Parallel Algorithms for the Longest Common Subsequence Problem

Mi Lu, *Senior Member, IEEE*, and Hua Lin

***Abstract*— A subsequence of a given string is any string obtained by deleting none or some symbols from the given string. A longest common subsequence of two strings is a common subsequence of both that is as long as any other common subsequences. The longest common subsequence problem is to find the longest common subsequence of two given strings. The bound on the complexity of this problem under the decision tree model is known as mn if the number of distinct symbols that can appear in strings is infinite, where m and n are the lengths of the two strings, respectively, and $m \leq n$. In this paper, we propose two parallel algorithms for this problem on the CREW-PRAM model. One takes $O(\log^2 m + \log n)$ time with $mn/\log m$ processors, which is faster than all the existing algorithms on the same model. The other takes $O(\log^2 m \log\log m)$ time with $mn/\log^2 m \log\log m$ processors when $\log^2 m \log\log m > \log n$, or otherwise $O(\log n)$ time with $mn/\log n$ processors, which is optimal in the sense that the *time* $\times$ *processors* bound matches the complexity bound of the problem. Both algorithms exploit nice properties of the LCS problem that are discovered in this paper.**

***Index Terms*— Concurrent-read exclusive-write parallel random-access machine (CREW-PRAM), grid directed graph, longest common subsequence, maximum-cost path, parallel algorithm, totally monotone array**

I. INTRODUCTION

A STRING is a sequence of symbols. Given a string, a *subsequence* of the string can be obtained from the string by deleting none or some symbols, but 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 both and is as long as any other common subsequences. For example, string "*like*" is the longest subsequence of strings "*kliuke*" and "*allaiiakeu*." In general, there may exist more than one longest subsequences for two strings. Given two strings A and B with length m and n, $m \leq n$, respectively, the LCS problem is to identify a longest common subsequence of A and B. Fast solutions for this problem are requested very often in genetic engineering, data compression, editing error correction, and syntactic pattern recognition [1], [5], [13].

The lower bound on the time complexity of this problem has been studied by Aho *et al.* in [1]. They have shown that under the decision tree model, in which all decisions concern whether two positions have the same symbol, the time complexity of the LCS problem is mn if the number of distinct symbols that can appear in the strings is infinite. Sequential algorithms matching this bound can be found, among others, in [8] and [9].

In recent years, exploiting the parallelism of this problem attracts many research interests and several parallel algorithms have been designed [2], [3], [11], [12]. Among them, Aggarwal and Park [2] and Apostolico *et al.* [3] have independently shown that this problem (and a more general problem called the *string-editing* problem) can be solved in $O(\log m \log n)$ time using $mn/\log m$ processors on the CREW-PRAM model, on which concurrent reads are allowed, but on which no two processors can simultaneously attempt to write in the same memory location. Their algorithms share the following basic idea: Relate the string editing problem to the problem of recognizing the shortest path from the source to the sink on a grid-directed graph. To identify the path, they use a divide-and-conquer scheme to compute the "distance matrix," which records the minimum lengths from every vertex on the left (or top) boundary of the grid-directed graph to every vertex on the bottom (or right) boundary. These two algorithms differed in the "conquer" stage. In [2], an efficient technique for searching in a *totally monotone* array has been applied, whereas in [3], the *cascading divide-and-conquer* scheme has been used. These two results have been the best known ones in terms of the time bound. However, in terms of the *time* $\times$ *processors* bound, none of them matches the bound mn.

Similarly to [2] and [3], this paper relates the LCS problem to the problem of finding the maximum-cost path on a grid directed graph. However, by defining a totally new concept for "cost matrix," we exploit very nice properties of the LCS problem. Two fast algorithms are developed both on CREW-PRAM model. The first algorithm task $O(\log^2 m + \log n)$ time with $mn/\log m$ processors, which is faster than any existing algorithms on the same model. (Remember that $m \leq n$.) The second algorithm takes $O(\log^2 m \log m)$ time with $mn/\log^2 m \log\log m$ processors when $\log^2 m \log\log m > \log n$; otherwise, it takes $O(\log n)$ time with $mn/\log n$ processors, which is optimal in the sense that the *time* $\times$ *processors* bound matches the complexity bound of the problem.

The remainder of this paper is organized as follows. Section II shows how the LCS problem can be viewed as the maximum-cost path problem on a grid-directed graph, and provides an overview of our algorithms. Section III

Manuscript received May 27, 1991; revised October 29, 1993, and January 31, 1994. This work was supported in part by the National Science Foundation (NSF) under Grant MIP-8809328, and in part by the Texas Advanced Technology Program.

The authors are with the Department of Electrical Engineering, Texas A&M University, College Station, TX 77843-3128 USA; e-mail: mlu@ee.tamu.edu.

IEEE Log Number 9401205.

1045-9219/94$04.00

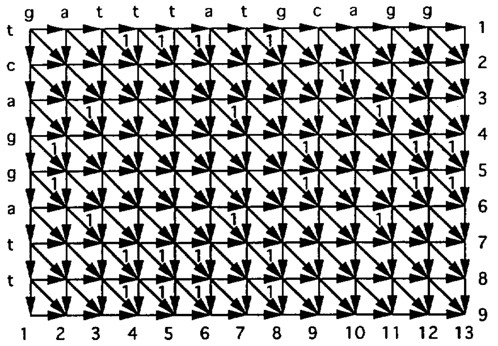


Fig. 1. The grid DAG associated with strings tcaggatt and gatttatgcagg.

concentrates on exploiting properties of the LCS problem. Section IV gives the faster algorithm, and Section V presents the optimal algorithm.

II. SOLVING THE LCS PROBLEM THROUGH GRID DAG

A. The LCS Problem via Grid-Directed Acyclic Graph

An $l_1 \times l_2$ *grid* directed acyclic graph (DAG) is a DAG whose vertices are the $l_1 \times l_2$ grid points of an $l_1 \times l_2$ grid. The only edges from vertex (i, j), the grid point on the ith row and the jth column, are to vertices $(i, j+1), (i+1, j)$ and $(i+1, j+1)$. Sometimes they are referred to as horizontal, vertical, and diagonal edges, respectively. Vertex (1, 1) is the *source*, and vertex (l_1, l_2) is the *sink*. Given an instance of the LCS problem, *i.e.*, string $A = a_1, a_2, \cdots, a_m$ and string $B = b_1, b_2, \cdots, b_n$, the grid DAG, G, *associated with* A and B is an $(m+1) \times (n+1)$ grid DAG such that each diagonal edge on G, say, from vertex (i, j) to vertex $(i+1, j+1)$, is associated with *cost* 1 if symbol a_i and symbol b_j in A and B are identical, and otherwise associated with cost 0. The cost of a path on G is defined as the sum of costs on the path. A *maximum-cost* path is the one with the maximum cost. Throughout we presume that m, the length of A, is a power of 2. As an example, Fig. 1 shows the grid DAG associated with strings *tcaggatt* and *gatttatgcagg*. The relation between the LCS problem and the maximum-cost path problem is seen as follows.

Observation 1: Any path with cost l on grid DAG G associated with strings A and B corresponds to a CS with length l of A and B. In particular, the maximum-cost path between the source and the sink corresponds to the LCS of A and B.

