中國工程學刊第十卷第六期（民國七十六年）
Journal of the Chinese Institute of Engineers. Vol. 10, No. 6, pp. 691 - 699 (1987)

SYSTOLIC ALGORITHMS FOR THE LONGEST COMMON SUBSEQUENCE PROBLEM

Chang-Biau Yang

Institute of Computer and Decision Sciences
National Tsing Hua University
Hsinchu, Taiwan 30043, R.O.C.

R. C. T. Lee*

Department of Electrical Engineering
National Tsing Hua University
Hsinchu, Taiwan 30043, R.O.C.

Key Words: parallel processing, systolic algorithm, VLSI, longest common subsequence problem.

ABSTRACT

The concept of systolic array processors is very suitable for VLSI implementation. In this paper, we propose two systolic algorithms to solve the longest common subsequence problem by dynamic programming approach and also prove that these two algorithms are correct. The order of the time-processor-product of our algorithms is equal to that of the corresponding sequential method.

最長共同子序列問題的心跳式計算方法

楊　昌　彪

國立清華大學計算機管理決策研究所

李　家　同

國立清華大學電機工程學系

摘　要

心跳式列陣處理器的理念非常適合做成超大型積體電路。動態規劃是人們用來解決最長共同子序列問題的一種常見方法，在這篇文章中，我們以動態規劃為基礎，提出了兩個心跳式計算方法來解決這個問題，並且證明我們的方法正確無誤，我們方法的時間處理器乘數和這個動態規劃方法所需時間相等。

THE LONGEST COMMON SUBSEQUENCE PROBLEM

The longest common subsequence (LCS for short) problem has been studied by many researchers [4, 5, 7, 8, 17, 19, 20]. For a complete review of this subject, see Ref. [7]. The spoken word recognition problem can be shown to be quite similar to the LCS problem [14, 15].

A string consists of a sequence of symbols. A subsequence of a string is obtained by deleting zero or more symbols, which are not necessarily consecutive, from the original string. For example, '*abc*' is a subsequence of '*caadbec*'. Note that '*caadbec*' itself is a subsequence of '*caadbec*'. A string *C* which is a subsequence of both strings *A* and *B* is called a common subsequence of *A* and *B*. For instance, '*ac*' is a common subsequence of '*abc*' and '*aace*'. Two strings may have more than one common subsequence. For instance, '*a*' and '*ac*' are both common subsequences of '*abc*' and '*aace*'. The longest common subsequence problem is: Given two strings *A* and *B*, find a common subsequence of *A* and *B* which is the longest.

A simple method to solve the LCS problem is

* R.C.T. Lee is also with Academia Sinica, Taipei, Taiwan, R.O.C.

to use the dynamic programming approach [2, 4, 6, 7, 17, 20]. Let A and B be two input strings. Let the lengths of A and B be m and n respectively. Without losing generality, we may assume that $m \leq n$. Let $A=a_1, a_2, \cdots, a_m$ and $B=b_1, b_2, \cdots, b_n$. Let $L(i, j)$ denote the length of an LCS of $a_1, a_2, \cdots, a_i$, $1 \leq i \leq m$, and $b_1, b_2, \cdots, b_j$, $1 \leq j \leq n$. Using the dynamic programming approach, we have

$$L(i, j)=L(i-1, j-1)+1, \qquad \text{if } a_1=b_2$$

$$L(i, j)=max\,(L(i, j-1), L(i-1, j)), \qquad \text{otherwise.}$$

We also have the following boundary conditions:

$$L(i, 0)=0, \qquad \text{for } i=0, 1, 2, \cdots, m$$

$$L(0, j)=0, \qquad \text{for } j=0, 1, 2, \cdots, n.$$

By the above equations, we can write a sequential program to find $L(m, n)$ as follows:

```
do  j:=0 to n
    L(0, j):=0;
enddo;
do  i:=1 to m
    L(i, 0):=0;
    do j:=1 to n
      if A(i)=B(j) then L(i, j):=L(i-1, j-1)+1
      else L(i, j):=max (L(i, j-1), L(i-1, j));
    enddo;
enddo;
```

After executing the program, the length of a LCS of strings A and B, i.e., $L(m, n)$ is found. Figure 1 shows an L matrix which is established by the above program for A='*bacad*' and B='*accbadcb*'. Each element of the L matrix corresponds to an $L(i, j)$. It is obvious that the time complexity of establishing the L matrix by the dynamic programming approach in a sequential method is $O(mn)$.

B

A			a	c	c	b	a	d	c	b
		0	0	0	0	0	0	0	0	0
	b	0	0	0	0	1	1	1	1	1
	a	0	1	1	1	1	2	2	2	2
A	c	0	1	2	2	2	2	2	3	3
	a	0	1	2	2	2	3	3	3	3
	d	0	1	2	2	2	3	4	4	4

Fig. 1. An example for the L matrix.

After each $L(i, j)$ has been found, we can use a backtracking algorithm to find a longest common subsequence of two given strings [7]. In this paper, we are only interested in finding the length of the longest common subsequence efficiently.

A BRIEF INTRODUCTION TO SYSTOLIC ALGORITHMS

Recently, because of the progress of integrated circuit technology, VLSI design has been discussed by many researchers. Kung proposed an architecture, systolic array processor, which has the following properties [3, 10-13, 18].

