BMC Bioinformatics

BioMed Central

Research

Open Access

A fast parallel algorithm for finding the longest common sequence of multiple biosequences

Yixin Chen*[1], Andrew Wan[1] and Wei Liu[2]

Address: [1]Department of Computer Science and Engineering, Washington University in St. Louis, St. Louis, MO 63130, USA and [2]Department of Computer Science, Yangzhou University, Yangzhou 225009, China

Email: Yixin Chen* - chen@cse.wustl.edu; Andrew Wan - awan@wustl.edu; Wei Liu - yzliuwei@126.com

* Corresponding author

from Symposium of Computations in Bioinformatics and Bioscience (SCBB06) in conjunction with the International Multi-Symposiums on Computer and Computational Sciences 2006 (IMSCCS|06)
Hangzhou, China. June 20–24, 2006

Published: 12 December 2006

BMC Bioinformatics 2006, **7**(Suppl 4):S4 doi:10.1186/1471-2105-7-S4-S4

Abstract

Background: Searching for the longest common sequence (LCS) of multiple biosequences is one of the most fundamental tasks in bioinformatics. In this paper, we present a parallel algorithm named FAST_LCS to speedup the computation for finding LCS.

Results: A fast parallel algorithm for LCS is presented. The algorithm first constructs a novel successor table to obtain all the identical pairs and their levels. It then obtains the LCS by tracing back from the identical character pairs at the last level. Effective pruning techniques are developed to significantly reduce the computational complexity. Experimental results on gene sequences in the *tigr* database show that our algorithm is optimal and much more efficient than other leading LCS algorithms.

Conclusion: We have developed one of the fastest parallel LCS algorithms on an MPP parallel computing model. For two sequences X and Y with lengths n and m, respectively, the memory required is max{4*(n+1)+4*(m+1), L}, where L is the number of identical character pairs. The time complexity is O(L) for sequential execution, and $O(|\mathrm{LCS}(X, Y)|)$ for parallel execution, where $|\mathrm{LCS}(X, Y)|$ is the length of the LCS of X and Y. For n sequences $X_1, X_2, ..., X_n$, the time complexity is O(L) for sequential execution, and $O(|\mathrm{LCS}(X_1, X_2, ..., X_n)|)$ for parallel execution. Experimental results support our analysis by showing significant improvement of the proposed method over other leading LCS algorithms.

Background

Biological sequence [1] can be represented as a sequence of symbols. For instance, a protein [2] is a sequence of 20 different letters (amino acids), and DNA sequences (genes) can be represented as sequences of four letters A, C, G and T corresponding to the four sub-molecules forming DNA. When a new biosequence is found, we want to know what other sequences it is most similar to. Sequence comparison [3-5] has been used successfully to establish the link between cancer-causing genes and a gene evolved in normal growth and development. One way of detecting the similarity of two or more sequences is to find their LCS.

The LCS problem is to find a substring that is common to two or more given strings and is the longest one of such strings. Since the LCS problem is essentially a special case of the global sequence alignment, all the algorithms for the sequence alignment can be used to solve the LCS problem. Presented in 1981, the Smith-Waterman algorithm [6] is a well known LCS algorithm which was evolved from the Needleman-Wunsch [7] algorithm, and can guarantee the correctness. Aho and et al. [8] gave a lower bound of $O(mn)$ on time for the LCS problem using a decision tree model. It is shown in [9] that the problem can be solved in $O(mn)$ time using $O(mn)$ space by dynamic programming. Mayers and Miller [10] used the technique proposed by Hirschberg [11] to reduce the space complexity to $O(m+n)$ on the premise of the same time complexity.

To further reduce the computation time, some parallel algorithms [12-14] have been proposed for different computational models. For the CREW-PRAM model, Aggarwal [15] and Apostolico et al [16] independently proposed an $O(\log m \log n)$ time algorithm using $mn/\log m$ processors. Lu et al [17] designed two parallel LCS algorithms, one uses $mn/\log m$ processors with a time complexity of $O(\log^2 n+\log m)$, and the other uses $mn/(\log^2 m \operatorname{loglog} m)$ processors with a running time of $O(\log^2 m \operatorname{loglog} m)$. For the CRCW-PRAM model, Apostolico et al [16] gave an $O(\log n\ (\operatorname{loglog} m)^2)$ time algorithm using $mn/\text{loglogm}$ processors. Babu and Saxena [18] improved these algorithms for the CRCW-PRAM model. They designed an $O(\log m)$ algorithm with mn processors and an $O(\log^2 n)$ time parallel algorithm. Many parallel LCS algorithms have also been proposed using systolic arrays. Robert et al [19] proposed a parallel algorithm with $n + 5m$ steps using $m(m+1)$ processing elements. Chang et al [20] put forward an algorithm with $4n+2m$ steps using mn processing elements. Luce et al [21] designed a systolic array with $m(m+1)/2$ processing elements and $n+3m+q$ steps where q is the length of the LCS. Freschi and Bogliolo [22] addressed the problem of computing the LCS between run-length-encoded (RLE) strings. Their algorithm requires $O(m+n)$ steps on a systolic array of $M+N$ processing elements, where M and N are the lengths of the original strings and m and n are the number of runs in their RLE representation.

For the LCS problem of multiple sequences, the time complexity tends to grow very fast when the number of the sequences increases. For instance, using the Smith-Waterman algorithm to solve the LCS for multiple sequences, the time complexity is $O(2^n-1)\prod_{i=1}^{n} n_i$, where n is the number of sequences, and n_i is the length of the ith sequence. It is not practicable when n is large. Some improvements have been made on the algorithm. The MSA program [23] can process up to ten closely related sequences. It is an implementation of the Carrillo and Lipman algorithm [24] that identifies in advance the portions of the hyperspace not contributing to the solution and excludes them from the computation. Stoye described a new divide and conquer algorithm DCA [25] that extends the capability of MSA. Recently, OMA [26], an iterative implementation of DCA is proposed to speed up the DCA strategy and reduce memory requirements. Based on Feng and Doolittle's algorithm [28], Clustal-W [27] is one of the most widely used multiple sequence alignment software that can also be used for LCS computation.

Results

In this paper, we present a fast algorithm named FAST_LCS for efficient computation of LCS. The algorithm first seeks the successors of the initial identical character pairs according to a successor table to obtain all the identical pairs and their levels. Then by tracing back from the identical character pair in the last level, it obtains the result of LCS.

The key technique of our algorithm is the use of several effective **pruning operations**. In the process of generating the successors, pruning techniques can remove the identical pairs which can not generate the LCS so as to reduce the search space and accelerate the search speed. The algorithm can be extended to find the LCS of multiple biosequences.