So, to solve the LCS problem, we need to find only the maximum-cost path beginning at the source and ending at the sink on grid DAG G. To find this path, we are actually to find the maximum-cost paths from every vertex on the top row to every vertex on the bottom row. Similar to the previous research [2], [3], those paths will be identified under a divide-and-conquer scheme. We divide the $(m+1) \times (n+1)$ grid DAG, G into two $(m/2+1) \times (n+1)$ grid DAG's, the upper half, G_U and the lower half, G_L, and then find the maximum-cost paths on G_U and G_L in a recursive fashion.

$$D_{G_U} = \begin{pmatrix} 2 & 7 & 9 & 12 \\ 3 & 7 & 9 & 12 \\ 4 & 7 & 9 & 12 \\ 5 & 7 & 9 & 12 \\ 6 & 7 & 9 & 12 \\ 7 & 9 & 11 & 12 \\ 8 & 9 & 11 & 12 \\ 9 & 11 & 12 & \infty \\ 10 & 11 & 12 & \infty \\ 11 & 12 & \infty & \infty \\ 12 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{pmatrix} \qquad D_{G_L} = \begin{pmatrix} 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & \infty \\ 4 & 5 & \infty & \infty \\ 5 & 6 & \infty & \infty \\ 6 & 8 & \infty & \infty \\ 7 & 8 & \infty & \infty \\ 8 & 11 & \infty & \infty \\ 9 & 11 & \infty & \infty \\ 11 & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty \\ 13 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty \end{pmatrix}$$

Fig. 2. Examples of cost matrix.

A vertex v on the bottom row is *the jth breakout vertex with respect to vertex* $(1, i)$ if v is the leftmost vertex on the bottom row, such that there is a path of cost j from vertex $(1, i)$ to v. Sometimes we call v a breakout vertex of vertex $(1, i)$ for short. In Fig. 1, vertices (9, 2), (9, 3), (9, 4), (9, 5), and (9, 13) are the first, second, third, fourth, and fifth breakout vertices of the source. Note that there are no fifth breakout vertices with respect to some vertices, for example, (1, 8), because the maximum cost from vertex (1, 8) to the bottom row is 4.

A fact about breakout vertices is this: The maximum-cost path between a vertex, say, v, on the top row and its jth breakout vertex, say w, on the bottom row on G must have cost j, if v does have the jth vertex. This is because all of the cost 1's appear on diagonal edges only. Indeed, if there is a path between vertex v and w with cost greater than j, then vertex w must not be a jth breakout vertex of v, because we can always find a vertex w' to the left of w such that there exists a path between v and w' with a cost of j. In general, the maximum-cost path between two vertices is not unique. Throughout this paper, we are interested only in the leftmost one among the maximum-cost paths between two vertices, in the sense that no vertices on the other paths lie to its left.

The maximal possible cost of a path on an $(m+1) \times (n+1)$-grid DAG is m. Hence, any vertex on the top row of the grid DAG has at most m number of breakout vertices on the bottom row. The information about breakout vertices can therefore be stored in an $n \times m$ matrix called *cost matrix*. A cost matrix associated with grid DAG G, denoted by D_G, is defined as follows: For $1 \leq i \leq n$ and $1 \leq j \leq m$, $D_G(i, j) = k$ if vertex $(m+1, k)$ is the jth breakout vertex of vertex $(1, i)$, and $D_G(i, j) = \infty$ if vertex $(1, i)$ does not have a jth breakout vertex. Note that an entry in D_G is really not a cost, but rather the location of a breakout vertex on the bottom row of G. By D_G^i we denote the ith row of D_G. Fig. 2 shows the cost matrices associated with G_U, the upper half of G shown in Fig. 1, and G_L, the lower half of G.

B. The Main Structure of Our Algorithms

Now we give an overview of our algorithms. Basically, both algorithms consist of four phases.

1) Compute D_{G_i} for $1 \leq i \leq m$, where G_i is a $2 \times (n+1)$-grid DAG consisting of the ith and $(i+1)$th rows of G.
2) Recursively compute D_G from D_{G_U} and D_{G_L}.

3) Identify vertices on the maximum-cost path between the source and the sink on G.
4) Identify the LCS that corresponds to the maximum-cost path.

Both algorithms that we designed for the LCS problem share the same implementations of Phase 1 and Phase 4, but differ in their implementations of Phase 2 and Phase 3. Phase 1 and Phase 4 are simple, so a short description on them is provided in the next subsection. The implementations for Phase 3 will be stated later in Sections IV and V. The implementations of Phase 2, which are critical for both algorithms, are quite complicated. We provide some basic ideas in Section II-D, and leave details to Sections IV and V.

C. The Implementations of Phase 1 and Phase 4

Our basic strategy for the LCS problem is divide-and-conquer. Given two strings A and B, to compute cost matrix D_G of G associated with A and B, initially we need to compute m cost matrices, with each being associated with a $2 \times (n+1)$-grid DAG as a base for "merge." Let us now discuss the computation of the cost matrix of such a grid DAG, say, G_h. Suppose G_h consists of the hth and $(h+1)$th rows of G. Let $b_{j_1}, b_{j_2}, \cdots, b_{j_r}$ the j_1th, j_2th,$\cdots$, j_rth symbols in B, be all symbols identical to a_h, the hth symbols in A, where $j_1 < j_2 < \cdots < j_r$. The following facts are apparent, according to the definition of G_h. Any vertex on the top row of G_h has at most one breakout vertex; any vertex properly to the right side of vertex $(1, j_r)$, the j_rth vertex on the top row of G_h, has no breakout vertex at all; and, finally, any other vertex $(1, j)$, $1 \leq j \leq j_r$, has a breakout vertex $(2, j_k+1)$, the (j_k+1)th vertex on the bottom row of G_h, where j_k satisfies $j_{k-1} < j < j_k$ (j_0 is defined as 0). In other words, the values of entries from $D_{G_h}(1,1)$ to $D_{G_h}(j_1, 1)$ are j_1+1, the values of entries from $D_{G_h}(j_1+1, 1)$ to $D_{G_h}(j_2, 1)$ are j_2+1, and so on. For those entries $D_{G_h}(j, 1)$ where $j_r < j \leq n$, the value is ∞. As an example, for grid DAG G shown in Fig. 1, because $a_1 = t$ and $b_3 = b_4 = b_5 = b_7 = t$, we have $D_{G_1} = (4,4,4,5,6,8,8,\infty,\infty,\infty,\infty,\infty)^T$.

$j_1, j_2, \cdots, j_r$ can be identified by sorting symbols of B, which can be done in $O(\log n)$ time with n processors [6]. $j_1, j_2, \cdots, j_r$, or ∞ can then be properly assigned to the entries of D_{G_h} by the procedure below. D_{G_h} can be generated in $O(\log n)$ time by using $n/\log n$ processors; therefore, Phase 1 can be done in $O(\log n)$ time by using $mn/\log n$ processors, for there are m such matrices to be computed in total.

The **procedure** for generating D_{G_h} is as follows.

1) Assign $j_k - j_{k-1}$ to $D_{G_h}(j_{k-1}+1, 1)$, for $1 < k \leq r$, and assign j_1+1 to $D_{G_h}(1,1)$.
2) Compute the prefix for the entries of D_{G_h} from $D_{G_h}(1,1)$ to $D_{G_h}(j_r+1, 1)$ (refer to [7]), that is $D_{G_h}(k,1) \leftarrow \sum_{j=1}^{k} D_{G_h}(j,1)$, for $1 \leq k \leq j_r+1$.
3) Assign ∞ to entries of D_{G_h} from $D_{G_h}(j_r+2, 1)$ to $D_{G_h}(n, 1)$.

Now we turn to Phase 4. In Phase 4, we trace the maximum-cost path $p = \langle v_1, v_2, \cdots, v_l \rangle$ obtained in Phase 3 from the source to the sink, and check the cost on each edge $e = (v_k, v_{k+1})$. Symbol a_i in A is to be marked if we find that edge e has a cost of 1 and vertex v_k has column index i. The LCS of A and B that corresponds to p can be obtained by ranking those marked symbols. Since the number of edges on p is bounded by $n+m$, and because checking the cost on an edge takes constant time, marking symbols in A can be done in constant time with n processors or in $O(\log n)$ time with $n/\log n$ processors. The ranking job can be done in $O(\log n)$ time with $n/\log n$ processors using a standard technique [7]. Thus $O(\log n)$ time using $n \log n$ processors suffices for Phase 4.