(1) Modularity: A systolic array processor consists of many processing elements (PE's for short). Each PE can work independently.
(2) Regularity: Almost all PE's in a systolic array processor have the same functions and circuits. After we design the circuit of one PE, we can repeat it to other PE's. Thus, the needed time of a chip design is decreased drastically.
(3) Simplicity: The function and circuit in each PE are very simple. Therefore, it is easy to design the circuit of one PE and many PE's can be built into one VLSI chip.
(4) Local interconnection: In one VLSI chip, data communication becomes significant and dominates the cost and performance of it. Systolic array processors minimize the undesirable data communication. Each PE in a systolic array processor is connected to and communicates with its neighboring PE's only.
(5) Pipelining and multiprocessing: Many schemes of parallel processing have been discussed in Ref. [9]. There are many PE's in a systolic array processor and they can work in parallel. Furthermore, several data streams move at constant velocity over fixed paths in the systolic array processor, interacting at cells where they meet. Data referenced in one PE may be referenced in its neighboring PE's. This scheme serves the pipelining.
(6) Balancing computation with I/O: Too many I/O operations in a system will decrease its performance and unlimited available I/O bandwidth in a system is impossible. Usually, data are pushed into a systolic array processor from the external memory and then referenced by many computations through some fixed paths. This scheme prevents the situation where too many I/O operations occur.

By the above properties, the concept of systolic array processors is very suitable to be implemented for VLSI chips. There have been many problems solved by systolic algorithms [1, 3, 10-12, 16, 18, 22]. In this paper, we shall design two systolic algorithms to solve the LCS problem through the dynamic programming approach.

A SYSTOLIC ALGORITHM TO COMPUTE THE LENGTH OF AN LCS OF TWO STRINGS

Figure 2 shows the computing sequence for each $L(i,j)$. That is, $L(i,j)$ can be calculated after $L(i-1,j)$, $L(i,j-1)$ and $L(i-1,j-1)$ have been determined. Each shaded PE indicates its corresponding $L(i,j)$ value has already been found.

Based upon the above observation, we may easily propose a wavefront parallel algorithm to compute $L(i,j)$ (The concept of the wavefront algorithms was described in Ref. [11]). Figure 3 shows the first three time-steps of this parallel algorithm. Note that the number of time-steps needed in this parallel algorithm reduces to $m+n-1$. The number of PE's required is mn.

A commonly used measurement to measure the efficiency of a parallel algorithm is the time-processor-product (TPP for short). For this algorithm, TPP is equal to $(m+n-1)mn$.

In the following, we shall show that this straightforward wavefront algorithm can be improved to a linear systolic algorithm. This is based upon the observation that there is only one wavefront at each time-step and no rolling back of wavefronts is needed.

Let us assume that the lengths of strings A and B are m and n respectively and $m \leq n$. We shall now use $2n$ PE's. Consider the case when A='ab' and B='abc'. Figure 4 shows how one linear systolic algorithm works. Note that string A flows from the right side and string B flows from the left side.

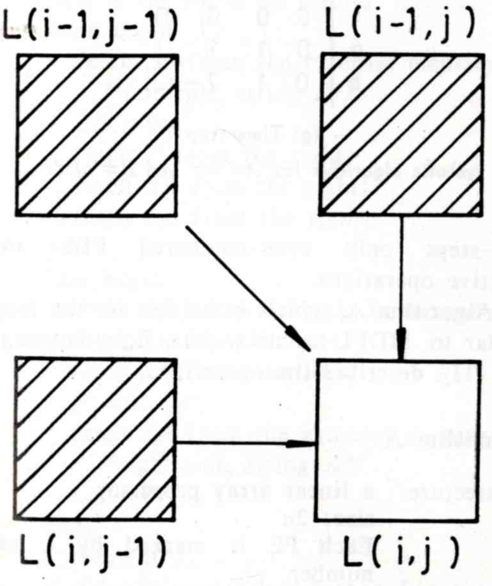

Fig. 2. Computing sequence of the L matrix.

Each PE in Fig. 4 contains six registers storing the data as follows:

SA: storing a symbol of string A, a_i
LA: storing $L(i,j-1)$, data flowing with a_i
SB: storing a symbol of string B, b_j
LB: storing $L(i-1,j)$, data flowing with b_j
L: storing the computed result $L(i,j)$
P: storing $L(i-1,j-1)$.

Each PE is marked by a natural number. The leftmost PE is numbered 1 and the rightmost PE is numbered $2n$. At the first time-step, the first symbol of string B is pushed into PE(1) and LB of