Experimental results on the gene sequences of the *tigr* database, using an MPP parallel computer Shenteng 1800, show that our algorithm can obtain the exact optimal results and is much faster than some other leading LCS algorithms.

Conclusion

In this paper, we have developed FAST_LCS, one of the fastest parallel LCS algorithms on an MPP parallel computing model. For two sequences X and Y with lengths n and m, respectively, the memory complexity of FAST_LCS is $\max\{4*(n+1)+4*(m+1), L\}$, where L is the number of identical character pairs. The time complexity is $O(L)$ for sequential execution of the algorithm, and $O(|\text{LCS}(X,Y)|)$ for parallel execution, where $|\text{LCS}(X,Y)|$ is the length of the LCS of X and Y. The algorithm can be extended to solve the LCS for multiple biosequences. For n sequences

X_1, X_2, ..., X_n, the time complexity is O(L) for sequential execution, and $O(|\mathrm{LCS}(X_1, X_2, ..., X_n)|)$, which is independent of the number of sequences, for parallel execution. Experimental results support our analysis by showing significant improvement of the proposed method over some other leading LCS algorithms.

Methods

The identical character pair and its successor table

Let $X = (x_1, x_2, ..., x_n)$, $Y = (y_1, y_2, ..., y_m)$ be two biosequences, where $x_i, y_i \in \{A, C, G, T\}$. We can define an array CH of the four characters so that CH(1) = "A", CH(2)= "C", CH(3)= "G" and CH(4)= "T". To find the LCS of X and Y, we first build the successor tables of the identical characters for the two strings. The successor tables of the identical characters of X and Y are denoted as TX and TY, which are two dimensional arrays of size 4(n+1) × 4*(m+1). For sequence $X = (x_1, x_2, ..., x_n)$, $TX\ (i, j)$ in table TX is defined as follows.

$$TX(i,j) = \begin{cases} \min\{k \mid k \in SX(i,j)\} & SX(i,j) \neq \phi \\ - & \text{otherwise} \end{cases} \quad (1)$$

Here, $SX\ (i, j) = \{k \mid x_k = \mathrm{CH}(i), k > j\}$, where i = 1,2,3,4 and j = 0,1,...n. It can be seen from the definition that if $TX(i, j)$ is not "-", it indicates the position of the next character identical to CH(i) after the jth position in sequence X, otherwise it means there is no such character after the jth position.

Example 1

Let X = "T G C A T A" and Y = "A T C T G A T". Their successor tables TX and TY are shown in Table 1.

For the sequences X and Y, if $x_i = y_j = \mathrm{CH}(k)$, we call them an identical pair of CH(k) and denote it as (i, j). The set of all the identical character pairs of X and Y is denoted as $S(X, Y)$.

Table 1: Their successor tables *TX* and *TY* in Example 1.

TX:

i	CH(*i*)	0 1 2 3 4 5 6
1	**A**	4 4 4 4 6 6 -
2	**C**	3 3 3 - - - -
3	**G**	2 2 - - - - -
4	**T**	1 5 5 5 5 - -

TY:

i	CH(*i*)	0 1 2 3 4 5 6 7
1	**A**	1 6 6 6 6 6 - -
2	**C**	3 3 3 - - - - -
3	**G**	5 5 5 5 5 - - -
4	**T**	2 2 4 4 7 7 7 -

Let (i, j) and (k, l) be two identical character pairs of X and Y. If $i < k$ and $j < l$, we call (i,j) a predecessor of (k, l), or (k, l) a successor of (i, j), and denote the relationship as (i, j) < (k, l).

Let $P(i, j) = \{(r, s) \mid (i, j) < (r, s), (r, s) \in S(X,Y)\}$ be the set of all the successors of identical pair (i, j), if $(k, l) \in P(i, j)$ and there is no $(k', l') \in P(i, j)$ satisfying the condition: (k', l') < (k, l) and $x_{k'} = x_k$, we call (k, l) the direct successor of (i, j), and denote the relationship as (i, j)>(k, l).

If an identical pair $(i, j) \in S(X, Y)$ and there is no $(k, l) \in S(X, Y)$ satisfying (k, l) < (i, j), we call (i, j) an initial identical pair.

For an identical pair $(i, j) \in S\ (X, Y)$, its level is defined as follows:

$$level(i,j) = \begin{cases} 1 & \text{if } (i,j) \text{ is an initial identical character pair} \\ \max\{level(k,l)+1 \mid (k,l) < (i,j)\} & \text{otherwise} \end{cases} \quad (2)$$

From the definitions above, the following theorems can be easily deduced:

Theorem1. Denote the length of the LCS of X and Y as |LCS(X, Y)|, then $|\mathrm{LCS}(X, Y)| = \max\{\mathrm{level}\ (i, j) \mid (i, j) \in S(X, Y)\}$.

Proof: Suppose the identical character pairs corresponding to the longest common subsequence of X, Y are (x_{i1}, y_{j1}), (x_{i2}, y_{j2}), ..., (x_{ir}, y_{jr}), here $r = |\mathrm{LCS}(X, Y)|$. Since (i_1, j_1) is an initial identical character pair, we have: $(i_k, j_k) > (i_{k+1}, j_{k+1})$, for k = 1,2,...,r-1, and the level of (x_{ik}, y_{jk}) is k. Then we can conclude that the maximal level of all the identical character pairs is r, i.e. $r = \max\{\mathrm{level}(i, j) \mid (i, j) \in S(X, Y)\}$. The reason is as follows: if r is not the maximal level of the identical character pairs of X, Y, there must be an integer r' > r and identical character pairs: $(x_{i1'}, y_{j1'}) > (x_{i2'}, y_{j2'}) >$ .....$>(x_{ir}', y_{jr}')$. It corresponds to another common subsequence of X and Y with length $r' > r$. This is in contradiction with the condition $r = |\mathrm{LCS}(X, Y)|$.

The operation of producing successors

In our algorithm, all direct successors of all the initial identical character pairs are first produced in parallel using the successor tables. Then the direct successors of all those successors produced in the first step are generated in parallel. Repeat these operations of generating the direct successors until no more successors could be produced. Therefore, producing all the direct successors for the identical character pairs is a basic operation in our algorithm.

For an identical character pair $(i, j) \in S(X, Y)$, the operation of producing all its direct successors is as follows:

$(i,j) \rightarrow \{(TX(k,i),TY(k,j)) | k = 1,2,3,4,\ TX(k,i) \neq \text{'-'}$ and $TY(k,j) \neq \text{'-'}\}$ (3)

From (3) we can see that this operation is to couple the elements of the ith column of TX and the jth column of TY to get the pairs.