D. Ideas for Implementing Phase 2

A cost matrix contains all information about the costs of the maximum-cost paths between vertices on the top row and vertices on the bottom row of a grid DAG, which allows us to compute D_G, given D_{G_U} and D_{G_L}. Before proposing a formula for computing D_G from D_{G_U} and D_{G_L}, we would like to first examine their relations through the grid DAG.

Consider vertex $(1, i)$, its jth breakout vertex, say, $(m+1, i_v)$, and the maximum-cost path, say, p, between them. Clearly, the cost of p is j, and i_v is the value of entry $D_G(i,j)$. Path p intersects the common boundary of G_U and G_L at some vertex, say, $(m/2+1, i_q)$. Thus, vertex $(m/2+1, i_q)$ partitions path p into two subpaths, say, p_1 and p_2: Path p_1 goes from vertex $(1, i)$ to $(m/2+1, i_q)$ with certain cost, say, k, and path p_2 goes from $(m/2+1, i_q)$ to $(m+1, i_v)$ with cost $j-k$. Since p is a leftmost path (we are interested in only the leftmost paths), each of p_1 and p_2 must be a leftmost path also. Consequently, vertex $(m/2+1, i_q)$ on G is the kth breakout vertex of $(1, i)$ on G_U, whereas vertex $(m+1, i_v)$ on G is the $(k-j)$th breakout vertex of $(1, i_q)$ on G_L, respectively. In other words, $D_{G_U}(i,k) = i_q$ and $D_{G_L}(i_q, k-j) = i_v$, when $k \neq 0$ and $k \neq j$. From these two equations, plus $D_G(i,j) = i_v$, we have $D_G(i,j) = D_{G_L}(D_{G_U}(i,k), k-j)$. When $k = 0, p_1$ must go straight down from $(1, i)$ to $(m/2+1, i_q)$ (again, this is because p is a leftmost path); thus, $i_q = i$. Consequently, p_2, with a cost of j, must be the maximum-cost path from $(m/2+1, i)$ to $(m+1, i_v)$, implying that $D_{G_L}(i,j) = i_v$. Therefore, we have $D_G(i,j) = D_{G_L}(i,j)$. Similarly, one can prove that when $j = k$, we have $D_G(i,j) = D_{G_U}(i,j)$.

Theorem 1: For $1 \leq i \leq n$ and $1 \leq j \leq m$, we have the following condition:

$$D_G(i,j) = \min_{1 \leq k \leq j} \{D_{G_U}(i,j), D_{G_L}(i,j), D_{G_L}(D_{G_U}(i,k), j-k)\},$$

where both $D_{G_U}(i,j)$ and $D_{G_L}(i,j)$ are defined as ∞ for $j > m/2$, and $D_{G_L}(D_{G_U}(i,k), j-k)$ is also defined as ∞ for $D_{G_U}(i,k) = \infty$.

As an application of Theorem 1, we can calculate $D_G(1,3)$ from D_{G_U} and D_{G_L}, shown in Fig. 2, as follows.

$$D_G(1,3) = \min\{D_{G_U}(1,3), D_{G_L}(1,3), D_{G_L}(D_{G_U}(1,1), 2), D_{G_L}(D_{G_U}(1,2), 1)\} = 4.$$

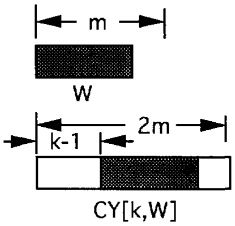


Fig. 3. Illustrating the concept of k-copy.

Proof: The correctness of Theorem 1 in the normal case in which the jth breakout vertex of vertex $(1, i)$ exists has been shown through the above discussion. Now we shall show that this theorem is also correct when vertex $(1, i)$ does not have a jth breakout vertex. To do this, it suffices to show that $D_G(i, j)$ will be assigned ∞ under this circumstance.

Obviously, when vertex $(1, i)$ does not have a jth breakout on G, neither vertex $(1, i)$ on G_U nor vertex $(1, i)$ on G_L has its jth breakout vertex. So, the first two items, $D_{G_U}(i, j)$ and $D_{G_L}(i, j)$, in the above formula must be ∞, which leaves us to show that $D_{G_L}(D_{G_U}(i, k), j - k) = \infty$ for $1 \leq k \leq j$. For a contradiction, we assume the existence of $k, 1 \leq k \leq j$, such that $D_{G_L}(D_{G_U}(i, k), j - k) = i_1$ and i_1 is finite. Recalling the definitions in Theorem 1, we see that $D_{G_U}(i, k)$ must be finite. Let $D_{G_U}(i, k) = i_2$. Then $D_{G_L}(i_2, j - k) = i_1$. In other words, there exist two paths on G. One has cost k and goes from vertex $(1, i)$ to $(m/2 + 1, i_2)$, and the other has cost $j - k$ and goes from vertex $(m/2 + 1, i_2)$ to $(m + 1, i_1)$. A path on G with cost j thus can be obtained by combining these two paths. The existence of such a path contradicts the fact that vertex $(1, i)$ does not have a jth breakout vertex. ☐

A trivial but inefficient algorithm for computing D_G is to apply Theorem 1 directly. Indeed, computing entry $D_G(i, j)$ of D_G from D_{G_U} and D_{G_L} is nothing more than identifying the minima among $O(m)$ entries, which can be done in $O(\log m)$ time using $m/\log m$ processors. However, because there are in total $n \times m$ entries of D_G to be computed, $nm^2/\log m$ processors are needed in order to complete the computation in $O(\log m)$ time, which implies a computational time of $O(\log^2 m)$ for generating D_G with $nm^2/\log m$ processors. We must apply Theorem 1 in a much more efficient way. A better-organized form of this theorem is proposed below.

A *k-copy* of a row-vector W of size m is a row-vector of size $2m$, denoted as $CY[k, W]$ for some k between 1 and m, such that entries of $CY[k, W]$ from $CY[k, W](k)$ to $CY[k, W](m + k - 1)$ are copies of the entries of W from $W(1)$ to $W(m)$, and other entries of $CY[k, W]$ hold ∞ (see Fig. 3). Given D_{G_U} and D_{G_L}, the $n \times m/2$ cost matrices associated with G_U and G_L, we define n matrices $M[D_G^i]$, for $1 \leq i \leq n$, as follows (see also Fig. 4).

1) The size of $M[D_G^i]$ is $l \times m$, where l is the number of breakout vertices of vertex $(1, i)$ on G_U.
2) For $1 \leq j \leq l$, the jth row of $M[D_G^i]$ is $CY[j + 1, D_{G_L}^{D_{G_U}(i,j)}]$ (here $D_{G_L}^{D_{G_U}(i,j)}$ is the tth row of D_{G_L} with $t = D_{G_U}(i, j)$).

As an example, let D_{G_U} and D_{G_L} be two matrices shown in Fig. 2. Fig. 5 shows matrices $M[D_G^1]$ and $M[D_G^2]$, respectively. Using matrix $M[D_G^i]$, Theorem 1 can be rewritten as follows.

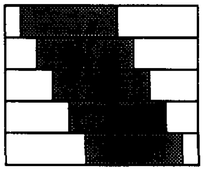


Fig. 4. Illustrating the structure of $M[D_G^i]$.

Corollary 1: For $1 \leq i \leq n$ and $1 \leq j \leq m$, we have the following condition:

$$D_G(i, j) = \min_{1 \leq k \leq l} \{D_{G_U}(i, j), D_{G_L}(i, j), M[D_G^i](k, j)\},$$

where l is the number of breakout vertices of vertex $(1, i)$ on G_U.

The efficient use of this corollary is addressed in Section IV and Section V, after we exploit some nice properties of D_G and $M[D_G^i]$.

III. PROPERTIES OF D_G AND $M[D_G^i]$

The objective of this section is to illustrate the information redundancy in D_G and $M[D_G^i]$, called *k-variant*, as well as another property of $M[D_G^i]$, called *totally monotone*. Let us start with exploiting properties of cost matrix D_G.

