Parallel Computation of Longest-Common-Subsequence [†]

Mi Lu
Electrical Engineering Department
Texas A&M University
College Station, TX 77843

Abstract

A parallel algorithm for finding the longest common subsequence of two strings is presented. Our algorithm is executed on r processors, with r equal to the total number of pairs of positions at which two symbols match. Given two strings of length m and n respectively, $m \leq n$, with preprocessing allowed, our algorithm achieves $O(\log\rho\log^2 n)$ time complexity where ρ is the longest common subsequence. Fast computing of Longest-Common-Subsequence is made possible due to the exploiting of the parallelism.

I. Introduction

A *string* is a sequence of symbols. The recognition of two information-bearing strings that are related, even though not identical, is requested very often in visual pattern matching, speech recognition, editing error correction, bird song classification, artificial intelligence, data retrieval and genetics [1].

Given a string, a *subsequence* of the string can be obtained from the string by deleting none or some symbols (not necessarily consecutive ones). If string C is a subsequence of both string A, and string B, then C is a *common subsequence* (CS) of A and B. String C is a *longest common subsequence* (LCS) of string A and B if C is a common subsequence of A and B with maximal length. For example, if "*development*" and "*depend*" are the two input strings, then "*dee*" is a CS of the strings, and "*depen*" is a LCS of them. The LCS problem was first studied in molecular biology where similar sequences in the analysis of amino acids and nucleic acid involve hundreds of symbols [2]. For example, ARV-2DNA, the human retrovirus associated with AIDS disease, consists of about 10,000 symbols. Manual analysis is not promising due to the massive time needed and the limitation in the human recognition ability. A special application of the LCS problem is the pattern-matching problem, where a string A of m symbols is a substring of a string B of n symbols, $m \leq n$. In syntactic pattern recognition, a pattern is usually represented by a string, or a graph of pattern primitives. The processes on the strings are usually slow due to the large amount of data to be processed. Efficient algorithms need to be generated [3] for the pattern-matching recognition.

The string-to-string correction (or string-editing) problem is a generalization of the LCS problem. In the editing error analysis, one string is needed to be transformed/corrected to others, say, some given keywords. The "distance" of the edited string away from the given string can be a function of the cost of different operations such as deletion, insertion and substitution. Also, it can be a function of the possibilities of different characters to be mistyped/misspelled for another. For example, because of the conventional keyboard arrangement, it may be far more likely that a character "*j*" be mistyped as an "*h*" than as a "*r*". Thus preference should be given to correcting the word "*jail*" to "*hail*" rather than to "*rail*".

Hirschberg [4] solved the LCS problem in quadratic time and linear space first, and proved with others that the bounds on the complexity of LCS problem is $O(mn)$ for two strings with length m and n respectively if the comparisons of two symbols provides only "equal-not equal" information [5]. He later on developed two algorithms with time performance $O(\rho n + n\log n)$ and

[†] This research is partially supported by the National Science Foundation under Grant No. MIP-8809328.

$O(\rho(m+1-\rho)\log n)$ respectively, where ρ is the longest common subsequence. Hunt presented an algorithm for this problem which needs $O((r+n)\log n)$ running time and $O(r+n)$ space, where r is the total number of ordered pairs of positions at which the two sequences match [6]. In [7], Hsu and Du presented their new results for the LCS problem. With $O(n\log n)$ preprocessing time, their algorithms improved the time complexity to $O(\rho m\log(n/m)+\rho m)$ and $O(\rho m\log(n/\rho)+\rho m)$ respectively. They also gave a method in [8] to compute a longest common subsequence for a set of strings. All the previous algorithms are sequential. Although some parallel arrays are proposed as the special-purpose hardware to perform sequence comparison, or to be programmed to solve more general problems, very few parallel algorithms have been designed for solving the LCS problem on parallel machines [9].

Our algorithm is based on the divide-and-conquer strategy which allows concurrency and hence improved efficiency. Better time performance has been achieved by exploiting the parallelism in the current work. With preprocessing allowed, our algorithm is run on r processors of CREW PRAM, and needs $O(\log\rho\log^2 n)$ time, where r is the total number of pairs of positions at which two strings match, ρ is the longest common subsequence, and n is the length of the longer string out of the two.

In the next section, we give the preliminaries of the algorithm design. Our algorithm for finding LCS is presented in section III along with the analysis of the time performance of our solution. Section IV includes the discussion on preprocessing. Concluding remarks and related research problems are given in section V.