For instance, the successors of the identical character pair (2,5) in Example 1 can be obtained by coupling the elements of the 2nd column of TX and the 5th column of TY. They are (4, 6), (3, -), (-, -) and (5, 7). Here (3, -) and (-,-) do not represent identical character pairs, they only indicate the end of the process of searching for the successors in the branches they located. After discarding (3,-) and (-,-), the successors of (2, 5) are just (4, 6) and (5, 7).

Theorem 2. For an identical character pair (i, j), the method in (3) above can produce all its direct successors.

Proof: By (3), we can produce all direct successors $(TX(k, i),TY(k, j))$, where k = 1,2,3,4, of (i, j). According to (1),$TX(k, i)$ is the location of the nearest character identical to CH(k) after x_i in string X, and$TY(k, j)$ is the location of the nearest such character of CH(k) after y_j in string Y. This means that identical pairs $(TX(k, i),TY(k, j))$, where k = 1,2,3,4, contain all the direct successors of (i, j). Consequently, by the same operation on the newly generated identical pairs $(TX(k, i),TY(k, j))$, where k = 1,2,3,4, we can get all of their direct successors. It can be seen that by repeating this operation of producing successors, we can obtain all the successors of (i, j).

It is obvious that $(TX(k, 0),TY(k, 0))$, where k = 1,2,3,4, are all the initial identical pairs of X and Y. By Theorem 2, we know that starting from these initial identical pairs, all the other identical pairs of X, Y, and their levels can be produced.

The operations of pruning

In the process of generating the successors, pruning techniques can be implemented to remove the identical pairs which can not generate the LCS so as to reduce the search space and improve the efficiency.

Pruning Operation 1

If on the same level, there are two identical character pairs (i, j) and (k, l) satisfying $(k, l)>(i, j)$, then (k, l) can be pruned without affecting the correctness of the algorithm in obtaining the LCS of X and Y.

Rationale

The reason we can prune (k, l) is as follows. Suppose the two identical character pairs (i, j) and (k, l) are produced by the identical pairs (i_1, j_1) and (k_1, l_1) on the previous level. Let the LCS produced via (k_1, l_1) and (k, l) be $a_1a_2...a_ma_{m+1}...a_r$, here a_m corresponds to (k_1, l_1) and a_{m+1} corresponds to (k, l). Similarly, let the subsequence produced via (i_1, j_1) and (i, j) be $b_1b_2...b_mb_{m+1}...b_s...b_q$, here, b_m corresponds to (i_1,j_1) and b_{m+1} corresponds to (i, j). Since $(k, l)>(i, j)$, by Theorem 2 (k, l) must be produced after (i, j). Then there must exist b_s $(m+1 <s <q)$ corresponding to (k, l). Since $a_{m+1}...a_r$ and $b_sb_{s+1}...b_q$ are both the local longest common subsequences obtained by the operations of producing successors on (k, l), we have "$a_{m+1}...a_r$" = "$b_sb_{s+1}...b_q$" which means q-s = r-$(m+1)$, and $q = r$+s-(m+1). Since s>$m+1$, we have $q > r$. Therefore the subsequence "$a_ma_{m+1}...a_r$", which is produced on the mth level via (k_1, l_1), can not be included in the LCS of X and Y, and (k, l) can be pruned without affecting the algorithm to get the LCS of X and Y.

This pruning operation can be implemented to remove all those redundant identical pairs. After each level of identical pairs are generated, the algorithm checks all the newly generated identical pairs on the same level to find all such identical pairs (i, j) and (k, l) satisfying $(k, l) > (i, j)$ and then prune (k, l).

For instance, (4, 6) and (5, 7) in Example 1 are the successors of the identical pair (2, 5). Since they are on the same level and (4, 6)>(5, 7), we can prune (5, 7).

For another identical character pair (1,2) in Example 1, its successors are (4,6), (3,3), (2,5) and (5,2) which are obtained by coupling the 1st column of TX and the 2nd column of TY.

Since (3, 3) < (4, 6), (4, 6) can be omitted by pruning operation 1. On the next level, the successors of (3, 3), (2, 5) and (5, 2) are (4, 6), (5, 4), (5, 7) and (6, 6). Since identical pair (6, 6) is a successor of (5, 4), (5, 7) is a successor of (4, 6) and they are on the same level, (6, 6) and (5, 7) can be pruned.

Pruning Operation 2

If on the same level, there are two identical character pairs (i_1, j) and (i_2, j) satisfying $i_1 < i_2$, then (i_2, j) can be pruned without affecting the correctness of the algorithm in obtaining the LCS of X and Y.

Rationale

The reason we can prune (i_2, j) is as follows. Let the successors of (i_1, j) be $(l_2, j_2)>(l_3, j_3)>...>(l_r, j_r)$, then the length of common subsequence of "$x_{i1}x_{i1+1} ...x_n$" and "$y_jy_{j+1} ...y_m$" is just r. Let the successors of (i_2, j) are $(k_2, j_2')>(k_3,$

j_3')>...>(k_q, j_q'). Because i1 < i2, if an LCS contains a subsequence following (i2, j), this exact subsequence(note that all the x index is larger than i2) can be added at the end of (i1, j). Since (i1, j) and (i2, j) are on the same level, there must exist an LCS containing (i1, j). In other words, for any LCS containing identical character pair (i2, j), there is at least a corresponding LCS containing identical character pair (i1, j). Thus (i_2, j) can be pruned without affecting the algorithm to get the LCS of *X* and *Y*.

By extending pruning operation 2, we can get the following pruning operation.

Pruning Operation 3

If there are identical character pairs (i$_1$, j), (i$_2$, j), ...,(i$_r$, j) and $i_1 < i_2 < ... < i_r$, then we can prune(i_2, j),...,(i_r, j).

Framework of FAST_LCS and complexity analysis

Based on the operations of generating the successors of the identical character pairs using successor tables and the pruning operations, we present a fast parallel LCS algorithm named FAST_LCS. The algorithm consists of two phases: searching for all the identical character pairs and tracing back to get the LCS. The first phase begins with the initial identical character pairs; then continuously searches for successors using the successor tables. In this phase, the pruning technology is implemented to discard those search branches that obviously can not obtain the optimum solution so as to reduce the search space and speed up the process of searching.

The framework of the FAST_LCS algorithm is as follows, where the phase of searching for all the identical character pairs consists of steps 1, 2, 3 and the phase of tracing back is in step 4.

Step 1. Build tables *TX* and*TY*;

Step 2. Find all the initial identical character pairs:(*TX*(*k*, 0),*TY*(*k*, 0)), where *k* = 1,2,3,4, and add the records of the initial identical pairs (*k*, *TX*(*k*, 0), *TY*(*k*, 0), 1, *Φ*, *active*), *k* = 1,2,3,4 to the table *pairs*.