A. Properties of D_G

Here are some simple facts about D_G, proofs of which can be found in [10].

Proposition 1:

1) $D_G(i, j_1) < D_G(i, j_2)$ if $j_1 < j_2$ and $D_G(i, j_1) \neq \infty$.
2) $D_G(i_1, j) \leq D_G(i_2, j)$ if $i_1 < i_2$.
3) $D_G(i + 1, j) \leq D_G(i, j + 1)$.
4) If $D_G(i_1, j_1) = k$ and $k \neq \infty$, then there exists j_2 such that $D_G(i_2, j_2) = k$ for any i_2, $i_1 < i_2 < k$.
5) If $D_G(i_1, j_1) = D_G(i_2, j_2) = k_1$, then there exists j such that $D_G(i, j) = k_1$ for any i, $i_1 < i < i_2$.

The importance of Propositions 1(4) and 1(5) is that they suggest the similarity between rows in D_G. Moreover, since rows in matrix $M[D_G^i]$ are nothing but copies of rows in D_{G_L}, they also suggest the similarity among $M[D_G^i]$'s. The following definitions are helpful in formalizing the information redundancy in D_G and in $M[D_G^i]$. Given a row-vector, *a sub-row-vector* of it is obtained from it by deleting none or some entries (not necessarily consecutive ones). If a row-vector is a sub-row-vector of more than two row-vectors, say, $W_1, W_2, \cdots, W_l$, then it is a *common sub-row-vector* of them. (Without causing misunderstanding, we simply call it a common row-vector.) Those W_i's, each with size m, are *k-variant* if there exists a common row-vector of them such that the size of it is at least $m - k$. For example, any two consecutive row-vectors in D_{G_U} and D_{G_L} shown in Fig. 2 are 1-variant.

$$M[D_G^1] = \begin{pmatrix} \infty & 3 & 4 & 5 & \infty & \infty & \infty & \infty \\ \infty & \infty & 8 & 11 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 13 & \infty & \infty & \infty \end{pmatrix}, \quad M[D_G^2] = \begin{pmatrix} \infty & 4 & 5 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 8 & 11 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 13 & \infty & \infty & \infty \end{pmatrix}.$$

Fig. 5. $M[D_G^1]$ and $M[D_G^2]$.

Proposition 2:

1) D_G^i and D_G^{i+1}, the ith and $(i+1)$th row of D_G, are 1-variant for $1 \le i \le n$.
2) Any $k+1$ consecutive rows of D_G are k-variant.

Proof: Proposition 2(2) is an immediate result of Proposition 2(1), so we prove only the latter. Without loss of generality, we assume that ∞ exists in both D_G^i and D_G^{i+1}. We also assume that there are two numbers, l_1 and l_2, such that $D_G(i, l_1) \ne \infty$, $D_G(i, l_1+1) = \infty$, $D_G(i+1, l_2) \ne \infty$, and $D_G(i+1, l_2+1) = \infty$. To show that D_G^i and D_G^{i+1} are 1-variant, it suffices to show that each of these two rows contains at most one "finite" entry that is not in the other. Noticing that $l_1 \ge l_2$ (suggested by Proposition 1(2)), we actually need to show only that there exists at most one finite entry that is in D_G^i, but not in D_G^{i+1}. Three cases need to be considered, depending on the first entry in D_G^i and D_G^{i+1}: $D_G(i,1) = D_G(i+1,1)$, $D_G(i,1) < D_G(i+1,1)$, and $D_G(i,1) > D_G(i+1,1)$.

The third case could not occur, according to Proposition 1(2). Proposition 2(1) is held in the first case. In fact, D_G^i is identical to D_G^{i+1} when $D_G(i,1) = D_G(i+1,1)$. Indeed, for instance, if $D_G(i,2)$ is not ∞, then there exists j_2 such that $D_G(i+1, j_2) = D_G(i,2)$ according to Proposition 1(4), where j_2 must be less than or equal to 2. On the other hand, j_2 should not be 1, because $D_G(i+1,1) = D_G(i,1) \ne D_G(i,2)$. Therefore, we conclude that $j_2 = 2$; *i.e.*, $D_G(i+1,2) = D_G(i,2)$. Similarly, one can show that if $D_G(i,3)$ is not ∞, then $D_G(i+1,3) = D_G(i,3)$, and so on. As for the second case, except for $D_G(i,1)$, all entries in D_G^i must also be in D_G^{i+1}. Indeed, since $i+1 \le D_G(i,1) < D_G(i,j)$ for $2 \le j \le l_1$ (by Proposition 1(1)), entry $D_G(i,j)$ can be found in D_G^{i+1} according to Proposition 1(4). □.

To structurize the useful information in D_G and in $M[D_G^i]$, we introduce several concepts. Consider $k+1$ consecutive row-vectors in D_G, say, D_G^i for $i_1 \le i \le i_1 + k$. Let L be a common row-vector of them. L can be partitioned into *groups*, $L_1, L_2, \cdots, L_r$, such that entries in each group are consecutive entries in every D_G^i. The *remnant* of D_G^i with respect to L (the remnant of D_G^i for short) is a row-vector $R[D_G^i] = (R_1^i, R_2^i, \cdots, R_{r+1}^i)$ such that $D_G^i = (R_1^i, L_1, R_2^i, \cdots, L_r, R_{r+1}^i)$. Note that there may be no entry in R_j^i. The size of $R[D_G^i]$ is defined as the sum of sizes of R_j^i's for $1 \le j \le r+1$. In this paper, we are interested in only the largest partition, in the sense that for any two entries from two distinct groups, there exists i', $i_1 \le i' \le i_1 + k$, such that these two entries are not consecutive in $D_G^{i'}$. Clearly, under this assumption, the partition for groups are unique. Consider the first and second rows of D_{G_U} in Fig. 2 as an example. The common row-vector of them is $L = (7, 9, 12)$, and the remnants of them are $R[D_{G_U}^1] = (2)$ and $R[D_{G_U}^2] = (3)$, respectively.

Proposition 3:

1) For $k+1$ consecutive row-vectors D_G^i of D_G, $i_1 \le i \le i_1 + k$, common row-vector L of $D_G^{i_1}$ and $D_G^{i_1+k}$ is a common row-vector of the $k+1$ row-vectors.
2) L can be partitioned into at most $2k$ groups.
3) Let $R[D_G^i] = (R_1^i, \cdots, R_{r+1}^i)$ be the corresponding remnant of D_G^i, where $r > 0$. Then $R_j^{i_1}$ is a subvector of $R_j^{i_2}$ when $i_1 < i_2$ and $2 \le j \le r+1$.

Proof: To show that L is the common row-vector of D_G^i's, for $i_1 \le i \le i_1 + k$, we need to show only the following:

1) Any "finite" entry in L appears in every D_G^i.
2) The number of "∞" entries in L are no more than that in any D_G^i's.

Statement 1) is true because if, for instance, w is a "finite" entry in L, i.e., there exist j_1 and j_2 such that $D_G(i_1, j_1) = D_G(i_1 + k, j_2) = w$, then for any i, $i_1 < i < i_1 + k$, there exists j such that $D_G(i, j) = w$ (by Proposition 1(5)). Statement 2) is true because, in fact, Proposition 1(2) suggests that there are at least as many ∞ entries in D_G^i as in $D_G^{i_1}$ for $i_1 < i$.