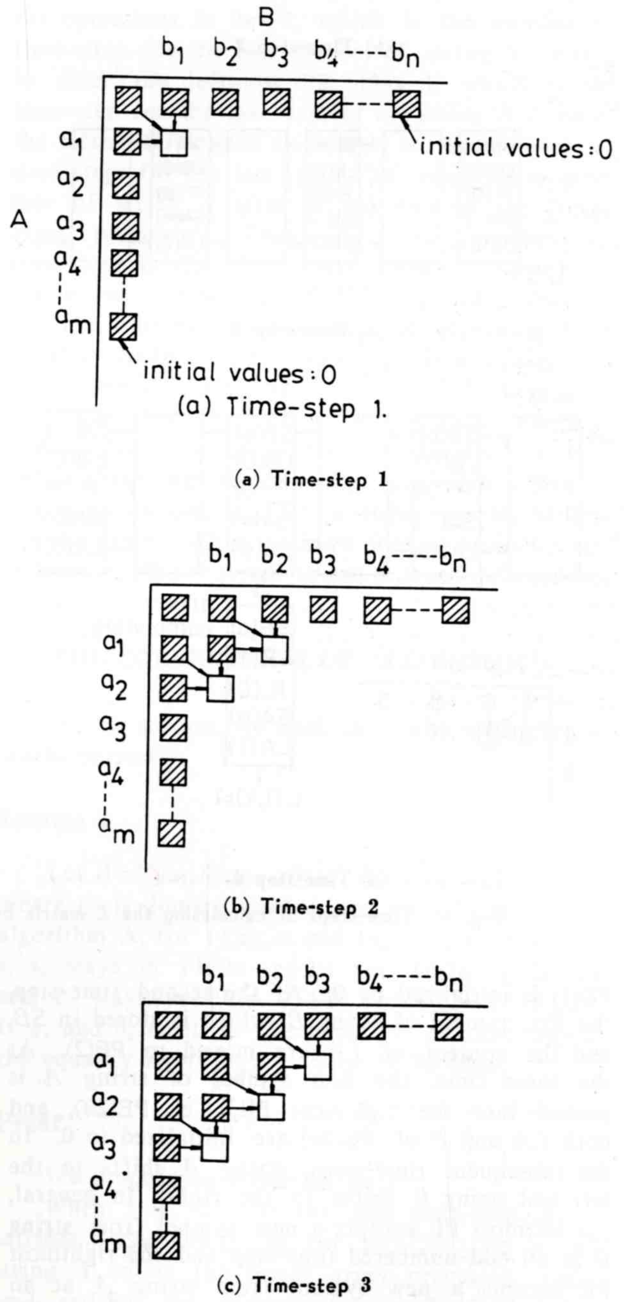

Fig. 3. Calculating the L matrix by wavefront parallelism. Note that a shaded square indicates that the corresponding value has been determined.

