dynamic programming method," Tech. Rep., Stanford University, Palo Alto, Calif, USA, 1990.
[6] D. Maier, "The complexity of some problems on subsequences and supersequences," *Journal of the ACM*, vol. 25, no. 2, pp. 322–336, 1978.
[7] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974.
[8] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, vol. 18, no. 6, pp. 341–343, 1975.
[9] J. D. Ullman, A. V. Aho, and D. S. Hirschberg, "Bounds on the complexity of the longest common subsequence problem," *Journal of the ACM*, vol. 23, no. 1, pp. 1–12, 1976.
[10] J. W. Hunt and T. G. Szymansi, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.
[11] S. Bespamyatnikh and M. Segal, "Enumerating longest increasing subsequences and patience sorting," *Information Processing Letters*, vol. 76, no. 1-2, pp. 7–11, 2000.
[12] W. J. Masek and M. S. Paterson, "A faster algorithm computing string edit distances," *Journal of Computer and System Sciences*, vol. 20, no. 1, pp. 18–31, 1980.
[13] A. Aggarwal and J. Park, "Notes on searching in multidimensional monotone arrays," in *Proceedings of the 29th Annual Symposium on Foundations of Computer Science*, pp. 497–512, White Plains, NY, USA, 1988.
[14] A. Apostolico, M. J. Atallah, L. L. Larmore, and S. McFaddin, "Efficient parallel algorithms for string editing and related problems," *SIAM Journal on Computing*, vol. 19, no. 5, pp. 968–988, 1990.
[15] 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.

[16] W. Liu and L. Chen, "A fast longest common subsequence algorithm for biosequences alignment," *IFIP International Federation for Information Processing*, vol. 258, pp. 61–69, 2008.
[17] D. Korkin, Q. Wang, and Y. Shang, "An efficient parallel algorithm for the multiple longest common subsequence (MLCS) problem," in *Proceedings of the 37th International Conference on Parallel Processing (ICPP '08)*, pp. 354–363, Portland, Ore, USA, September 2008.
[18] Q. Wang, D. Korkin, and Y. Shang, "Efficient dominant point algorithms for the multiple longest common subsequence (MLCS) problem," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI '09)*, pp. 1494–1499, July 2009.
[19] Q. Wang, D. Korkin, and Y. Shang, "A fast multiple longest common subsequence (MLCS) algorithm," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 3, pp. 321–334, 2011.
[20] J. Yang, Y. Xu, and Y. Shang, "An efficient parallel algorithm for longest common subsequence problem on GPUs," in *Proceedings of the World Congress on Engineering (WCE '10)*, vol. 1, pp. 499–504, July 2010.
[21] M. L. Fredman, "On computing the length of longest increasing subsequences," *Discrete Mathematics*, vol. 11, no. 1, pp. 29–35, 1975.
[22] I. H. Yang, C. P. Huang, and K. M. Chao, "A fast algorithm for computing a longest common increasing subsequence," *Information Processing Letters*, vol. 93, no. 5, pp. 249–253, 2005.
[23] G. S. Brodal, K. Kaligosi, I. Katriel, and M. Kutz, "Faster algorithms for computing longest common increasing subsequences," in *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM '06)*, vol. 4900, pp. 330–341, 2006.
[24] M. Michael, *Cloud Computing: Web-Based Applications That Change the Way You Work and Collaborate Online*, SAMS Press, 2009.
[25] J. Dean, "Experiences with MapReduce, an abstraction for large-scale computation," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT '06)*, p. 1, IEEE Press, September 2006.
[26] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 15, no. 1, pp. 107–113, 2008.
[27] Pseudomonas aeruginosa PAO1 chromosome, complete genome, http://www.ncbi.nlm.nih.gov/nuccore/110645304?report=fasta.
[28] http://dip.doe-mbi.ucla.edu/dip/Download.cgi.