L can be obtained from either D^{i_1} or D^{i_1+k} by deleting at most k entries; therefore, it should be of no problem to partition L into no more than $2k$ groups such that entries in each group correspond to consecutive entries in both $D_G^{i_1}$ and $D_G^{i_1+k}$. Now, to show Proposition 3(2), we shall show that the partition is valid for D_G^i's for $i_1 < i < i_1 + k$. That is, if $w_1, w_2, \cdots, w_h$ are h consecutive entries in both $D_G^{i_1}$ and $D_G^{i_1+k}$ under this partition, then they are also consecutive in any D_G^i, for $i_1 < i < i_1 + k$. The existence of those entries in D_G^i's is certain by Proposition 1(5). To show the consecutivity, we assume that there exists entry w in D_G^i such that $w_l < w < w_{l+1}$, $1 \le l < h$, for a contradiction. Since w_l is an entry in $D_G^{i_1+k}$, we have $i_1 + k < w_l$. Because of $i < i_1 + k, i_1 + k < w_l$, and $w_l < w$, we have $i < i_1 + k < w$. By Proposition 1(4), w must be an entry also in $D_G^{i_1+k}$. The assumption that w is between w_l and w_{l+1}, thus implying that w_l and w_{l+1} are not consecutive in $D_G^{i_1+k}$, a contradiction. The proof of Proposition 3(3) is similar to the proof of Proposition 3(1).

The following theorem is an immediate result from Proposition 3.

Theorem 2: Each row-vector D_G^i of any $k+1$ consecutive rows of D_G can be represented by a common row-vector of these consecutive rows and a remnant such that the common row-vector consists of at most $2k$ groups, and the remnant contains at most k entries.

B. Properties of $M[D_G^i]$

The information redundancy in matrix $M[D_G^i]$, along with another important property of $M[D_G^i]$, called *totally monotone*, is illustrated in this section.

Let $X = (X^1, X^2, \cdots, X^m)$ and $Y = (Y^1, Y^2, \cdots, Y^m)$ be two matrices of the same size, where X^i and Y^i are the ith columns of X and Y, respectively. *X is obtained from Y by k-shift*, denoted by $X = S[k, Y]$, if there is a number k such that $X = (Y^{m-k+1}, \cdots, Y^m, Y^1, \cdots, Y^{m-k-1}, Y^{m-k})$. Let $Y_1, Y_2, \cdots, Y_q$ be matrices, all with same number of columns. By $(Y_1, Y_2, \cdots, Y_q)^T$, we denote the following matrix in this paper: $\begin{pmatrix} Y_1 \\ Y_2 \\ \cdots \\ Y_q \end{pmatrix}$. Matrix $M = (M_1, \cdots, M_r)^T$, where M_i's are matrices each with m columns, is a *common matrix* of l matrices $A_1, A_2, \cdots, A_l$, each with m columns also, if there exists an integer $K(i, j)$ such that for any A_i, $1 \le i \le l$, not only is the new matrix $(S[K(i,1), M_1], S[K(i,2), M_2], \cdots, S[K(i,r), M_r])^T$ a submatrix of A_i, but also the rows in $S[K(i,j), M_j]$ are consecutive rows in A_i. Matrix M_j in M, $1 \le j \le r$, is defined as the *jth group* of M. For example, the following matrix is a common matrix of $M[D_G^1]$ and $M[D_G^2]$ in Fig. 5:

$$\begin{pmatrix} \infty & \infty & 8 & 11 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 11 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 13 & \infty & \infty & \infty \end{pmatrix}.$$

In this example, $r = 1$ and $K(1,1) = K(2,1) = 0$. Intuitively, given $A_1, A_2, \cdots, A_l$, the larger the size of M, the more $A_1, A_2, \cdots, A_l$ look alike. $A_1, A_2, \cdots, A_l$ are *k-variant* if there exists a common matrix M of them such that for each A_i, at most k rows of it are not in M. For example, matrices $M[D_G^1]$ and $M[D_G^2]$ in Fig. 5 are 1-variant.

Theorem 3: If $D_{G_U}^{i_1}$ and $D_{G_U}^{i_2}$ are k-variant, then $M[D_G^{i_1}]$ and $M[D_G^{i_2}]$ are k-variant. Moreover, if $L = (L_1, L_2, \cdots, L_r)$ is a common row-vector of $D_{G_U}^{i_1}$ and $D_{G_U}^{i_2}$, where L_j is the jth group of L, then a common matrix, $(M_1, M_2 \cdots, M_r)^T$, of $M[D_G^{i_1}]$ and $M[D_G^{i_2}]$ can be constructed as follows: The ith row of M_j is either $CY[i, D_{G_L}^{L_j(i)}]$ when $L_j(i) \neq \infty$ or a row of ∞'s when $L_j(i) = \infty$.

Proof: Two things need to be proved. First, $(M_1, M_2, \cdots, M_r)^T$ is a common matrix of $M[D_G^{i_1}]$ and $M[D_G^{i_2}]$; second, there are at most k rows in each of $M[D_G^{i_1}]$ and $M[D_G^{i_2}]$ that are not in the common matrix. We discuss only $M[D_G^{i_1}]$ in this proof; $M[D_G^{i_2}]$ can be handled similarly. For the sake of simplicity, we assume that there is no ∞ entry in L; the case in which ∞ entry exists should not be difficult to deal with.

To prove the first claim, we shall prove that there exists $K(i_1, j)$ such that $S[K(i_1, j), M_j]$ is a submatrix of $M[D_G^{i_1}]$, where $1 \le j \le r$. Remember that matrix M_j is decided by L_j, together with D_{G_L}, as defined in the theorem. Let r_j be the size of L_j, and let $L_j(1)$, the first entry in L_j, be the w_1th and w_2th entry in $D_{G_U}^{i_1}$ and $D_{G_U}^{i_2}$, respectively. We shall show that the submatrix, which consists of r_j number of rows of $M[D_G^{i_1}]$, from w_1th row to $(w_1 + r_j - 1)$th row, can be obtained from M_j by w_1-shift. Because of the lack of space, we explain only why the w_1th row of $M[D_G^{i_1}]$ can be obtained from the first row of M_j by w_1-shift. The other rows are handled similarly. Indeed, by the definition of $M[D_G^{i_1}]$, the w_1th row of $M[D_G^{i_1}]$ is $CY\left[w_1 + 1, D_{G_L}^{D_{G_U}(i_1, w_1)}\right]$. On the other hand, the first row of M_j is $CY\left[1, D_{G_L}^{L_j(1)}\right]$, as defined in the theorem, which is equivalent to $CY\left[1, D_{G_L}^{D_{G_U}(i_1, w_1)}\right]$, because $L_j(1) = D_{G_U}(i_1, w_1)$. Comparing these two copies of $D_{G_L}^{D_{G_U}(i_1, w_1)}$, one can see that the former is obtained from the latter by the w_1-shift.

To prove that at most k rows of $M[D_G^{i_1}]$ are not in $(M_1, M_2, \cdots, M_r)^T$, we need to notice only the fact that there are at most k entries that are in $D_{G_U}^{i_1}$, but not in L. □

Corollary 2: Any $k + 1$ consecutive matrices, $M[D_G^i]$, $M[D_G^{i+1}]$, $\cdots$, $M[D_G^{i+k}]$, for $1 \le i \le n - k$, are k-variant; moreover, each of them can be represented by a common matrix, with at most $2k$ groups, and at most k other row-vectors.

The importance of k-variant of $M[D_G^i]$ can be seen in the next two sections, where we shall show how we can get rid of redundant information in $M[D_G^i]$ and thus create a new matrix that has the same column minima as $M[D_G^i]$, but contains far fewer rows. It is worth pointing out that in both of our algorithms, matrix $M[D_G^i]$ is never physically generated for the obvious reason: It costs too much. Even matrix M, the common matrix of $M[D_G^i]$, is never physically generated, for the same reason. More details can be seen in Sections IV and V.