Step 3. Repeat the following until there is no record in *active* state in table *pairs*.

Step 3.1 For all *active* identical pairs (*k*, *i*, *j*, *level*, *pred*, *active*) in *pairs* parallel-do

Step 3.1.1 Produce all the direct successors of (*k*, *i*, *j*, *level*, *pred*, *active*).

Step 3.1.2 For each identical character pair (*g*, *h*) in the direct successors set of (*k*, *i*, *j*, *level*, *pred*, *active*), a new record (*k*', *g*, *h*, *level*+1, *k*, *active*) is generated and inserted into the table *pairs*.

Step 3.1.3 Change the state of (*k*, *i*, *j*, *level*, *pred*, *active*) into *inactive*.

Step 3.2 Use the pruning operations on all the successors produced on this level to remove all the redundant identical pairs from table *pairs*.

Step 4. Compute *r* = the maximum level in the table *pairs*.

For all the identical pairs (*k*, *i*, *j*, *r*,*l*,*inactive*) in *pairs* parallel-do

Step 4.1. *pred* = *l*; *LCS*(*r*) = x_i.

Step 4.2 While *pred* ≠ *Φ* do

Step 4.2.1 get the *pred*-th record (*pred*, *g*, *h*, *r*',*l*', *inactive*) from table *pairs*.

Step 4.2.2 *pred* = *l*'; *LCS*(*r*') = x_g.

In the algorithm, a table called *pairs* is used to store the identical character pairs obtained in the algorithm. In the table *pairs*, each record takes the form of (*k*, *i*, *j*, *level*, *pred*, *state*) where the data denote the index of the record, the identical character pair (*i*, *j*), its level, the index of its direct predecessor, and its current state, respectively. Each record in *pairs* has two states. For the identical pairs whose successors have not been searched, it is in the *active* state, otherwise it is in the i*nactive* state. In every step of search process, the algorithm searches for the successors of all the identical pairs in *active* state in parallel, and repeat this search process until there is no more identical pairs in *active* state in the table. The phase of tracing back starts from the identical pairs with the maximum level in the table, and traces back according to the *pred* of each identical pair. This tracing back process ends when it reaches an initial identical pair, and the trail indicates the LCS. If there are more than one identical pairs with the maximum level in the table, the tracing back procedure for those identical pairs can be carried out in parallel and several LCS can be obtained concurrently.

The LCS of *X*, *Y* is stored in the array *LCS*. In our algorithm, every identical pair must have the operation of producing successors at least once. Because of the pruning technology, this operation will only be run exactly once on each identical character pair. Therefore, assuming that the number of the identical character pairs of *X*, *Y* is *L*, the time complexity for a sequential execution of the algorithm FAST_LCS (*X*, *Y*) is *O*(*L*). Since the table *pairs* has to store all the identical character pairs, it requires *O*(*L*) memory space. Considering that the space cost of *TX*, *TY* are $4*(n+1)$ and $4*(m+1)$, the storage complexity of our algorithm is $\max\{4*(n+1)+4*(m+1), L\}$. In parallel implementation of the algorithm, since the process for each identical pair can be assigned to one processor, all the operations on the identical pairs can be carried out in parallel. Thus, the processing of each level requires $O(1)$ time, and the number of time steps required for a parallel execution of FAST_LCS is equal to the maximum level of the identical pairs. By Theorem 1, we know that the length of the LCS of *X*, *Y*, $|\mathrm{LCS}\ (X, Y)|$, is equal to the maximum level of the identical pairs. Therefore, the time complexity of parallel FAST_LCS is $O(|\mathrm{LCS}(X,Y)|)$.

Finding the LCS of multiple sequences using FAST_LCS

Our algorithm FAST_LCS can be easily extended to the LCS problem of multiple sequences. Suppose there are *n* sequences $X_1, X_2, ..., X_n$, where $X = (x_{i1}, x_{i2}, ..., x_{i,ni})$, n_i is the length of X_i, $x_{ij} \in \{A, C, G, T\}$ where j = 1, 2, ..., n_i. To find their LCS, similar to the case of two sequences, the algorithm for multiple sequences first builds the successor tables for all the sequences. Denote the successor tables of $X_1, X_2, ..., X_n$ as $TX_1, TX_2, ..., TX_n$, respectively, where TX_s is a two-dimensional array of size $4*(n_s + 1$ [-del-n$_i$+1]) for the sequence $X_\mathrm{s} = (x_{s1}, x_{s2}, ..., x_{s,ns})$, $s = 1,2,...n$, and successor table TX_s of identical characters is defined as:

$$TXs(i,j) = \begin{cases} \min\{k \mid k \in SXs(i,j)\} & SXs(i,j) \neq \phi \\ - & \text{otherwise} \end{cases} \qquad (4)$$

where $SX_s\ (i, j) = \{k|x_{sk} = \mathrm{CH}(i), k>j\}$ where i = 1, 2, 3, 4.

Similar to identical character pairs in the case of two sequences, we define the concept of ***identical character tuple*** for LCS of multiple sequences. For the sequences $X_1, X_2, ..., X_n$, if $x_{1,i_1} = x_{2,i_2} = ... = x_{n,i_n}$,we record them in an identical character tuple of the sequences $X_1, X_2, ...,X_n$ and denote it as $(i_1, i_2, ..., i_n)$. The set of all the identical character tuples of $X_1, X_2, ...,X_n$ is denoted as $S(X_1, X_2, ..., X_n)$.

Let $(i_1, i_2, ..., i_n)$ and $(j_1, j_2, ...,j_n)$ be two identical character tuples of $X_1, X_2, ..., X_n$. If $i_k < j_k$, for $k = 1,2,...n$, we call $(i_1, i_2, ..., i_n)$ a predecessor of $(j_1, j_2, ..., j_n)$, or $(j_1, j_2, ..., j_n)$ a successor of $(i_1, i_2, ..., i_n)$, and denote them as $(i_1, i_2, ..., i_n) < (j_1, j_2, ..., j_n)$.

Let $P(i_1, i_2, ..., i_n) = \{(j_1, j_2, ..., j_n) \mid (i_1, i_2, ..., i_n) < (j_1, j_2, ..., j_n), (j_1, j_2, ..., j_n) \in S(X_1, X_2, ..., X_n)\}$ be the set of all the successors of identical tuple $(i_1, i_2, ..., i_n)$, if $(k_1, k_2, ...,k_n)\Sigma P\ (i_1, i_2, ..., i_n)$ and there is no $(k_1', k_2', ..., k_n') \in P(i_1, i_2,..., i_n)$ satisfying the condition: $(k_1', k_2', ..., k_n') < (k_1, k_2, ..., k_n)$ and $x_{k_1'} = x_{k_1}$, we call $(k_1, k_2, ..., k_n)$ the direct successor of $(i_1, i_2, ..., i_n)$, and denoted it as $(i_1, i_2, ..., i_n)>(k_1, k_2, ..., k_n)$.