II. Preliminaries

Let string $A = a_1a_2\cdots a_i\cdots a_m$ and $B = b_1b_2\cdots b_j\cdots b_n$ be two input strings, and the length of the string $|A| = m$ and $|B| = n$ respectively. Assume $m \le n$ without loss of generality. Let $L(i,j)$ be the length of the LCS of $a_1a_2\cdots a_i$ and $b_1b_2\cdots b_j$, and $L(i,j)$ can be determined as follows [4]:

Compute L

```
1. $L(i,0) \leftarrow 0$ for $i = 0, 1, \cdots, m$; /* Initialization */
2. $L(0,j) \leftarrow 0$ for $j = 0, 1, \cdots, n$;
3. for $i = 1$ to $m$ do
   begin
4. for $j = 1$ to $n$ do
5. if $a_i = b_j$ then $L(i,j) \leftarrow L(i-1,j-1)+1$
6.      else $L(i,j) \leftarrow max\{L(i,j-1), L(i-1,j)\}$
   end
```

An $m \times n$ matrix, L, is constructed for computing $L(i,j)$ [10] such that the entry of row i and column j indicates $L(i,j)$. The $L(i,j)$'s are computed row by row with the above method and the length of LCS is given by $L(m,n)$ when the computation is completed. Figure 1 shows an L matrix of an example with $m = 6$ and $n = 9$. Further observation is made to reduce the entries to be considered in the L matrix [7]. Since only matched symbols will constitute an LCS, $L(i,j)$ is

							B				
			c	b	a	c	b	a	a	b	a
		0	1	2	3	4	5	6	7	8	9
	0	0	0	0	0	0	0	0	0	0	0
	a 1	0	0	0	1*	1	1	1*	1*	1	1*
	b 2	0	0	1*	1	1	2*	2	2	2*	2
A	c 3	0	1*	1	1	2*	2	2	2	2	2
	d 4	0	1	1	1	2	2	2	2	2	2
	b 5	0	1	2*	2	2	3*	3	3	3*	3
	b 6	0	1	2*	2	2	3*	3	3	4*	4

Figure 1

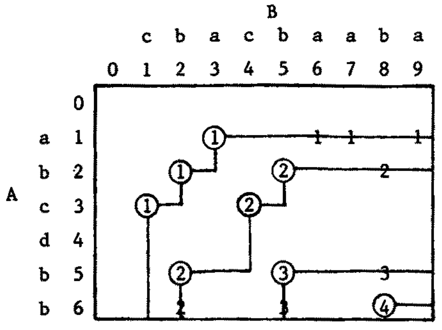

Figure 2

selected (indicated by "*" in Fig. 1) only if $a_i = b_j$.

The selected entries (i, j) are now referred to as "points", and the value of the corresponding $L(i, j)$ is called the *class* of that point, say p, denoted as $class(p)$. The left most point of a row in a class is denoted as the *break point* if it is the top point of its column in the class (see the circled ones in Figure 2). The horizontal and vertical lines connecting the points in the same class form the outline of the class. Point $q(i_q, j_q)$ is said to be dominated by point $p(i_p, j_p)$ *iff* $i_p < i_q$ and $j_p < j_q$. The definition of "dominate" is slightly different from the conventional one.

Instead of considering class 1, then class 2, class 3, $\cdots\cdots$, as is done sequentially in the previous results, our algorithm employs the divide-and-conquer strategy and runs faster. Assume that the total number of given points is r, then r processors will be operated under the PRAM model with each processor maintaining the record for one point.

Divide the given set of points by a horizontal line into two halves, the lower half and the upper half, such that the points in the lower half have i values greater than that of the points in the upper half. Recursively solve the problem for each half concurrently, and then merge the subresults. Before the merge, each PE containing a point is aware of the class of that point in its own half. (For simplicity, we hereafter say that the point, instead of the PE containing the point, is aware of certain information.) After the merge, the class of the points in the upper half will remain the same as before, but the points in the lower half need to recompute their classes.

Project the left most point of each class, say class k, in the upper half onto the horizontal dividing line. Denote the projection as *joint* (indicating by ϕ in Fig. 3), and index it by k. Locate a dummy joint at the left boundary of the plane with index 0.

For the example shown in Figure 3, the class outlines in the upper and lower halves before the merge are drawn in the solid lines. The indices of the joints projected from the upper half and the class number for the class outlines in the lower half are indicated. The dash lines show the new class outlines after the merge. Let a point $f(i_f, j_f)$ be point $s(i_s, j_s)$'s *father* if point f dominates point s and $class(f) = class(s) - 1$. Point $s(i_s, j_s)$ is the *son* of point $f(i_f, j_f)$. Initially, we choose for each point its father in the class above it and with j_f closest to j_s than any other points in $class(f)$. Refer to Figure 3, point F's son is point S, and point S's father is point F. Note that one point may have more than one sons, but a son has only one father. Each arrow in Figure 3 points from a son to its father, and two trees are hence formed with joint 0 and joint 2 as their roots respectively. We can find that after the merge, the class numbers of point A, B, and C are different from the ones before the merge. They are updated to those in the parentheses. This type of the update is due to the insertion of joint 1 and 2 with the smaller j than that of the points. Such kind of update will be taken care of by the "Update with Joint" included in the merge algorithm.