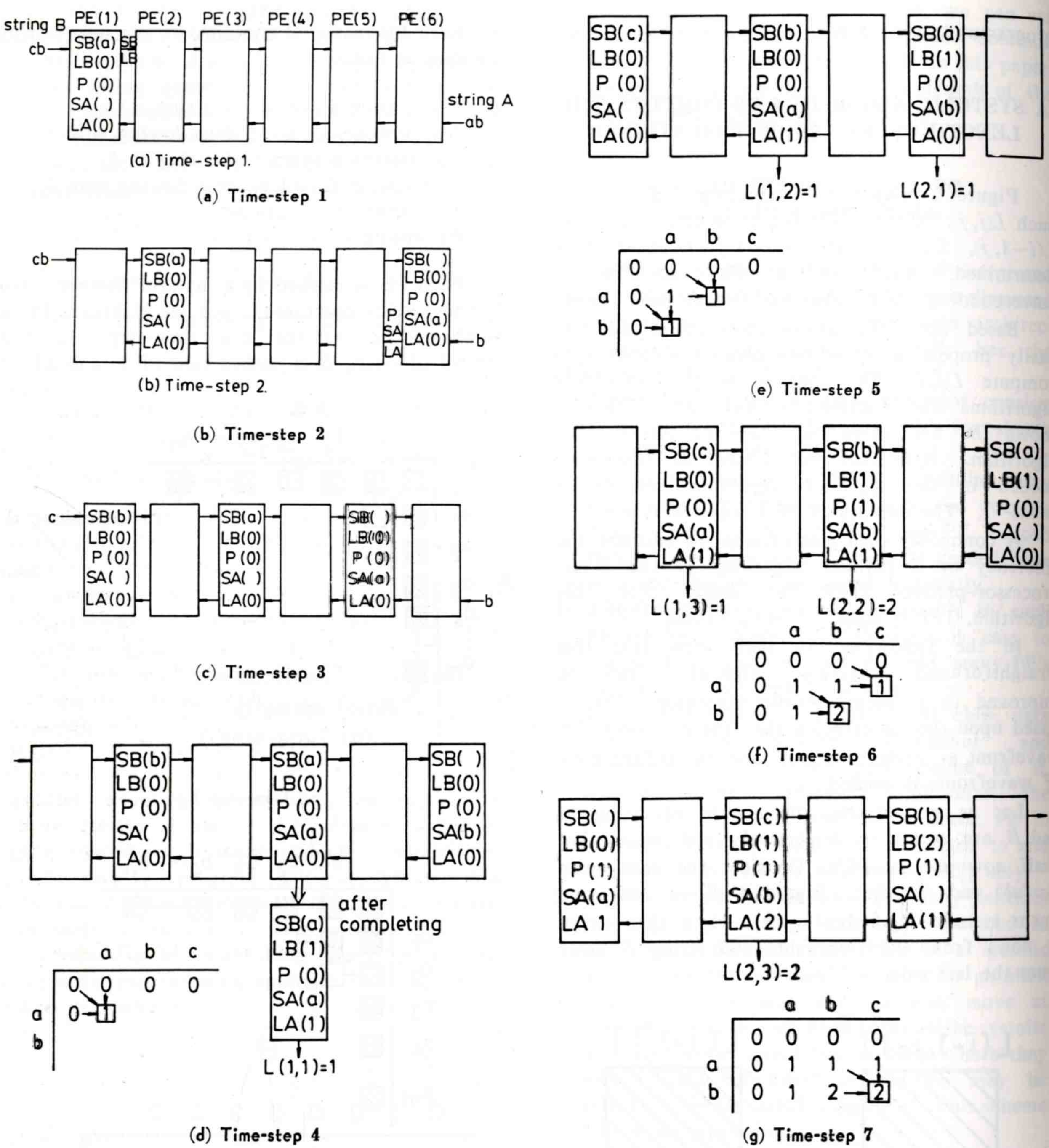

Fig. 4. Time-steps of calculating the L matrix by a linear systolic algorithm for A='ab' and B='abc'.

PE(1) is initialized by 0. At the second time-step, the first symbol of string B, which is stored in SB, and the content of LB are moved to PE(2). At the same time, the first symbol of string A is pushed into the rightmost PE, i.e., PE($2n$), and both LA and P of PE($2n$) are initialized to 0. In the subsequent time-steps, string A shifts to the left and string B shifts to the right. In general, the leftmost PE accepts a new symbol from string B at an odd-numbered time-step and the rightmost PE accepts a new symbol from string A at an even-numbered time-step. Note that for odd-numbered time-steps, only odd-numbered PE's execute effective operations, and for even-numbered time-steps only even-numbered PE's execute effective operations.

Algorithm A, which is written in the language similar to MDFL (matrix data flow language) in Ref. [11], describes the operations:

Algorithm A

architecture: a linear array processor
size: $2n$
Each PE is marked by a natural number.
The leftmost PE is numbered 1 and the rightmost PE is numbered $2n$.

computation: the length of an LCS of strings A and B, where
A='$a_1, a_2, \cdots, a_m$'
B='$b_1, b_2, \cdots, b_n$'
and $m \leq n$.

initial: String A is stored in the external memory on the right side of the linear array processor by the form:
A='@a_1, @a_2,$\cdots$, @a_m'.
String B is stored in the external memory on the left side of the linear array processor by the form:
B='b_1, @b_2,$\cdots$, b_n@'.
In the above, '@', which is a dummy symbol and it is not in the symbol set of strings A and B, is inserted into strings A and B in order to handle the data flow conveniently.
P, LA and LB in all PE's are set to zero.
SA and SB in all PE's are set to '@'.
Symbols of strings A and B are pushed into the linear array processor one by one.

final: The length of an LCS of strings A and B, $L(m, n)$, is stored in PE(m+1).

```
beginprogram
  TIMESTEP:=2*n+m-1;
  repeat
    all PE's do simultaneously
    begin
      if the PE is not on the left or right
         boundary
      then begin
        fetch SB from the left;
        fetch LB from the left;
        fetch P from the right;
        fetch SA from the right;
        fetch LA from the right;
      end
      else if the PE is on the left boundary
      then begin
        fetch SB from the external memory;
           /*left side, string B*/
        LB:=0;
        fetch P from the right;
        fetch SA from the right;
        fetch LA from the right;
      end
      else begin
        /*the PE is on the right boundary*/
        fetch SB from the left;
        fetch LB from the left;
        P:=0;
        fetch SA from the external memory;
           /*right side, string A*/
        LA:=0;
      end;
      /*end if*/
      if SA≠'@' and SA=SB then L:=P+1
      /*L remains 0 before two strings meet*/
        else L:=max(LA, LB);
        P:=LB;
        LA:=L;
        LB:=L;
        flow SB to the right;
        flow LB to the right;
        flow P to the left;
        flow SA to the left;
        flow LA to the left;
    end;
    TIMESTEP:=TIMESTEP-1;
  until TIMESTEP=0;
endprogram;
```

The total number of time-steps to complete the operations is $2n-2$, which is the number of time-steps for the last symbol of string B waiting to enter the leftmost PE, plus 1, which is the time-step for the last symbol of string B to enter the leftmost PE, plus m, which is the number of time-steps for the last symbol of string B to meet that of string A after it has entered the linear array processor. Thus, the total number of time-steps is $(2n-2)+1+m=2n+m-1$ (Remember that $n \geq m$), the number of PE's required is $2n$.

The time-processor-product of algorithm A is equal to $2n(2n+m-1)$. Note that if a straightforward method is used as discussed in the beginning of this section, the TPP is $(n+m-1)mn$, which is higher. The number of registers required in each PE of these two algorithms is the same. Besides, the complications of PE's in these two algorithms are the same. What we have done is to reduce the number of PE's required in one order of polynomial.

THE CORRECTNESS OF ALGORITHM A

In this section, we shall show why algorithm A works correctly.

Lemma 1

Let A='@a_1, @a_2,$\cdots$, @a_m', B='b_1@, b_2@,$\cdots$, b_n@', where @ is dummy symbol, and $n>0$, $0<m \leq n$. In algorithm A, for $1 \leq i \leq m$ and $1 \leq j \leq n$, at time-step s, a_i stays in PE($2n-s+2i$) for $1 \leq 2n-s+2i \leq 2n$ and b_j stays in PE($s-2j+2$) for $1 \leq s-2j+2 \leq 2n$. If a_i and b_j meet in the same PE at time-step s, the equality $s=n+i+j-1$ must be satisfied.