If an identical tuple $(i_1, i_2, ..., i_n) \in S(X_1, X_2, ..., X_n)$ and there is no $(k_1, k_2, ..., k_n) \in S(X_1, X_2, ..., X_n)$ satisfying $(k_1, k_2, ..., k_n)>(i_1, i_2, ..., i_n)$, we call $(i_1, i_2, ..., i_n)$ an initial identical tuple.

For an identical tuple $(i_1, i_2, ..., i_n) \in S(X_1, X_2, ..., X_n)$, its level is defined as follows:

$$level(i_1,i_2,...,i_n) = \begin{cases} 1 & \text{if } (i_1,i_2,...,i_n) \text{ is an initial identical tuple} \\ \max\{level(k_1,k_2,...,k_n)+1 \mid (k_1,k_2,...,k_n) \prec (i_1,i_2,...,i_n)\} & \text{otherwise} \end{cases} \qquad (5)$$

Similar to the case of two sequences LCS, the following theorems can be easily deduced.

Theorem 3. Denote the length of the LCS of $X_1, X_2,..., X_n$ as $|\mathrm{LCS}(X_1, X_2,...,X_n)|$, then

$|\mathrm{LCS}(X_1,X_2,...,X_n)| = \max \{level(i_1,i_2,...i_n) | (i_1,i_2,...i_n) \in S(X_1,X_2,...,X_n)\}$.

Proof of Theorem 3 is similar to that of Theorem 1.

In our parallel algorithm for LCS of multiple sequences, all direct successors of all the initial identical character tuples are produced in parallel using the successor tables. Then, the direct successors of all those successors produced in the first step are generated in parallel. Repeat these operations of generating the direct successors until no more successors could be produced. For an identical character tuple $(i_1, i_2, ..., i_n) \in S\ (X_1, X_2,..., X_n)$, this operation is as follows:

$(i_1, i_2,...,i_n) \rightarrow \{(TX_1(k, i_1), TX_2(k, i_2),...,TX_n(k, i_n)) | k = 1,2,3,4, TX_j(k, i_j) \neq \text{'-'}, j = 1,2,...,n\}$ (6)

From (6) we can see that this operation is to assemble the elements of the i_j-th column of TX_j, for $j = 1,2,...,n$ to get the new tuples.

Example 2. Let $n = 3$, and X_1 = "T G C A T A", X_2 = "A T C T G A T", and X_3 = "C T G A T T C". Their successor tables TX_1, TX_2 and TX_3 are shown in Table 2.

By assembling the 0th columns of the successor tables, we can get the initial identical triples (4,1,4), (3,3,1), (2,5,3) and (1,2,2).

The direct successors of the identical character triple (1,2,2) can be obtained by grouping the elements of the 1st column of TX_1 and the 2nd columns of TX_2 and TX_3, which are (4,6,4),(3,3,7) (2,5,3) and (5,4,5).

Table 2: Their successor tables TX_1, TX_2 and TX_3 in Example 2

TX_1:		
i	CH(*i*)	0 1 2 3 4 5 6
1	**A**	4 4 4 4 6 6 -
2	**C**	3 3 3 - - - -
3	**G**	2 2 - - - - -
4	**T**	1 5 5 5 5 - -
TX_2:		
i	CH(*i*)	0 1 2 3 4 5 6 7
1	**A**	1 6 6 6 6 6 - -
2	**C**	3 3 3 - - - - -
3	**G**	5 5 5 5 5 - - -
4	**T**	2 2 4 4 7 7 7 -
TX_3:		
i	CH(*i*)	0 1 2 3 4 5 6 7
1	**A**	4 4 4 4 - - - -
2	**C**	1 7 7 7 7 7 7 -
3	**G**	3 3 3 - - - - -
4	**T**	2 2 5 5 5 6 - -

Theorem 4. For an identical character tuple $(i_1, i_2, ..., i_n)$, the method in (6) can produce all its successors.

Proof of Theorem 4 is similar to that of Theorem 2.

From Theorem 4, we know that all the successors of the identical tuples on each level can be generated by the operation of producing successors. Starting from the initial identical tuples, all the identical tuples can be produced. In such process of generating the successors, pruning techniques can be implemented to remove the identical tuples which can not generate the longest common subsequence so as to reduce the search space. All the pruning operations for two-sequence LCS can be easily extended to the case of multiple sequences.

For instance, in Example 2, among the successors of triple (1,2,2), we have (2,5,3)>(4,6,4), since they are on the same level, we can prune (4,6,4). For another instance in Example 2, among the initial identical triples (4,1,4), (3,3,1), (2,5,3) and (1,2,2), since they are in the same level and (1,2,2)>(2,5,3) we can prune (2,5,3).

Assume that the number of the identical character tuples of the sequences $X_1, X_2, ..., X_n$ is L. In our algorithm, since every identical tuple must have the operation of producing successors exactly once, the time complexity for sequentially executing of the algorithm on the sequences $X_1, X_2, ..., X_n$ is $O(L)$. The algorithm uses a table *tuples* to store all the identical character tuples, it requires $O(L)$ memory space. Considering that the memory space cost of TX_j, for $j = 1,2,...,n$, is $4\sum_{i=1}^{n}(n_i+1)$, the storage complexity of our algorithm is $\max\{4\sum_{i=1}^{n}(n_i+1), L\}$. In a parallel implementation, since the computation for each identical tuple can be assigned on one processor, all the process on the identical tuples can be carried out in parallel. Therefore, the processing of each level requires $O(1)$ time, and the time required for the parallel computation is equal to the maximum level of the identical tuples which is equal to the length of the longest common subsequence of X_1, X_2, ..., X_n. Therefore the time complexity of the parallel execution of FAST_LCS on multiple sequences is $O(|\mathrm{LCS}(X_1, X_2, ..., X_n)|)$.

It should be pointed out that for most of the algorithms for multiple-sequence LCS, their time complexity strongly depends on the number of the sequences. For instance, if we use the Smith-Waterman algorithm to find the LCS of multiple sequences, the time complexity is $O(2^n-1)\prod_{i=1}^{n}n_i$, where n is the number of sequences, which is not practicable when n is large. The time complexity of our algorithm is $O(L)$ for sequential computation and O $(|\mathrm{LCS}(X_1, X_2, ..., X_n)|)$ for parallel implementation, where the time complexity of FAST_LCS is ***independent of the number of sequences n***. This means that our algorithm is much more efficient for finding the LCS of a large number of sequences.