We now turn to the second property of $M[D_G^i]$. A 2-D matrix Z is *monotone* if the minimum value in its ith column lies below or to the right of the minimum value in its $(i-1)$th column. (If more than one entry has the minimal value in the column, then we take the uppermost one.) In particular, Z is *totally monotone* if every 2×2 submatrix of it is monotone [2]. As an example, matrix $\begin{pmatrix} 4 & 6 & 4 \\ 6 & 4 & 1 \\ 5 & 4 & 3 \end{pmatrix}$ is monotone, but not totally monotone, whereas matrix $\begin{pmatrix} 4 & 6 & 4 \\ 6 & 4 & 1 \\ 6 & 4 & 3 \end{pmatrix}$ is not only monotone but also totally monotone.

Among many nice things about this type of matrix, we mention only two that are important to our algorithms. Submatrices of a totally monotone matrix obtained by deleting rows and columns are totally monotone, and efficient parallel algorithms are available to identify the column minima of a totally monotone matrix [2].

Theorem 4: $M[D_G^i]$ is totally monotone.

Proof: Consider an arbitrary 2×2 submatrix of $M[D_G^i]$:

$$\begin{pmatrix} M[D_G^i](i_1, j_1) & M[D_G^i](i_1, j_2) \\ M[D_G^i](i_2, j_1) & M[D_G^i](i_2, j_2) \end{pmatrix},$$

where $i_1 < i_2$ and $j_1 < j_2$. To prove that $M[D_G^i]$ is totally monotone, it suffices to prove that the above submatrix is monotone, that is, the following:

$$\begin{aligned} M[D_G^i](i_1, j_1) &> M[D_G^i](i_2, j_1) \\ &\Rightarrow M[D_G^i](i_1, j_2) > M[D_G^i](i_2, j_2) \quad (1). \end{aligned}$$

It can be seen that if (1) can be proved when $j_2 = j_1 + 1$, then it can be proved for any j_2, $j_2 > j_1$, inductively. So, we shall show only the following equation:

$$M[D_G^i](i_1, j_1) > M[D_G^i](i_2, j_1) \Rightarrow M[D_G^i](i_1, j_1+1) > M[D_G^i](i_2, j_1+1). \tag{2}$$

We first discuss the case in which four entries in (2) are finite, and leave the other cases to the last. For a contradiction, we assume that the following is true:

$$M[D_G^i](i_1, j_1) > M[D_G^i](i_2, j_1) \text{ and } M[D_G^i](i_1, j_1+1) \le M[D_G^i](i_2, j_1+1). \tag{3}$$

Let $v_1, v_2, \cdots, v_{m/2}$ be the entries of $D_{G_U}^i$ from the first one to the $(m/2)$th one. Then the i_1th row of $M[D_G^i]$ is $CY[i_1 + 1, D_{G_L}^{v_{i_1}}]$ by definition. Keeping this in mind, one can see that (3) is equivalent to the following:

$$D_{G_L}(v_{i_1}, j_1 - i_1) > D_{G_L}(v_{i_2}, j_1 - i_2), \tag{4}$$

and

$$D_{G_L}(v_{i_1}, j_1 - i_1 + 1) \le D_{G_L}(v_{i_2}, j_1 - i_2 + 1). \tag{5}$$

$v_{i_2} < D_{G_L}(v_{i_1}, j_1 - i_1)$ follows immediately from (4), because v_{i_2} is always strictly less than $D_{G_L}(v_{i_2}, j_1 - i_2)$. Thus, there exists j' such that the following is true:

$$D_{G_L}(v_{i_2}, j') = D_{G_L}(v_{i_1}, j_1 - i_1). \tag{6}$$

See Proposition 1(4). On the other hand, applying Proposition 1(1) to (5), we get the following equation:

$$D_{G_L}(v_{i_1}, j_1 - i_1) < D_{G_L}(vi_2, j_1 - i_2 + 1). \tag{7}$$

Combining (4), (6), and (7) we derive the following equation:

$$D_{G_L}(v_{i_2}, j_1 - i_2) < D_{G_L}(v_{i_2}, j') < D_{G_L}(v_{i_2}, j_1 - i_2 + 1),$$

which contradicts the fact that $D_{G_L}(v_{i_2}, j_1 - i_2)$ and $D_{G_L}(v_{i_2}, j_1 - i_2 - 1)$ are two consecutive entries of $D_{G_L}^{v_{i_2}}$.

Now we consider the cases in which some entries in (2) are ∞. In order to be consistent with the definition of monotone matrix, we have to distinguish ∞ entries. For any two ∞ entries in the same column in $M[D_G^i]$, the one having a smaller index is larger than the one having a larger index if both of them are to the right side of some finite entries; otherwise, the one having a smaller index is smaller than the one having a larger index. We shall show only the case where $M[D_G^i](i_1, j_1) = \infty$ and $M[D_G^i](i_2, j_1) \neq \infty$. Other cases are handled similarly. By Proposition 1(1), $M[D_G^i](i_1, j_2)$ should be ∞ because of $j_1 < j_2$. If $M[D_G^i](i_2, j_2)$ is finite, then we have proved the theorem. If $M[D_G^i](i_2, j_2)$ is ∞, the theorem is also true, because $M[D_G^i](i_2, j_1)$ and $M[D_G^i](i_2, j_2)$ are both to the right side of two finite entries, respectively. Thus, we have proved (2). □

IV. A FAST ALGORITHM FOR THE LCS PROBLEM

The purpose of introducing this algorithm in this paper is twofold. First, it solves the LCS problem in $O(\log^2 m + \log n)$ time, which is the fastest algorithm on the CREW-PRAM machine to the best of our knowledge. Second, understanding this algorithm is a warmup for understanding our next algorithm, which is more complicated but optimal. The four main phases of this algorithm have already been described in Section II, and the implementation for Phase 1 and Phase 4 has also been addressed there. Here we focus on the implementations of Phase 2 and Phase 3. Phase 2, which is to generate cost matrix D_G, is the most complicated part of this algorithm. With a divide-and-conquer strategy applied in generating D_G, our focus is on the merge stage, i.e., how to compute D_G based on D_{G_U} and D_{G_L}. Phase 3, which is to identify the maximum-cost path from D_G, is relatively easy, and only a brief discussion on it is given. At the end of this section, we provide an analysis of the complexity of this algorithm. A 2-D array is employed to represent a cost matrix in this algorithm—a big difference from our next algorithm.

A. How to Compute D_G from D_{G_U} and D_{G_L}

Recall the basic formula for computing D_G from D_{G_U} and D_{G_L} which is described in Corollary 1. This computation benefits greatly from the totally monotone property of $M[D_G^i]$ as well as the k-variant property of both D_G^i and $M[D_G^i]$.

Observation 2: From [2], the column minima of an $m \times m$ totally nonotone array Z can be computed in $O(\log m)$ time with $m \log m$ processors on CREW-PRAM.

Applying this observation to the basic formula for computing D_G immediately results in a trivial algorithm for generating D_G, given D_{G_U} and D_{G_L}, which takes $O(\log m)$ time with $mn \log m$ processors. Our first algorithm, however, does even better than this in the sense that the number of processors is reduced by a factor of $\log^2 m$. Let us first state the result.

Theorem 5: $O(\log m)$ time and $mn/\log m$ processors suffice to compute cost matrix D_G with size $n \times m$ from D_{G_U} and D_{G_L}.

The scheme for computing D_G is this: We partition D_G into $n/\log^2 m$ submatrices, each with $\log^2 m$ consecutive rows of D_G, and compute these submatrices independently. $m \log m$ processors are assigned to each submatrix, and they are required to generate the submatrix in $O(\log m)$ time. Without loss of generality, we concentrate on the computation of the submatrix, which consists of the first $\log^2 m$ rows of D_G. Notice that entry $D_G(i,j)$ is the minimum among $D_{G_U}(i,j)$, $D_{G_L}(i,j)$ and jth column minimum of $M[D_G^i]$. To show Theorem 5, we need to show only the following.

Lemma 1: Given cost matrix D_{G_U} and D_{G_L}, the column minima of $M[D_G^1], M[D_G^2], \cdots, M[D_G^{\log^2 m}]$ can be computed in $O(\log m)$ time with $m \log m$ processors.