Proof

We proceed by induction on s.

When $s=1$, i. e., at the first time-step, no i satisfies $1 \leq 2n-s+2i \leq 2n$ for $1 \leq i \leq m$. No a_i of string A enters the linear array processor. Only $j=1$ satisfies the inequality $1 \leq s-2j+2 \leq 2n$, for $1 \leq j \leq n$, and only b_i enters PE($s-2j+2$)= PE($1-2+2$)=PE(1). Thus, when $s=1$, this lemma is trivially true.

Assume that it is true for $s-1$, $s\leq 2n$. That is, at time-step $s-1$, a_i stays in PE($2n-s+1+2i$) for $1\leq 2n-s+1+2i\leq 2n$. Consider a time step s. At time-step s, each a_i in the linear array processor moves to the next left PE from the PE it stays in at time-step $s-1$. Thus, the number of the PE where a_i stays in is decreased by one, i.e., a_i enters PE($2n-s+2i$). If $2n-s+2i=1$, a_i stays in PE(1) at time-step $s-1$. At time-step s, $2n-s+2i=0$ and a_i moves out of the linear array processor. If $2n-s+1+2i=2n+1$, at time-step $s-1$, a_i, which is the first candidate for the next time-step to enter the rightmost PE, i.e., PE($2n$), is left in the external memory. At the next time-step s, $2n-s+2i=2n$ and a_i enters the rightmost PE. Therefore, for $1\leq i\leq m$, at time-step s, a_i stays in PE($2n-s+2i$) for $1\leq 2n-s+2i\leq 2n$.

Similarly, we can prove that b_j stays in PE($s-2j+2$) for $1\leq s-2j+2\leq 2n$ at time-step s.

If a_i and b_j meet in the same PE at time-step s, we know that a_i stays in PE($2n-s+2i$) and b_j stays in PE($s-2j+2$). Thus $2n-s+2i=s-2j+2$. The equality $s=n+1+j-1$ can be derived from this equation. That is, the equality $s=n+i+j-1$ is satisfied. Q. E. D.

Lemma 2

In algorithm A, for all i, j, $1\leq i\leq m$, $1\leq j\leq n$, a_i and b_j meet once and only once in some PE at some time-step. At time-step $2n+m-1$, a_m and b_n meet in PE($m+1$).

This lemma is easy to prove from lemma 1 and we omit the proof here.

Theorem 1

Algorithm A computes the length of an LCS of strings A and B at time-step $2n+m-1$, and the result $L(m, n)$ is stored in PE($m+1$).

Proof

What we have to prove is that the operations of each PE in algorithm A correspond to the dynamic programming approach:

if $a_i=b_j$ then $L(i, j)=L(i-1, j-1)+1$
else $L(i, j)=max\,(L(i, j-1), L(i-1, j))$.

We shall prove the theorem by induction on i and j.

Whenever any new symbol of string B enters the leftmost PE, LB is set to zero, which is described in lines 14-21 in algorithm A. Whenever any new symbol of string A enters the rightmost PE, P and LA are set to zeros. This is described in lines 22-28 in algorithm A. Each of SA and SB in all PE's is initialized by the dummy symbol '@', and P, LA and LB in all PE's are initialized to be zero. Lines 29-30 prevent L to be increased before the two strings meet. Thus, before the two strings meet in the same PE, P, LA and LB are still zeros. This satisfies the boundary conditions:

$$L(i, 0)=0, \qquad \text{for } i=0, 1, 2, \cdots, m$$
$$L(0, j)=0, \qquad \text{for } j=0, 1, 2, \cdots, n.$$

When $i=1$ and $j=1$, from lemma 1 and lemma 2, a_1 and b_1 meet in PE($n+1$) at time-step $n+1$. Before executing line 29, $LA=LB=P=0$, i.e., LA, LB and P store the correct values of $L(1, 0)$, $L(0, 1)$ and $L(0, 0)$ respectively, and a_1 and b_1 are stored in SA and SB respectively. After executing lines 29-30, we can obtain $L=1$ (That is $L(1, 1)=1$.) if $a_1=b_1$, and we can obtain $L=0$ (That is $L(1, 1)=0$.) if $a_1\neq b_1$.

Assume that this theorem is correct for $L(i-1, j)$, $L(i, j-1)$ and $L(i-1, j-1)$, where $1\leq i\leq m$ and $1\leq j\leq n$, and LA, LB and P also carry the correct values in the PE's which generate $L(i-1, j)$, $L(i, j-1)$ and $L(i-1, j-1)$ at some time-steps. From lemma 1 and lemma 2, at time-step $n+i+j-2$, a_{i-1} and b_j meet in PE($n+i-j$), lines 29-30 compute $L(i-1, j)$ correctly, and in line 33, the value of $L(i-1, j)$ is copied to LB. Then, in line 35, LB is flowed to the right PE, i.e., PE($n+i-j+1$), for the reference at the next time-step. Similarly, at time-step $n+i+j-2$, a_i and b_{j-1} meet in PE($n+i-j+2$). Before executing line 29, the value of $L(i-1, j-1)$ is stored in LB (The value stored in LB should be correct since we have assumed that $L(i-1, j)$ is correct.). Lines 29-30 compute $L(i, j-1)$. Line 31 copies the value of LB to P, i.e., P is assigned to the value of $L(i-1, j-1)$. Line 32 makes LA assigned to the value of $L(i, j-1)$. In lines 36 and 38, the contents of P and LA are moved to the left PE, i.e., PE($n+i-j+1$), for the reference at the next time-step.