On the other hand, we can find in Figure 3 that point D changed its class number to 5 (marked by an asterisk) after the merge. A dashed arrow points from D to C which is a point in another tree such that C dominates D and with the maximal class number. Since B has a new class number 4, D updated its class number to 5. Such kind of update is referred to as "Update between Trees" which is included in the merge algorithm.

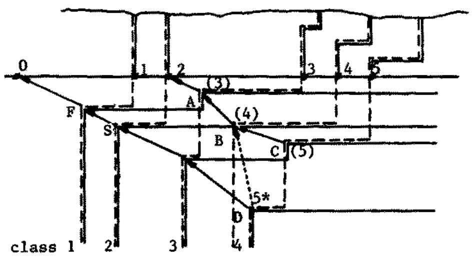

Figure 3

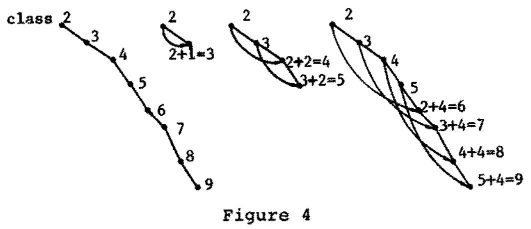

Figure 4

In a word, the merge step includes two phases in which the class of a point will be updated. When executing the first phase of the update, "Update with Joint", each point in the lower half is to find its father in the class above it. A point in class 1 is to find its father among the joints. A tree is hence formed by connecting consecutive fathers and sons, as is indicated by directed bold lines in Figure 3. A joint is the root of a tree, and is to notify its descendants of its index. A point receiving information from its ancestor can decide its new class by computing its depth from the root.

The second phase of the merge step, "Update between Trees". is conducted in a binary tree fashion, that is, in iteration i, 2^i trees should be merged together. Denote the left 2^{i-1} trees to be merged as *left trees*, and the right 2^{i-1} trees as *right trees*. Each point, say $p(i_p, j_p)$, is to find in the right trees the point $t(i_t, j_t)$ such that point t dominates point p and $class(t)$ is the maximum. If $class(t) + 1 > class(p)$, then $class(p)$ should be updated to $class(t) + 1$. Otherwise, it remains the same. If there are in total ρ trees in the lower half of the plane, then $\log\rho$ iterations are needed to complete the "Update between Trees".

The notification of the root of the tree to its descendants is performed in a "data compression" fashion. First, a point communicates with its son, and then with its son's son, that is the grandson, and then its grandsons's grandson, and so on. Following is the approach to complete the propagation.

Propagation

Figure 4 shows a propagation along a tree of 8 nodes, with the root indexed by 2. First, the root transferred its index, 2, to its son, and the son updated its class by increasing 1 on the received data, i.e. $2 + 1 = 3$. Then, the root and its son transferred their classes, 2 and 3 respectively, to their grandsons. Their grandsons updated their classes by increasing 2 on the received data, i.e. $2 + 2 = 4$ and $3 + 2 = 5$ respectively. Finally, the root, and its son, and their grandsons transferred their classes to their grandson's grandsons respectively. Their grandson's grandsons updated their classes by increasing 4 on the received data. This type of propagation is referred to as *Propagation 1*. Another type of propagation, *Propagation 2*, differs from it in that there may be multiple sources from which datum are propagated. In addition, a PE compares the received data with the one it contains before determining whether accepts the data or not.

In the detailed algorithm, instead of considering that the ancestors transfer the data to the descendants, we let the descendants to make requests of the data from the ancestors. Suppose originally, each point keeps the address of the PE maintaining its father. When making the request of the data from the father, the address of the PE maintaining the father's father can also be obtained. Now since each point has the address of the PE maintaining its grandfather, it is not difficult to obtain the address of the PE maintaining its grandfather's grandfather by similarly performing the described operation. For both of the two types of propagation, $\log\rho$ iterations are needed to complete the propagation along a tree of depth ρ.

Premax

Let $C_0, C_1, \cdots, C_n$ be the n data items in a list, $M(k) = max_{i=0}^{k}(C_i)$, for $k = 0, 1, \cdots, n-1$. We have $M(0) = C_0$, and $M(k) = max[M(k-1), C_k]$, for $k = 0, 1, \cdots, n-1$. Distribute the data on n PE's, the above recursive comparison can be performed as shown in Figure 5 for an example of $n = 8$. In the first step, each C_i maintained in PE(i) is compared with C_{i+1}, with the result $max[C_i, C_{i+1}]$ stored in PE($i+1$), for $i = 0, 1, \cdots, 6$. In step 2, the intermediate result in PE(i) is compared with the one in PE($i+2$), for $i = 0, 1, \cdots, 5$. In the final step, the intermediate result in PE(i) is compared with the one in PE($i+4$), for $i = 0, 1, \cdots, 3$. Consequently, PE(k) will have $M(k)$ as the final result, for $k = 1, 2, \cdots, n-1$. $O(\log n)$ steps are needed for n data items in the given list.