The column minima of $M[D_G^i]$ are computed by procedure ColMin, which takes as input D_{G_U} and D_{G_L}. Procedure ColMin takes advantage of the information redundancy in $M[D_G^i]$. According to Corollary 1, matrices $M[D_G^1], M[D_G^2], \cdots, M[D_G^{\log^2 m}]$ are $(\log^2 m - 1)$-variant, and each of them can be represented by their common matrix, together with at most $(\log^2 m - 1)$ other row-vectors. The key idea used in procedure ColMin is to compute the column minima of every group of the common matrix, and thus to reduce the size of $M[D_G^i]$ by replacing the submatrices that correspond to groups of the common matrix with the column minima of the submatrices.

For a formal description, we introduce some notations. Let $M = (M_1, \cdots, M_r)^T$ be the common matrix of $M[D_G^i]$ for $1 \leq i \leq \log^2 m$, in which groups are constructed following the rule given in Theorem 3. Let $X[i,j]$ be the submatrices of $M[D_G^i]$, corresponding to M_j, i.e., $X[i,j] = S[K(i,j), M_j]$. By vector Cmin$[Z]$, we denote the column minima of matrix Z, i.e., Cmin$[Z](i)$ is referred to as the minimum of the ith column of matrix Z.

In procedure ColMin, matrix $M[D_G^i]$ is reduced to a new matrix, denoted by $M'[D_G^i]$, by replacing its submatrix $X[i,j]$ with Cmin$[X[i,j]]$. It should be clear that the new matrix shares the same column minima with the original one, i.e., Cmin$[M[D_G^i]]$ = Cmin$[M'[D_G^i]]$. Listed below are three nice things about the new matrices. First, they are small in size. Indeed, applying Corollary 2, readers can easily see that the number of rows in each of $M'[D_G^i]$'s are bounded by $3\log^2 m$, instead of $m/2$ for the original ones. Second, to generate $M'[D_G^i]$'s for $1 \leq i \leq \log^2 m$, we need to compute Cmin$[M_j]$'s only for $1 \leq j \leq r$, instead of computing Cmin$[X[i,j]]$'s for $1 \leq i \leq \log^2 m$ and $1 \leq j \leq r$, independently. This is because Cmin$[X[i,j]]$ can be obtained from Cmin$[M_j]$ by $K(i,j)$-shift, because $X[i,j] = S[K(i,j), M_j]$. Finally, it can be seen that $M'[D_G^i]$'s are totally monotone. Next additional notations are given before procedure ColMin is presented.

Let $L = (L_1, L_2, \cdots, L_r)$ be a common row-vector of row-vectors W_i, for $1 \leq i \leq l$, where L_j is the jth group of L. The *position* of L_j in the row-vector W_i, denoted by Pos$[W_i, L_j]$, is defined as the index of the entry in W_i, which is identical to $L_j(1)$. $P[W_i, L]$ is defined as a function such that $P[W_i, L](j) = \text{Pos}[W_i, L_j]$. Similarly, let $Y = (Y_1, Y_2, \cdots, Y_l)^T$ be a common matrix of Z_i's for $1 \leq i \leq l$, where Y_j is the jth group of Y. The *position* of Y_j in matrix Z_i, denoted by Pos$[Z_i, Y_j]$, is defined as the index of the row-vector in Z_i which is obtained from the first row in Y_j by certain shifting. $P[Z_i, Y]$ is defined as a function such that $P[Z_i, Y](j) = \text{Pos}[Z_i, Y_j]$.

```
Procedure ColMin {Find column minima of $M[D_G^i]$'s, for $1 \leq i \leq \log^2 m$}
Input: cost matrix $D_{G_U}$ and $D_{G_L}$.
Output: Cmin$[M[D_G^i]]$'s for $1 \leq i \leq \log^2 m$.
1) Identify common row-vector $L$ of $D_{G_U}^i$ for $1 \leq i \leq \log^2 m$:
    a) Compute $L = (L_1, \cdots, L_r)$; and
    b) Compute $P[D_{G_U}^i, L]$, for $1 \leq i \leq \log^2 m$.
2) Identify common matrix $M$ of $M[D_G^i]$ for $1 \leq i \leq \log^2 m$:
    a) Compute $P[M[D_G^i], M]$, for $1 \leq i \leq \log^2 m$; and
    b) Compute $K(i,j)$ such that $X[i,j] = S[K(i,j), M_j]$, for $1 \leq i \leq \log^2 m$ and $1 \leq j \leq r$.
3) Compute Cmin$[M_j]$, for $1 \leq j \leq r$;
4) Compute Cmin$[M[D_G^i]]$ for $1 \leq i \leq \log^2 m$.
```

B. The Analysis of Procedure ColMin

We prove Lemma 1 by claiming that $O(\log m)$ time suffices for completing each step in ColMin with $m \log m$ processors.

Lemma 2: Common row-vector L of $D_{G_U}^i s$, for $1 \leq i \leq \log^2 m$, can be identified in $O(\log m)$ time using $m/2$ processors.

Common row-vector L of $D_{G_U}^i$s can be computed from $D_{G_U}^1$ and $D_{G_U}^{\log^2 m}$, according to Proposition 3(1). This computation takes advantage of the fact that entries in $D_{G_U}^1$ and $D_{G_U}^{\log^2 m}$ are monotonely increasing in value (by Proposition 1(1)). One processor is assigned to each entry in $D_{G_U}^1$. Typically, processor P_v assigned to entry v of $D_{G_U}^1$ independently executes a binary search on $D_{G_U}^{\log^2 m}$, and marks v if it finds an entry w that is identical to v. Clearly, those marked entries are entries shared by both $D_{G_U}^1$ and $D_{G_U}^{\log^2 m}$, and therefore L is obtained simply by ranking them. In order to further partition L into groups, $L_1, L_2, \cdots, L_r$, we search and mark every pair of consecutive entries of L, say, l_1 and l_2, such that they are not consecutive in $D_{G_U}^{\log^2 m}$. Since the entries in the same group should be consecutive in $D_{G_U}^{\log^2 m}$, l_1 and l_2 must belong to two groups, and, moreover, l_1 must be the last entry in some group L_i, and l_2 be the first entry in group L_{i+1}. Thus, we partitioned L into groups. It is easily seen that the above tasks can be done in $O(\log m)$ time with $m/2$ processors.

$P[D_{G_U}^i, L]$ is to be computed by identifying the position of $L_j(1)s$ in $D_{G_U}^i$ for $1 \leq j \leq r$. Since the number of groups r is bonded by $\log^2 m$ (see Corollary 1), and since there are $\log^2 m$ such functions to be computed in total, Step 1(b) can be done in $O(\log m)$ time with $\log^4 m$ processors.

Lemma 3: Given common row-vector $L = (L_1, \cdots, L_r)$ of $D_{G_U}^i$'s for $1 \leq i \leq \log^2 m$, the corresponding position functions $P[M[D_G^i], M]$'s and $K(i,j)$'s for $1 \leq i \leq \log^2 m$ and $1 \leq j \leq r$ can be identified in $O(1)$ time with $\log^4 m$ number of processors.

As a matter of fact, once $L = (L_1, L_2, \cdots, L_r)$ is obtained, $M = (M_1, \cdots, M_r)^T$ can be constructed following the rule in Theorem 3. As for the computation of $P[M[D_G^i], M]$, recall that the (Pos$[D_{G_U}^i, L_j]$)th row of $M[D_G^i]$ is $CY[\text{Pos}[D_{G_U}^i, L_j] + 1, D_{G_L}^{L_j(1)}]$ (from the definition of $M[D_G^i]$), and also recall that the first row of M_j is $CY[1, D_{G_L}^{L_j(1)}]$ (from Theorem 2). So, the former can be obtained from the latter by (Pos$[D_{G_U}^i, L_j]$)-shift, implying Pos$[M[D_G^i], M_j]$ = Pos $[D_{G_U}^i, L_j]$. Thus, in general, we have $P[M[D_G^i], M] = P[D_{G_U}^i, L]$. Since $P[D_{G_U}^i, L]$ has already been computed in Step 1(b), Step 2(b) can be obtained in constant time. This analysis also implies that $K(i,j)$ = Pos$[D_{G_U}^i, L_j]$. Constant time with $\log^4 m$ processors should suffice to identify the common matrix of $M[D_G^i]$'s.