At the next step, i.e., time-step $n+1+j-1$, a_i and b_j meet in PE($n+i-j+1$). In lines 6-28, PE($n+i-j+1$) obtains the correct values for LB, P and LA from the proper PE's. From the above analysis, we know that LB stores $L(i-1, j)$, P stores $L(i-1, j-1)$ and LA stores $L(i, j-1)$. Lines 29-30 compute $L(i, j)$ correctly as the dynamic programming approach does. Therefore, for all i, j, $1\leq i\leq m$, $1\leq j\leq n$, $L(i, j)$ is computed correctly.

From lemma 2, at time-step $2n+m-1$, a_m and b_n meet in PE($m+1$). From the above proof, $L(m, n)$ is computed correctly and is stored in PE($m+1$). Q. E. D.

From the above proofs, algorithm A works correctly.

AN IMPROVED SYSTOLIC ALGORITHM

In the scheme of algorithm A in the previous section, it is obvious that only half of PE's execute effective operations at any time-step. That is, half of PE's are useless at every time-step. Besides, the number of PE's required depends on n, which is the larger of lengths of the two input strings (Remember that $n \geq m$.). In this section, we propose an improved method. The number of time-steps required in this improved algorithm is $n+2m-1$ and the number of PE's required is reduced to m, which is smaller than n.

Figure 5 shows an example of this improved scheme for A='ab' and B='abc'. Each PE in Fig. 5 also contains six registers as shown in Fig. 4 and its function is the same as that in Fig. 4. In this scheme, we use only m PE's. Each PE is also marked by a natural number. The leftmost PE is numbered 1 and the rightmost PE is numbered m. In the first m time-steps, the symbols of string A enter the linear array processor sequentially from the rightmost PE and string B stays still in the external memory. After time-step m, each symbol of string A stays still in the PE and never moves. The symbols of string B enter the linear array processor sequentially from the leftmost PE. The operations are the same as those in Fig. 4 except that string B is not moved in the first m time-steps and string A is not moved in the subsequent time-steps. The details are described in algorithm B.

Algorithm B

architecture: a linear array processor
size: m
Each PE is marked by a natural number.
The leftmost PE is numbered 1 and the rightmost PE is numbered m.

computation: the length of an LCS of strings A and B, where
A='$a_1, a_2, \cdots, a_m$'
B='$b_1, b_2, \cdots, b_n$'
and $m \leq n$.