Results

The results of sequential computation on two sequences

We test our algorithm FAST_LCS on the rice gene sequences of the *tigr* [29] database and compare the performance of FAST_LCS with that of Smith-Waterman algorithm [30] and FASTA algorithm [31,32] which are currently the most widely used LCS algorithms. Since both our algorithm and Smith-Waterman's can obtain exactly correct solution, we compare the computation speed of our algorithm FAST_LCS with that of Smith-Waterman algorithm. Also, we compare the precision of our algorithm with that of FASTA using the same computation time.

Table 3 compares the computation speed of FAST_LCS with that of Smith-Waterman algorithm on groups of gene sequence pairs with different lengths. Since a test on one pair of sequences takes very short time, it is hard to compare the speed of the algorithms using a single pair of sequence. Therefore we test the algorithms on groups of sequence pairs with similar lengths. We test five groups of sequence pairs each of which consisting of 100 pairs of sequences. The total time for each group by the two algorithms are listed in Table 3.

Fig. 1 shows the comparison of the computation time of our algorithm with that of Smith-Waterman algorithm. From the table and the figure, we can see that our algorithm is obviously faster than Smith-Waterman algorithm for sequence sets of all different lengths. The difference of the computation time between the two algorithms grows exponentially when the length of sequences becomes greater than 150. This means that our algorithm is much faster and more efficient than Smith-Waterman's for finding the LCS of long sequences.

Table 3: Comparison of computation speed of FAST_LCS with that of Smith-Waterman algorithm

			Time of FAST_LCS (s)		Time of S-W algorithm(s)	
Name of Sequences	Length *l*	Number of pairs	Total time	Average time	Total time	Average time
gi|21466196 ~ gi|21466195 ~ ... gi|21466168 ~ gi|21466167 ~ gi|21466166 ~ gi|30250556 ~ gi|30230255 ~ gi|30230254 ~ gi|30229613 ~ gi|30229612 ~ ... gi|30229449 ~ gi|30229448 ~	0≤*l*≤50	100	0.49	0.0049	1.09	0.0109
gi|30229047 ~ gi|30229046 ~ ... gi|30229001 ~ gi|30229000 ~ gi|30228999 ~ gi|30228998 ~ ... gi|30228849 ~ gi|30228848 ~	50≤ *l*≤ 100	100	5.88	0.0588	11.55	0.1155
gi|30229447 ~ gi|30229446 ~ ... gi|30229249 ~ gi|30229248 ~	10≤ *l*≤ 150	100	29.41	0.2941	65.95	0.6595
gi|30228846 ~ gi|30228845 ~ ... gi|30228648 ~ gi|30228647 ~	15≤ *l*≤ 200	100	94.11	0.9411	172.213	1.7213
gi|30229247 ~ gi|30229246 ~ ... gi|30229049 ~ gi|30229048 ~	20≤ *l*≤ 250	100	230.51	2.3051	425.16	4.2516

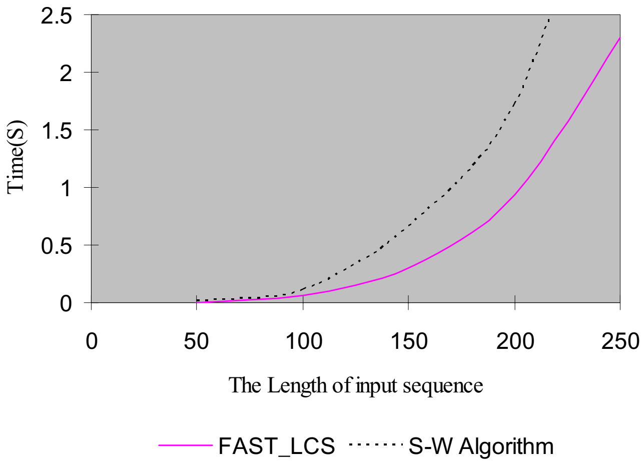

Figure 1
Comparison of the computation time of FAST_LCS with that of Smith – Waterman algorithm.

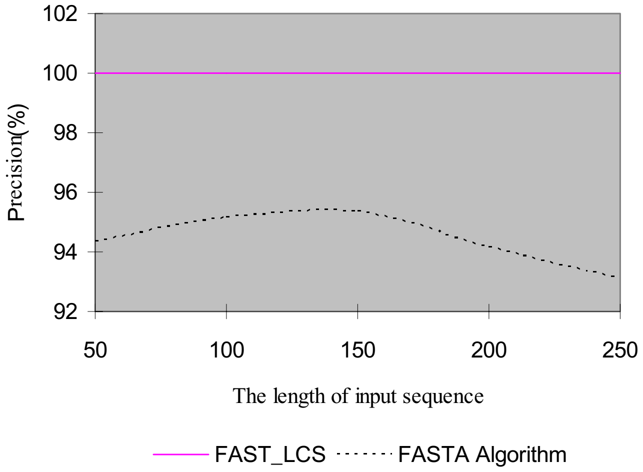

Figure 2
Comparison of the precision of FAST_LCS with that of FASTA using the same computation time.

We also compare the precision of our algorithm with that of FASTA on the premise of the same computing time. Here precision is defined as:

$$\text{Precision} = \frac{\text{The length of the common subsequence computed by the algorithm}}{\text{The length of the longest common subsequence in correct match}}$$

Fig. 2 shows the comparison of the precision of the results by our algorithm with that by FASTA using the same computation time. From Fig. 2, we can see that our algorithm can obtain the correct result no matter how long the sequence is, while the precision of FASTA declines when the length of the sequences is increased. Therefore the precision of our algorithm is much higher than that of FASTA.

The results of sequential computing on multiple sequences

We test our algorithm FAST_LCS on the multiple sequences and compare with the Clustal-W[27] algorithm which is a popular algorithm for multiple-sequence LCS. Fig. 3 and Fig. 4 show the comparison of the computation time of our algorithm FAST_LCS with that of Clustal-W algorithm. Table 4 lists the computation time of the two algorithms on 5 sets of different numbers of sequences with length of 50. Comparison of the computation time of the two algorithms on sequences sets of different numbers of sequences is shown in Fig. 3.

From Fig. 3 and Table 4, we can see that FAST_LCS is faster than Clustal-W for sets with different numbers of sequences. When the number of sequences is larger than five, the speed up is significant.

Table 5 lists the computation time of the two algorithms on five sequences sets with different lengths. Comparison of the computation time of the two algorithms on sequence sets of different lengths is shown in Fig. 4. From Table 5 and Fig. 4, we can see that FAST_LCS is faster than Clustal-W for sequence sets with different lengths.