III. The algorithm and the time complexity

Before the execution of our algorithm, we assume the completion of some preprocessing, so that the r points (i, j) have been identified such that a_i in A matches b_j in B. The details of the preprocessing will be described in the next section.

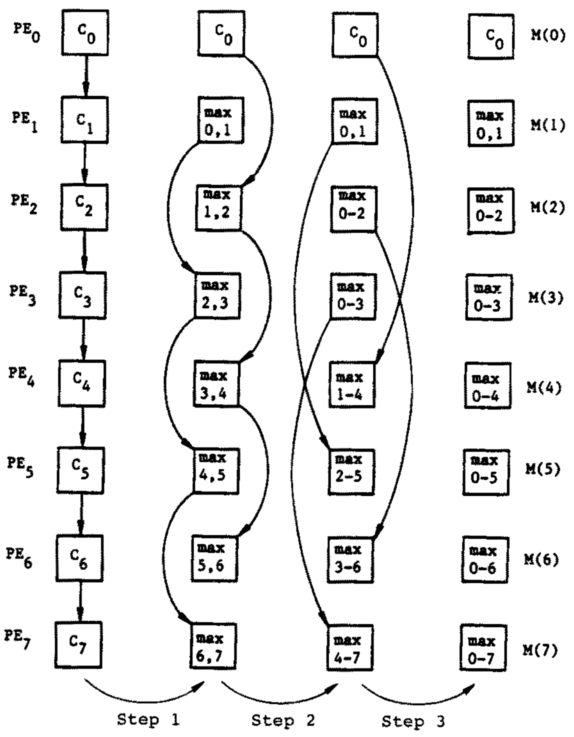
PE0 PE1 PE2 PE3 PE4 PE5 PE6 PE7
c0 c1 c2 c3 c4 c5 c6 c7
c0, max 0,1, max 1,2, max 2,3, max 3,4, max 4,5, max 5,6, max 6,7
c0, max 0,1, max 0-2, max 0-3, max 1-4, max 2-5, max 3-6, max 4-7
c0, max 0,1, max 0-2, max 0-3, max 0-4, max 0-5, max 0-6, max 0-7
M(0) M(1) M(2) M(3) M(4) M(5) M(6) M(7)
Step 1
Step 2
Step 3