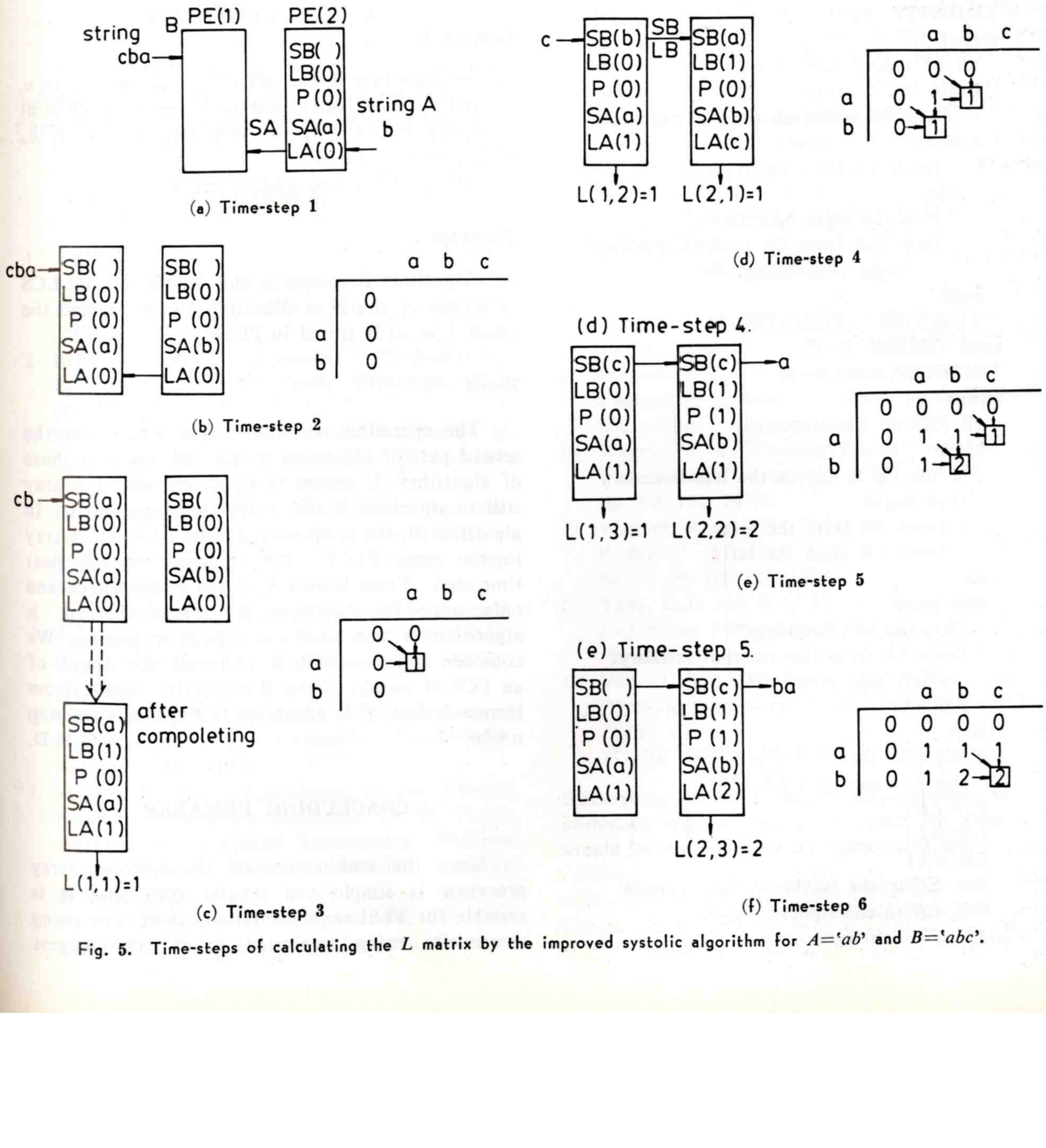

Fig. 5. Time-steps of calculating the L matrix by the improved systolic algorithm for A='ab' and B='abc'.

```
initial: String A is stored in the external memory
         on the right side of the linear array
         processor by the form:
             A='a1, a2,···, am'.
         String B is stored in the external memory
         on the left side of the linear array
         processor by the form:
             B='b1, b2,···, bn'.
         P, LA and LB in all PE's are set to zero.
         SA and SB in all PE's are set to a dummy
         symbol '@' which is not in the symbol set
         of strings A and B.
comments: In the first m time-steps, string A
          enters the linear array processor from
          the right end and string B stays still
          in the external memory.
          In the subsequent time-steps, string A
          stays still and string B enters the linear
          array processor from the left end.
final: The length of an LCS of strings A and B,
       L(m, n), is stored in PE(m).
 1: beginprogram
 2:   TIMESTEP:=m;
 3:   repeat
 4:     all PE's do simultaneously
 5:     begin
 6:       if the PE is not on the right boundary
 7:       then
 8:         fetch SA from the right
 9:       else
            /*on the right boundary*/
10:         fetch SA from the external memory
              /*right side, string A*/
11:     end;
12:     TIMESTEP:=TIMESTEP-1;
13:   until TIMESTEP=0;
14:   TIMESTEP:=m+n-1;
15:   repeat
16:     all PE's do simultaneously
17:     begin
18:       if the PE is not on the left boundary
19:       then begin
20:         fetch SB from the left;
21:         fetch LB from the left;
22:       end
23:       else begin
            /*on the left boundary*/
24:         fetch SB from the external memory;
              /*left side, string B*/
25:         LB:=0;
26:       end;
27:       if SA=SB then L:=P+1
28:         else L:=max(LA, LB);
29:       P:=LB;
30:       LA:=L;
31:       LB:=L;
32:       flow SB to the right;
33:       flow LB to the right;
34:     end;
35:     TIMESTEP:=TIMESTEP-1;
36:   until TIMESTEP=0;
37: endprogram;
```

In algorithm B, the total number of time-steps for completing the operations is $n+2m-1$, where m time-steps are needed to move string A to the linear array processor, n time-steps for all symbols of string B to enter the linear array processor, and $m-1$ time-steps for the last symbol of string B to arrive at the rightmost PE, i.e., to meet the last symbol of string A. The total number of PE's required is m. Both numbers of time-steps and PE's are smaller than those in algorithm A.

The main difference between algorithm A and algorithm B is that some PE's are useless in algorithm A at any time-step while no PE's are useless in algorithm B. The time-processor-product for algorithm B is $(n+2m-1)m$, which is smaller as compared with that for algorithm A.

We shall show how algorithm B works correctly in the following.

Lemma 4

In algorithm B, for all i, j, $1\leq i\leq m$, $1\leq j\leq n$, a_i and b_j meet once and only once in PE(i) at time-step $m+i+j-1$. At time-step $n+2m-1$, a_m and b_n meet in PE(m).

The proof is easy and is omitted here.

Theorem 2

Algorithm B computes the length of an LCS of strings A and B at time-step $n+2m-1$, and the result $L(m, n)$ is stored in PE(m).

Proof

The operations in lines 14-36 which are the second part of algorithm B are the same as those of algorithm A except that P, SA and LA stay still in algorithm B and move in algorithm A. In algorithm B, the temporary results P and LA stay in the same PE for the reference at the next time-step. From lemma 3, a_i and b_j meet once and only once in algorithm B. From theorem 1, algorithm A can compute $L(m, n)$ correctly. We conclude that algorithm B computes the length of an LCS of strings A and B correctly. Again, from lemma 3, $L(m, n)$ is generated in PE(m) at time-step $n+2m-1$. Q. E. D.