We also compare the precision of our FAST_LCS algorithm with that of Clustal-W algorithm. Fig. 5 shows the comparison of precision of FAST_LCS with that of Clustal-W on the sets with different numbers of sequences, and Fig. 6 shows the comparison of precision of the two algorithms on the sets of sequences with different lengths. From these two figures, we can see that no matter how the length and the number of sequences are increased, our

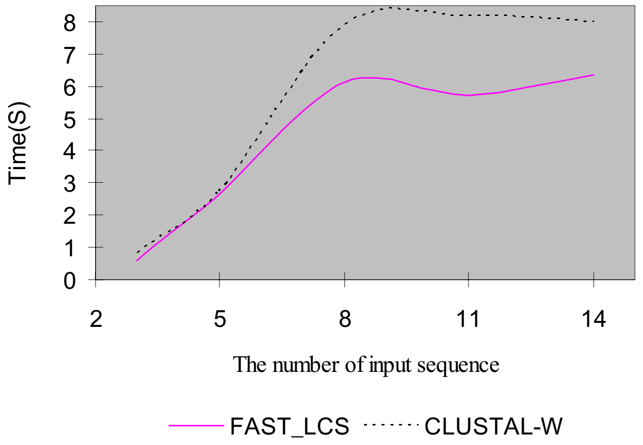

Figure 3
Comparison of the computation time of FAST_LCS with that of Clustal-W on sequence sets of different numbers of sequences.

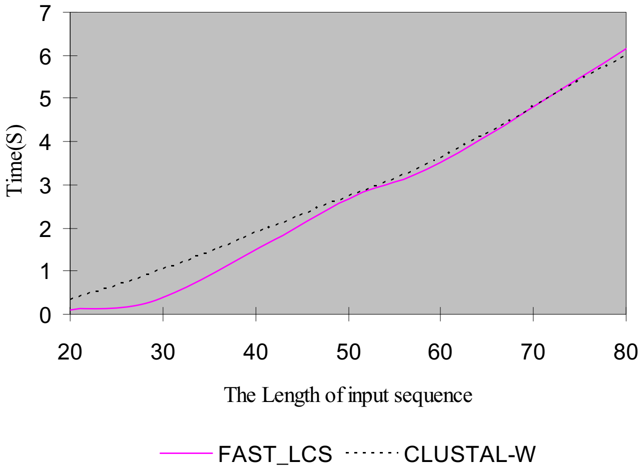

Figure 4
Comparison of the computation time of FAST_LCS with that of Clustal-W on sequence sets of different lengths.

algorithm can obtain the exactly correct results. The precision of Clustal-W declines when the number or the length of the sequences is increased. Therefore the precision of our algorithm is much higher than Clustal-W.

The results of parallel computing

We also test our algorithm on the rice gene sequence from the tigr database [29] on the Shenteng 1800 supercomputer using MPI (C bounding). In the parallel implementation of FAST_LCS, the identical character pairs in the active state are assigned and processed in different processors. The experimental results by using different numbers of processors are shown in Fig. 7. Three pairs of gene sequences are tested. The names, lengths, and computation times are listed in Table 6. From Fig. 7 and Table 6 we can see that the computation will become faster as the number of processors increases. Due to the communication overhead between processors, the speedup of our algorithm is slower than linear, which conforms to the Amdahl's Law.

Table 5: Comparison of computation time of FAST_LCS with that of Clustal-W on sequences sets of different lengths

	The length of input sequences				
Algorithm	20	30	50	60	80
Time of FAST_LCS (S)	0.109	0.391	2.656	3.516	6.166
Time of Clustal-W (S)	0.312	1.053	2.732	3.612	5.992

Acknowledgements

This work is supported in part by research funds from an Early Career Principal Investigator Award from the Department of Energy of the United States, the Washington University in St. Louis, the Chinese National Natural Science Foundation under grant No. 60673060, the Chinese National Foundation for Science and Technology Development under contract 2003BA614A-14, and the Natural Science Foundation of Jiangsu Province under contract BK2005047.

This article has been published as part of *BMC Bioinformatics* Volume 7, Supplement 4, 2006: Symposium of Computations in Bioinformatics and Bioscience (SCBB06). The full contents of the supplement are available online at http://www.biomedcentral.com/1471-2105/7?issue=S4.

Table 4: Comparison of computation time of FAST_LCS and that of Clustal-W on sequences sets of different numbers of sequences

Sequence name	Sequence Number	Time of FAST_LCS (s)	Time of Clustal-W (s)
gi|21466194 ... gi|21466196	3	0.609	0.804
gi|21466192 ... gi|21466196	5	2.656	2.732
gi|21466189 ... gi|21466196	8	6.14	7.91
gi|21466186 ... gi|21466196	11	5.71	8.20
gi|21466183 ... gi|21466196	14	6.34	8.49

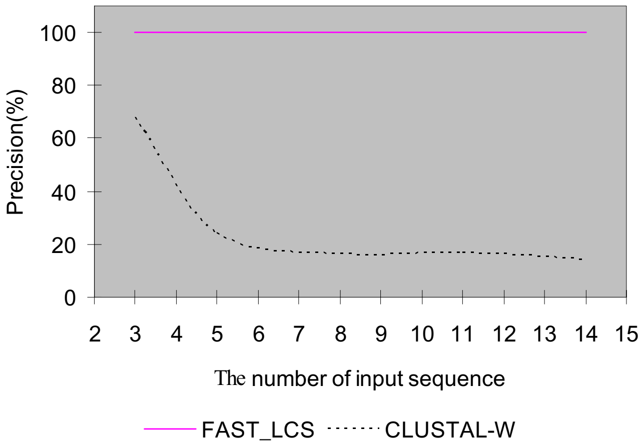

Figure 5
Comparison of the precision of FAST_LCS with that of Clustal-W on sequence sets with different numbers of sequences.

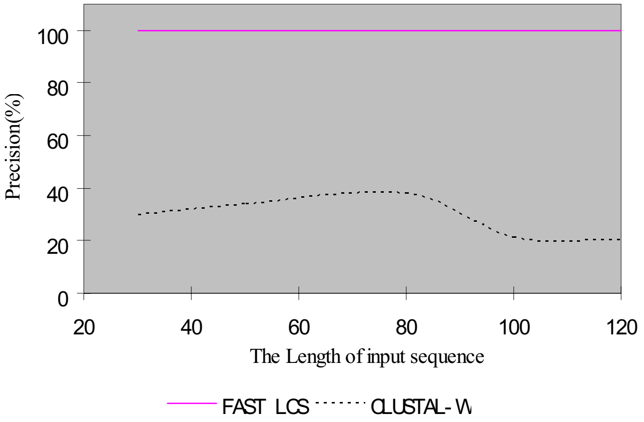

Figure 6
Comparison of the precision of FAST_LCS with that of Clustal-W on sequence sets of different lengths.