Figure 5

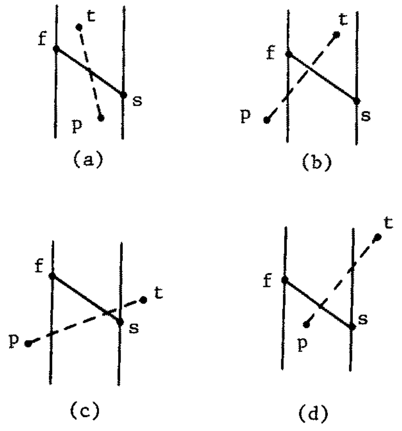

Figure 6

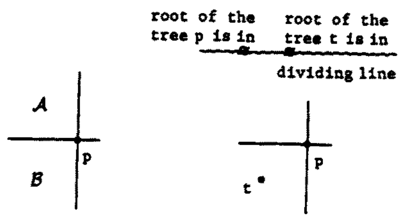

Figure 7

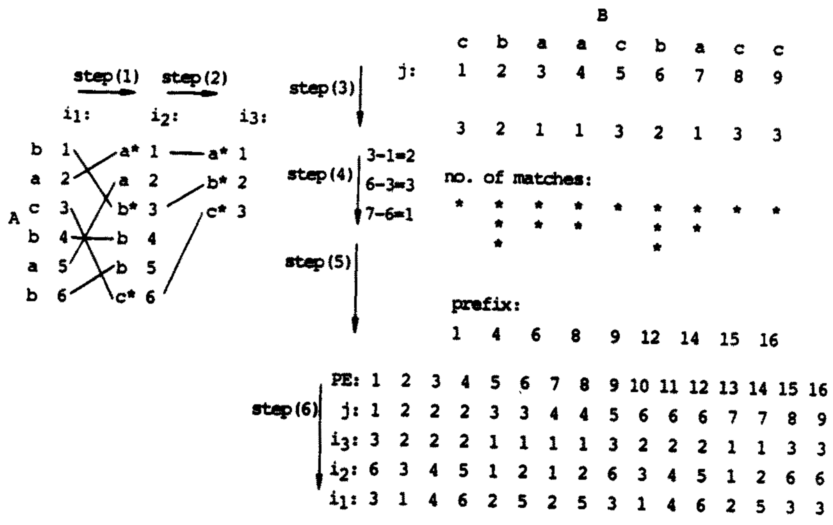
step(1)
step(2)
step(3)
step(4)
step(5)
step(6)
i1: i2: i3:
A
b 1, a 2, c 3, b 4, a 5, b 6
a* 1, a 2, b* 3, b 4, b 5, c* 6
a* 1, b* 2, c* 3
3-1=2
6-3=3
7-6=1
B
c b a a c b a c c
j: 1 2 3 4 5 6 7 8 9
3 2 1 1 3 2 1 3 3
no. of matches:
prefix:
1 4 6 8 9 12 14 15 16
PE: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
j: 1 2 2 2 3 3 4 4 5 6 6 6 7 7 8 9
i3: 3 2 2 2 1 1 1 1 3 2 2 2 1 1 3 3
i2: 6 3 4 5 1 2 1 2 6 3 4 5 1 2 6 6
i1: 3 1 4 6 2 5 2 5 3 1 4 6 2 5 3 3

Figure 8

Distribute the r points on r processors so that each processor contains one point. Based on the divide-and-conquer approach, we suppose that the class of each point in the upper or lower half has been identified before the merge.

Each PE which maintains a left most point of a class in the upper half now maintains a joint, that is, a point with the same j value but with the i value equal to that of the dividing line. The class of the point is considered as the index of the joint.

The merge algorithm can be described as follows.

```
Algorithm LCS
  1. Sort the points on each class outline by $j$ independently and concurrently.
/* Update with Joint */
  2. Each point in the lower half finds its father in the class above it; for those points in class
     1, each finds its father among the joints. Trees are formed. Set $flag(joint)$ to be 1, and
     $class(joint)$ to be the index of the joint.
  3. Perform Propagation 1 (class) on each tree.
/* Update between Trees */
do $\log(\rho+1)$ times (with $\rho = max(class(joint))$)
  begin
  4. Sort all the points in a tree by their $j$ values.
  5. Perform $Premax(class, M_c, Id)$ in each tree.
  6. Each point $p(i_p, j_p)$ finds in the right trees the point $t(i_t, j_t)$ such that $j_t < j_p$ and closest to $j_p$.
     Obtain $M_c(t)$ and $Id(t)$ from point $t$, and record them as $temp_1(p) = M_c(t)$, $temp_2(p) = Id(t)$.
  7. $dif(p) = temp_1(p) - class(p)$.
  8. if $dif(p) > dif(father(p))$ then set $flag(p) = 1$ and Perform Propagation 2 (dif).
  9. $class(p) = class(p) + dif(p) + 1$.
 10. if $dif(p) > dif(father(p))$ then $father(p) = temp_2(p)$;
  end
end /* Algorithm LCS */
```

The subroutine *Propagation* is presented below.

```
Algorithm Propagation 1 (class)
  1. Each point $p$ performs $addr = father(p)$.
for $i = 1$ to $\log\rho$ do (with $\rho$ being the maximum depth of the tree)
   begin
  2. Each point makes a request from PE($addr$).
     if $flag(addr) = 1$ then
     begin
  3.       $temp(p) = class(addr)$;
  4.       $class(p) = temp(p) + 2^{i-1}$;
  5.       $flag(p) = 1$.
     end
  6. $addr = addr(addr)$
   end
end /* Algorithm Propagation 1 */
```

Algorithm *Propagation 2 (value)* is similar to *Propagation 1 (class)* but with Step 3 and 4 modified as follows.

```
  3'.      $temp(p) = value(addr)$;
  4'.      if $temp(p) > value(p)$ then $value(p) = temp(p)$;
```

Following is the algorithm *Premax*.

```
Algorithm Premax (class, $M_c$, Id)
  1. Each PE($i$) performs $M_c(i) = class(i)$.
```

```
for $k = 0$, to $\log k - 1$, do
  begin
 2. All the PE($i$)'s route $M_c(i)$ and $i$ to PE($i + 2^k$). All the PE($i$)'s record the received data in
    $temp_1(i)$ and $temp_2(i)$ respectively.
 3. PE($i$)'s with $i > 2^k - 1$ perform
    if $temp_1(i) > M_c(i)$, then
     begin
 4.    $M_c(i) = temp_1(i)$;
 5.    $Id(i) = temp_2(i)$.
     end
  end
end /* Algorithm Premax */
```

We now prove that *Algorithm LCS* can find the correct class number for each point.

Lemma 1:

If a point q is dominated by a point p, then $class(q) > class(p)$.