CONCLUDING REMARKS

Since the architecture of the systolic array processor is simple and regular such that it is suitable for VLSI implementation, there are many researchers being interested in it. The longest common subsequence problem, which we have discussed in this paper, can be easily solved by using the dynamic programming approach. Here, we combine the ideas of the systolic array processor and the dynamic programming approach to design our parallel algorithms.

We first solve this problem in a two-dimensional array processors by a straightforward scheme. In this scheme of two-dimensional array processors there is only one wavefront propagating and this wavefront never rolls back, as shown in Fig. 3. After the straightforward scheme for two-dimensional array processors is discovered, we map it to a linear array for reducing the number of processors used. Based upon this idea, we can generalize this mapping scheme for many problems. This general scheme has been discussed in Ref. [21]. We can apply the mapping scheme to obtain a systolic algorithm to recognize spoken words based upon the results in Refs. [14] and [15].

ACKNOWLEDGEMENT

This research work was partially supported by the National Science Council of the Republic of China under the contract NSC73-0201-E007-01.

REFERENCES

1. Apostolico. A. and A. Negro. "Systolic Algorithms for String Manipulations," *IEEE Trans. Comput.*, Vol. C-33, No. 4, pp. 361-364 (1984).
2. Bellman, R. E., *Dynamic Programming*, Princeton University Press, Princeton, N. J, (1957).
3. Foster, M. J. and H. T. Kung, "The Design of Special-Purpose VLSI Chips," *IEEE Comput.*, Vol. 13, No. 1, pp. 26-40 (1980).
4. Hirschberg, D. S., "A Linear Space Algorithm for Computing Maximal Common Subsequences," *Commun. ACM*, Vol. 18, No. 6, pp. 341-343 (1975).
5. Hirschberg, D. S., "Algorithms for the Longest Common Subsequence Problem," *J. Assoc. Comput. Mach.*, Vol. 24, No. 4, pp. 664-675 (1977).
6. Horowitz, E. and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Baltimore, Md. (1978).
7. Hsu, W. J., "*New Algorithms for the Longest Common Subsequence Problem*," Ph. D. Thesis, Institute of Computer Engineering, National Chiao Tung University, Hsinchu, Taiwan, R. O. C. (1983).
8. Hunt, J. W. and T. G. Szymanski, "A Fast Algorithm for Computing Longest Common Subsequences," *Commun. ACM*, Vol. 20, No. 5, pp. 350-353 (1977).
9. Hwang, K. and F. A. Briggs, *Computer Architecture and Parallel Processing*, McGraw-Hill, New York, N. Y. (1984).
10. Kung, H. T., "Notes on VLSI Computation," *Parallel Processing System*, Edited by Evans, D. J., Cambridge Univ. Press, New York, N. Y., pp. 339-356 (1982).
11. Kung, S. Y., K. S. Arun, R. J. Galezer, and D. V. B. Rao, "Wavefront Array Processor: Language, Architecture, and Applications," *IEEE Trans. Comput.*, Vol. C-31, No. 11, pp. 1054-1066 (1982).
12. Mead, C. A. and L. A. Conway, *Introduction to VLSI Systems*, Section 8.3, Addison-Wesley, Reading, MA (1980).
13. Moldovan, D. I., "On the Analysis and Synthesis of VLSI Algorithms," *IEEE Trans. Comput.*, Vol. C-31, No. 11, pp. 1121-1126 (1982).
14. Sakoe, H. and S. Chiba, "Dynamic Programming Algorithm Optimization for Spoken Word Recognition," *IEEE Trans. Acoust. Speech. Signal Process.*, Vol. ASSP-26, No. 1, pp. 43-49 (1978).
15. Sakoe, H., "Two-Level DP-Matching—A Dynamic Programming-Based Pattern Matching Algorithm for Connected Word Recognition," *IEEE Trans. Acoust. Speech. Signal Process.*, Vol. ASSP-27, No. 6, pp. 388-595 (1979).
16. Savage, C., "A Systolic Design for Connectivity Problems," *IEEE Trans. Comput.*, Vol. C-33, No. 1, pp. 99-103 (1984).
17. Sellers, P. H., "An Algorithm for the Distance Between Two Finite Sequences," *J. Combinatorial Theory (A)*, Vol. 16, pp. 353-258 (1984).
18. Ullman, J. D., *Computational Aspects of VLSI*, Chapter 5, Computer Science Press, Potomac, Md. (1984).
19. Wagner, R. A., "Common Phrases and Minimum-Text Storage," *Commun. ACM*, Vol. 16, No. 3, pp. 148-152 (1973).
20. Wagner, R. A., "The String-to-String Correction Problem," *J. Assoc. Comput. Mach.*, Vol. 21, No. 1, pp. 168-173 (1974).
21. Yang, C. B. and R. C. T. Lee, "The Mapping of 2-D Array Processors to 1-D Array Processors," *Parallel Comput.*, Vol. 3, pp. 217-229 (1986).
22. Yeh, C. S., I. S. Reed, and T. K. Truong, "Systolic Multipliers for Finite Field GF(2^m)," *IEEE Trans. Comput.*, Vol. C-33, No. 4, pp. 357-360 (1984).

Discussions of this paper may appear in the discussion section of a future issue. All discussions should be submitted to the Editor-in-Chief.

Manuscript Received: December 24, 1986;
Revision Received: July 19, 1987;
Accepted: July 20, 1987.