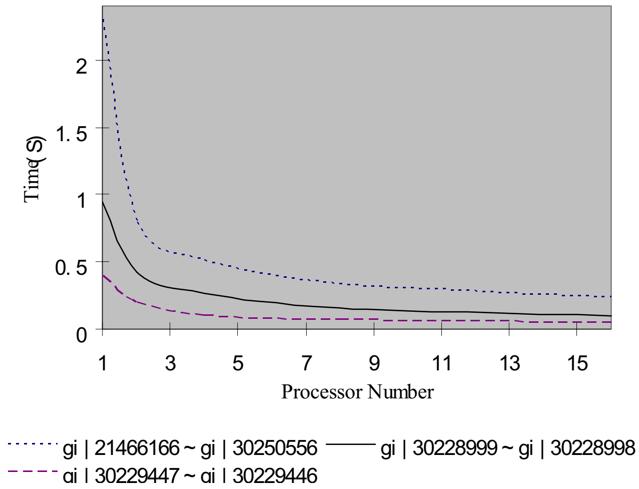

Figure 7
Parallel computational time of FAST_LCS using different processor numbers.

Table 6: Computational time of parallel FAST_LCS using different numbers of processors

		Computational time using different numbers of processors (s)				
Sequence name	Length	1	2	4	8	16
gi|21466166 ~ gi|30250556	250	2.3051	0.7955	0.51085	0.3300	0.2313
gi|30228999 ~ gi|30228998	200	0.9411	0.4247	0.2669	0.15421	0.09379
gi|30229447 ~ gi|30229446	150	0.3941	0.2015	0.10271	0.06975	0.04702

References

1. Hao B, Zhang SY: **The manual of bioinformatics.** *Shanghai Science and Technology Publishing Company* 2000.
2. Li YD, Sun ZR, *et al.*: **Bioinformatics – The practice guide for the analysis of gene and protein.** *Tsinghua University Publishing Company* 2000.
3. Edmiston EW, Core NG, Saltz JH, *et al.*: **Parallel processing of biological sequence comparison algorithms.** *International Journal of Parallel Programming* 1988, **17(3):**259-275.
4. Lander E: **Protein sequence comparison on a data parallel computer.** *Proceedings of the 1988 International Conference on Parallel Processing* 1988:257-263.
5. Galper AR, Brutlag DL: **Parallel similarity search and alignment with the dynamic programming method.** In *Technical Report California: Stanford University*; 1990.
6. Smith TF, Waterman MS: **Identification of common molecular subsequence.** *Journal of Molecular Biology* 1990, **215:**403-410.
7. Needleman SB, Wunsch CD: **A general method applicable to the search for similarities in the amino acid sequence of two proteins.** *J Mol Biol* 1970, **48(3):**443-453.
8. Aho A, Hirschberg D, Ullman J: **Bounds on the complexity of the longest common subsequence problem.** *J Assoc Comput Mach* 1976, **23:**1-12.
9. Gotoh O: **An improved algorithm for matching biological sequences.** *J Molec Biol* 1982, **162:**705-708.
10. Mayers EW, Miller W: **Optimal alignment in linear space.** *Comput Appl Biosci* 1998, **4:**11-17.
11. Hirschberg DS: **A Linear space algorithm for computing maximal Common Subsequences.** *Commun ACM* 1975, **18(6):**341-343.
12. Pan Y, Li K: **Linear array with a reconfigurable pipelined bus system – concepts and applications.** *Journal of Information Science* 1998, **106:**237-258.
13. Myoupo JF, David Seme D: **Time-efficient parallel algorithms for the longest common subsequence and related problems.** *Journal of Parallel and Distributed Computing* 1999, **57:**212-223.
14. Bergroth L, Hakonen H, Raita T: **A survey of longest common subsequence algorithms.** *Seventh International Symposium on String Processing Information Retrieval* 2000:39-48.
15. Aggarwal A, Park J: **Notes on searching in multidimensional monotone Arrays.** *Proc 29th Ann IEEE Symp Foundations of Comput Sci* 1988:497-512.
16. Apostolico A, Atallah M, Larmore L, Mcfaddin S: **Efficient parallel algorithms for string editing and related problems.** *SIAM J Computing* 1990, **19:**968-988.
17. Lu M, Lin H: **Parallel algorithms for the longest common subsequence Problem.** *IEEE Transaction on Parallel and Distributed System* 1994, **5:**835-848.
18. Babu KN, Systems W, Saxena S: **Parallel algorithms for the longest common subsequence problem.** *4th International Conference on High Performance Computing* 1997:18-21.
19. Robert Y, Tchuente M: **A Systolic Array for the Longest Common Subsequence Problem.** *Inform Process Lett* 1985, **21:**191-198.
20. Chang JH, Ibarra OH, Pallis MA: **Parallel parsing on a one-way array of finite-state machines.** *IEEE Trans Computers* 1987, **C-36:**64-75.
21. Luce G, Myoupo JF: **Systolic-based parallel architecture for the longest common subsequences problem.** *Integration* 1998, **25:**53-70.
22. Freschi V, Bogliolo A: **Longest common subsequence between run-length-encoded strings:a new algorithm with improved parallelism.** *Information Processing Letters* 2004, **90:**167-173.
23. Lipman DJ, Altschul SF, Kececioglu JD: **A tool for multiple sequence alignment.** *Proc Natl Acad Sci USA* 1989, **86:**4412-4415.
24. Carrillo H, Lipman DJ: **The multiple sequence alignment problem in biology.** *SIAM J Appl Math* 1988, **48:**1073-1082.
25. Stoye J, Moulton V, Dress AW: **DCA: an efficient implementation of the divide-andconquer approach to simultaneous multiple sequence alignment.** *Comput Appl Biosci* 1997, **13:**625-6.
26. Reinert K, Stoye J, Will T: **An iterative method for faster sum-of-pair multiple sequence alignment.** *Bioinformatics* 2000, **16(9):**808-814.
27. Thompson JD, Higgins DG, Gibson TJ: **CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position specific gap penalties and weight matrix choice.** *Nucleic Acids Research* 1994, **22:**4673-4680.
28. Feng DF, Doolittle RF: **Progressive sequence alignment as a prerequisite to correct phylogenetic trees.** *J Mol Evol* 1987, **25:**351-360.
29. [http://www.tigr.org/tdb/benchmark].
30. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ: **Basic local alignment search tool.** *J Mol Biol* 1990, **215:**403-410.
31. [http://alpha10.bioch.virginia.edu/fasta_www/cgi/].
32. [http://www.ebi.ac.uk/services/].