This is from line 5 in *Compute L*, and the definition of "dominate".

Obviously, if $P\{p_i\}$ is a set of the points such that p_i dominates point q, then $class(q) > max[class(p_i)]$, $\forall i \in \{i | p_i \text{ dominates } q\}$.

Lemma 2:

A point q in class k is dominated by at least one point in class $k - 1$.

This is true because if no point in class $k-1$ dominates point q, then point q could have been in class $k-1$ instead of class k, contradiction.

Lemma 3:

The class outlines are numbered by consecutive integers.

This is from line 5 and line 6 in *Compute L*.

Lemma 4:

Let $P\{p_i\}$ be the set of all the points p_i dominating point q, then $class(q) < max[class(p_i)] + 2$, $\forall i \in \{i | p_i \text{ dominates } q\}$.

Proof:

Let $max[class(p_i)] = M$. From Lemma 1, if $class(q) \geq M + 2$, then q does not dominate any point on the class outline of $M + 2$. Then, either (i) q is on the class outline of $M + 2$, or (ii) q is dominated by a point, say q', on this outline.

According to Lemma 2, there is at least one point in class $M+1$ which either dominates point q for case (i), or dominates point q' for case (ii) and hence dominates point q. Since M is the maximum class number of the points dominating point q, the above described point in class $M+1$ contradicts the assumption.

Theorem 1:

Let $P\{p_i\}$ be the set of all the points p_i dominating point q, then

$$class(q) = max[class(p_i)] + 1, \forall i \in \{i | p_i \text{ dominates } q\}.$$

From Lemma 1, we have $class(q) > max[class(p_i)]$. From Lemma 4, we have $class(q) < max[class(p_i)] + 2$. Since the class numbers are integers (Lemma 3), so $class(q) = max[class(p_i)] + 1, \forall i \in \{i | p_i \text{ dominates } q\}$.

Lemma 5:

The left tree(s) and the right tree(s) formed in *Algorithm LCS* do not intersect.

Proof:

In the execution of *Algorithm LCS*, suppose $\overline{fs}$ is an edge of a tree generated in Step 2 of *Algorithm LCS*, and point f is the father of point s. Suppose $\overline{tp}$ is an edge of another tree, with point t being the father of point p.

If $\overline{tp}$ intersects the edge $\overline{fs}$, then there must be $j_t > j_f$ and $j_p < j_s$, as is the case shown in Figure 6(a). The other cases shown in Figure 6(b), (c) and (d) are not realistic because t can not be point p's farther with $j_t > j_p$. Investigate the class numbers of point p and point t for the case in Figure 6(a). Following are the three possibilities:

(i) $class(p) = class(s)$.

Then $class(f) = class(t) = class(s) - 1$. Point s should have chosen point t as its father since j_t is closer to j_s. Otherwise it contradicts the way that the tree is generated as is indicated in Step 2 of *Algorithm LCS*.

(ii) $class(p) > class(s)$.

Then $class(t) \geq class(s)$. According to Lemma 1, point t does not dominate point s. This contradicts the existence of the case in Figure 6(a).

(iii) $class(p) < class(s)$.

Then $class(p) \leq class(f)$. According to Lemma 1, point f does not dominate point p. This contradicts the existence of the case in Figure 6(a).

Thus, any two edges with one in a tree and the other in another tree can not intersect, therefore the trees formed in Step 2 of *Algorithm LCS* do not intersect. Since the initially generated trees do not intersect, the left trees and the right trees formed by merging some of them, as is done in the later steps of *Algorithm LCS*, do not intersect neither.

Lemma 6:

The point t found in the right trees by point p in Step 6 of *Algorithm LCS* dominates point p.

Proof:

Since $j_t < j_p$, so t must locate in area $\mathcal{A}$ or area $\mathcal{B}$ (see Figure 7(a). If t is located in area $\mathcal{A}$, then t dominates p.

If t is located in area $\mathcal{B}$, and t is in a right tree, then we have t on the left of p, and the root of the tree that t is in on the right of the root of the tree that p is in, then there must be some edge in one tree which intersects the edge in another tree, this contradicts Lemma 5. So, point t found in Step 6 can not be in area $\mathcal{B}$. It must be in area $\mathcal{A}$ and thus dominates point p.

Theorem 2:

Algorithm LCS finds for point q the point p_m among all the points p_i's such that p_i dominates q and $class(p_m) = max[class(p_i)], \forall i \in \{i|p_i \text{ dominates } q\}$. Thus $class(q)$ is correctly computed as $class(q) = max[class(p_i)] + 1, \forall i \in \{i|p_i \text{dominates } q\}$.

Proof:

(i) In the execution of "Update with Joint", each point in class 1 in the lower half found the point p_m in the upper half such that p_m dominates q and $class(p_m)$ is the maximum.

From Step 2 in *Algorithm LCS*, each point in class 1 found its father among the joints. According to the definition of *father*, the joint chosen to be point $q(i_q, j_q)$'s father, say $f(i_f, j_f)$, must dominate point q and have j_f closest to j_q. Since joint $(f+1)$ has $j_{(f+1)} > j_q$ (otherwise q would have chosen joint $(f+1)$ as its father), and a *joint* is projected from the left most point in its class, all the points within class $(f+1)$ must lie on the right of point q, and thus f must be the maximum class number of those points dominating point q.

(ii) In the execution of "Update between Trees", each point q with $class(q) > 1$ in the lower half found the point p_m such that $class(p_m)$ is the maximum among the points which dominate point q.

From Lemma 6, the point t found in Step 6 of *Algorithm LCS* by p dominates point p. Any point t' in the same tree that t is in and with $j_{t'} < j_t$ should dominate p either. After executing Step 5 in *Algorithm LCS*, the maximum class number of point t and all the points t' is known, as $M_c(t)$. By step 7, $M_c(t) = class(p) + dif(p)$. Thus if the condition in step 8 does not hold, after step 9, point p will have updated its class number as $M_c(t) + 1$ where t is recognized as the point in the right tree with maximum class number dominating p. If the condition in step 8 holds, then $class(p) + dif(father(p)) > class(p) + dif(p)$. That means $class(father(p)) + 1 + dif(father(p)) > M_c(t)$, and $father(p)$ will have a greater class number, than t, assuming updated in the above way. In this case, p updates its class number based on its father's new class number, as $class(p) + dif(p) + 1$ with $class(p) = class(father(p)) + 1$ as was before and $dif(p) = dif(father(p))$ after performing *Propagation 2 (dif)*. According to Theorem 1, the class of p is correctly computed.

Executing on r processors, with r equal to the total number of pairs of positions at which two symbols match, our algorithm is with the time complexity analyzed as follows. The maximal number of points on a outline is $m+n$, with $n \geq m$. Hence the sorting in step 1 takes $O(\log n)$ time [11]. Step 2 involves a binary search on the class outline, hence $O(\log n)$ is required in the worst case. *Propagation* includes $O(\log \rho_1)$ iterations in step 3 and for the worst case in step 8, if ρ_1 is the total number of classes in the lower half to be merged. Step 4 to step 10 need to be repeated $\log(\rho_2+1)$ times with ρ_2 equal to the total number of classes in the upper half to be merged. In Step 4, sorting of k points in the trees requires $O(\log k)$ time. *Premax* involved in Step 5 needs $\log k$ steps to find the desired maximal data for the k given data items. A point is to search in Step 6 for the point with the smaller and closest j in the right trees, which needs $O(\log k)$ time given k points in the trees. For each of these steps, $k=n$ is the worst case, thus $O(\log n)$ will be the time bound. Step 7, 9 and 10 involve constant time computation only. The whole merge algorithm needs to be executed $O(\log n)$ times to complete the divide-and-conquer operation, thus, the total time needed is $O(\log \rho \log^2 n)$ with ρ being the longest common subsequence.

IV. Discussion of preprocessing

At the beginning of the algorithm presented in the previous section, we assumed that all the pairs of the positions on which the two strings match have been determined. This is completed by preprocessing which we have not discussed yet. Given two strings with length m and n respectively, let the pair (i,j) indicate the match of two symbols at position i in string A and position j in string B. To find (i,j) pairs, we need m processors to maintain symbols in A and n for B. Distribute the symbols on the $m+n$ PE's, with one symbol per PE. Assume that the symbols have some order. Each PE containing a symbol in A generates a record with two fields: $< i, order >$ and those containing a symbol in B generate records $< j, order >$'s. i, j indicate the position of the symbol in string A and B respectively. *order* indicates the order of the symbol contained in the PE. Following are the operations to be performed in the preprocessing.

Preprocessing

(1) Sort the m symbols in A by *order* on m PE's. Ties are broken by i.

(2) Select the distinct symbols each with a smallest rank in the sorted sequence to form a "concentrated sequence" (see Figure 8 as a reference). Assume that there are in total t distinct symbols.

(3) Each PE maintaining a symbol in B performs a binary search on the t records to find an *order* which matches the *order* maintained in its record for a point.

(4) Compute for the two adjacent distinct symbols in the concentrated sequence generated in step (2), the difference of their ranks in the sequence generated in step (1). Each symbol in string B now knows the total number of the positions in A at which the symbol can be found.

(5) Perform a parallel prefix computation, each symbol in B, say in position j, obtains the total number, say c_j, of matches found for all the symbols prior to it. Prepare r PE's to execute the *Algorithm LCS* if r matches are found in total. Each PE maintaining a symbol in B notifies the c_jth PE among r PE's the position j in B, the symbol it is assigned and the position of the symbol in the sorted sequence of B.

(6) By reverse tracing (say $i_3 \rightarrow i_2 \rightarrow i_1$ in Fig. 8), each of the r PE's can get the information about the position i in A at which the symbol lies.

We next compute the running time to accomplish the preprocessing. In step (1), sorting m elements on m processors needs $O(\log m)$ time. Step (2) requires $O(\log m)$ time in the worst case. The binary search performed on m items in step (3) is running on $O(\log m)$ time. Step (4) needs only constant time. In step (5), the parallel prefix computation is performed on n PE's each containing a symbol in string B. The time complexity of parallel prefix computing for n data items is $O(\log n)$ [12], therefore the time needed in step (6) is no greater than $O(\log n)$.

Thus, distributing the symbols on $m+n$ processors, the preprocessing time to find the matches for two strings with length m and n is bounded by $O(\log m + \log n)$, including the assigning of the matched pairs to r PE's to execute the LCS algorithm.

V. Conclusion and related research

A parallel algorithm for computing the longest common string has been presented. The described algorithm suggested the employment of a divide-and-conquer approach which exploited the concurrency in solving LCS problems and hence achieved efficient results, compared with existing algorithms of which most are sequential.

A related research problem which can be solved is the maxima problem occurred very often in computational geometry. Given a set S of n points in the plane, a point p in S is a *maximal element* (or, briefly *maxima*) of S if there does not exist q other than p in S such that $x_q \geq x_p$ and $y_q \geq y_p$. As an example, the maxima of a given set of points shown in Figure 9 have been connected by dash lines. In addition, excluding the maxima on the dash lines, we can find the second layer maxima and so on. This problem can be solved by extending the algorithm presented in the earlier sections.

Let the four extreme points N, S, E and W define the four quadrants, assign different signs $+$ and $-$ to the coordinates of the points. It can be observed that the problem of determining the maxima in a quadrant is similar to the problem of determining the points with smallest class number in the L matrix. Furthermore, the layers of maxima can be found in parallel, based on the approach of divide-and-conquer, and the methods we provided in the LCS algorithm design. Distributing n points on n PE's with one point per PE, the layers of the maxima in a given set can be determined in $O(\log \rho \log^2 n)$ time, where ρ is the number of the layers. We believe that the importance of the generated idea lies not only in the solutions of these problems, but also in that they provided valuable insight into the difficulty of parallelism exploiting.

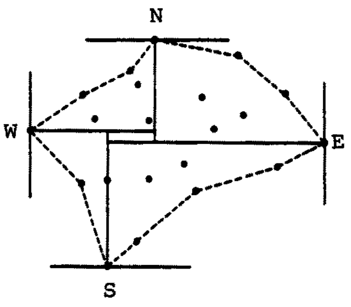

Figure 9

References

[1] D. Sankoff and J. B. Kruskal, editors, *Time warps, string edits, and macromolecules: the theory and practice of sequence comparison*, Reading, MA: The MIT Press, 1985.

[2] J. L. Modelevsky, "Computer applications in applied genetic engineering," *Advances in Applied Microbiology* Vol. 30, 1984, pp. 169-195.

[3] Y. Chiang and K. S. Fu, "Parallel processing for distance computation in syntactic pattern recognition," *Proceedings of IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, Nov. 1981.

[4] D. S. Hirschberg, "A linear space algorithm for computing maximal common subsequences," *Communications of the ACM*, Vol. 18, No. 6, June 1975, pp. 341-343.

[5] A. V. Aho, D. S. Hirschberg and J. D. Ullman, "Bounds on the complexity of the maximal common subsequence problem," *Proceedings of 15th Annual IEEE Symposium on Switching and Automata Theory*, 1974, pp. 104-109.

[6] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *Journal of the ACM*, Vol. 24, no. 4, Oct. 1977, pp. 664-675.

[7] W. J. Hsu and M. W. Du, "New algorithms for the LCS problem," *Journal of Computer and System Sciences*, 29, 1984, pp. 133-152.

[8] W. J. Hsu and M. W. Du, "Computing a longest common subsequence for a set of strings," *Bit*, 24, 1984, pp. 45-59.

[9] D. P. Lopresti and R. Hughey, "The B-SYS programmable systolic array", Technical Report CS-89-32, Department of Computer Science, Brown University, Providence, June 1989.

[10] P. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of ACM*, 21 (1), 1974, pp. 168-173.

[11] R. Cole, "Parallel Merge Sort," *SIAM J. on Comp.*, Vol. 17, No. 4, Aug. 1988, pp. 770-785.

[12] A. Gibbons and W. Rytter, "Efficient Parallel Algorithms", Cambridge University Press, Cambridge, 1988, pp. 13-18.