Lemma 4: Given D_{G_U}, D_{G_L}, $P[M[D_G^i], M]$s and $K(i,j)$'s for $1 \leq i \leq \log^2 m$ and $1 \leq j \leq r$, Cmin$[M_j]$'s can be computed in $O(\log m)$ time with $m \log m$ processors.

For the sake of simplicity, without loss of generality, we assume that any group M_j has no more than $m/\log^2 m$ rows; otherwise, we partition the larger groups into several smaller groups, and the total number of groups should still be bounded by $O(m/\log^2 m)$. Consider the computation of Cmin$[M_j]$. For the worst case, we assume that it is of size $(m/\log^2 m) \times m$. We first compute Cmin$[Z]$ where Z is an $(m/\log^2 m) \times (m/\log^2 m)$ matrix consisting of every $(\log^2 m)$ column of M_j. Clearly, Z is totally monotone, because M_j is totally monotone, and thus Aggarwal and Park's algorithm [2] is applicable to compute Cmin$[Z]$. By Observation 2, this computation can be done in $O(\log m)$ time by using $m/\log m$ processors. Then applying Aggarwal and Park's algorithm, the remaining minima of M_j can be computed in $O(\log m)$ time, also using $m/\log m$ processors. In total, all Cmin$[M_j]$'s for $1 \leq j \leq r$ can be computed in $O(\log m)$ time with $m \log m$ processors.

Lemma 5: Given D_{G_U}, D_{G_L}, Cmin$[M_j]$'s, $P[M[D_G^i], M]$'s, and $K(i,j)$ for $1 \leq i \leq \log^2 m$ and $1 \leq j \leq r$, Cmin$[M[D_G^i]$'s for $1 \leq i \leq \log^2 m$, can be computed in $O(\log m)$ time with $m \log m$ processors.

Instead of computing Cmin$[M[D_G^i]]$ directly, we compute Cmin$[M'[D_G^i]]$, a much smaller matrix. Recall that $M'[D_G^i]$ consists of row-vectors Cmin$[X[i,j]$'s for $1 \leq j \leq \log^2 m$ and at most $\log^2 m$ other row-vectors from $M[D_G^i]$. Also recall that Cmin$[X[i,j]]$ is obtained from Cmin$[M_j]$ by $K(i,j)$-shift. The construction of $M'[D_G^i]$'s should be easy under the assistance of $P[M[D_G^i], M]$. Since $M'[D_G^i]$'s are totally monotone, the similar approach we just used for computing Cmin$[M_j]$'s can be used to compute Cmin$[M'[D_G^i]]$'s using the required time and number of processors. Thus, we have completed the proof of Lemma 1, and have proved Theorem 5.

C. The Implementation of Phase 3

The objective of this section is to describe a scheme to identify vertices on the maximum-cost path between the source and the sink of the $(m+1) \times (n+1)$ grid DAG G, given D_G. For a maximum-cost path from the source to the sink on G, say, $p = \langle v_1, v_2, \cdots, v_l \rangle$, there could be more than one vertex of p that are on the same row of G, because $m \leq n$. A vertex, say, v_i, on p is a *cross-vertex* if v_i is the leftmost one to the left of vertices on p. We denote the cross-vertex on the jth row of G as $v[j]$ to distinguish cross-vertices from other vertices on p. Clearly, $v_1 = v[1]$. Phase 3 in the main structure described in Section II-B is implemented by two stages.

1) Identify cross vertices $v[i]$ on p, for $1 \leq i \leq m+1$.
2) Identify the other vertices on p.

We start with the first stage. All cross-vertices on a maximum-cost path can be obtained as the side effect of computing cost matrix D_G. Suppose we are computing $D_G(i,j)$, that is, identifying in G the j-breakout vertex, denoted as y, of vertex $(1,i)$, denoted as x. Let p be the maximum-cost path from x to y, and let vertex q be the cross-vertex of p on the boundary betwen G_U and G_L, meaning $q = v[m/2+1]$. By side effect, we mean that q can be identified in Phase 2 without spending extra time. In fact, according to Theorem 1, $D_G(i,j)$ is the minimum among $D_{G_U}(i,j), D_{G_L}(i,j)$, and $D_{G_L}(D_{G_U}(i,k), j-k)$'s, for $1 \leq k \leq j$. If $D_G(i,j) = D_{G_U}(i,j)$, then obviously q is vertex $(m/2+1, D_{G_U}(i,j))$ on G. If $D_G(i,j) = D_{G_L}(i,j)$, then q is vertex $(m/2+1, i)$ on G. Otherwise, suppose that $D_G(i,j) = D_{G_L}(D_{G_U}(i,k), j-k)$; then q is the $(m/2+1, D_{G_U}(i,k))$th vertex on G. Those cross-vertices such as q are stored in a global matrix. Thus, given two vertices x and y, we can print out the corresponding cross-vertex q in constant time. Recursively, the cross-vertices on the maximum-cost path between source and sink can be printed out in $O(\log m)$ time using $m/\log m$ processors.

The second stage is simple. Suppose that $v[i]$ and $v[i+1]$ are vertex (i, j_1) and vertex $(i+1, j_2)$ on G, respectively. Then vertices on the ith row of G from (i, j_1+1) to (i, j_2-1) must be all the vertices between $v[i]$ and $v[i+1]$ on p. So, once all cross-vertices have been identified through the first stage, there should be no difficulty in printing out all vertices on p in $O(\log n)$ time with n processors.

D. The Complexity of the Algorithm

We first state the result.

Theorem 6: $O(\log^2 m + \log n)$ time with $mn/\log m$ processors suffices to identify the LCS of A and B with lengths m and n, respectively, where $m \leq n$.

Recall the four phases described in Section II-B. We have shown in Section II-C that Phase 1 can be done in $O(\log n)$ time with $mn/\log n$ processors, and that Phase 4 can be done in $O(\log n)$ time with $n/\log n$ processors. We have just shown in Section IV-C that $O(\log n)$ time and n processors suffice for Phase 3. So, to show Theorem 6, we shall show that $O(\log^2 m)$ time and $mn/\log m$ processors suffice for Phase 2. Let $T(k)$ be the time taken to generate a cost matrix of size $(k+1) \times (n+1)$. The execution time of Phase 2, which is $O(\log^2 m)$, can be obtained from the following recurrence suggested by Theorem 5:

$$T(k) \leq T(k/2) + c_1 \log k,$$

where $T(k/2)$ is the time taken to generate the two smaller cost matrices, and $c_1 \log k$ is the time taken for merge. $T(2)$ is defined as 0, because time $T(2)$ is charged on Phase 1. In order to use no more than $mn/\log m$ processors throughout the algorithm, Brent's principle [4] must be applied in Phase 2. Notice that Phase 2 is a recursive process, and has $\log m$ merge stages, each costing $O(mn)$ operations. Therefore, these operations in each stage can be performed in $O(\log m)$ time using $mn/\log m$ processors, according to Brent's principle.

V. AN OPTIMAL ALGORITHM FOR THE LCS PROBLEM

The four phases of this algorithm are described in Section II-B, in which Phase 1 and Phase 4 are shared with the previous algorithm. In this section, we focus on the implementations of Phase 2 and Phase 3. The fundamental difference between this algorithm and the previous one lies in the data structure used for representing cost matrices. In the previous algorithm 2-D arrays are used to implement cost matrices; unfortunately, this approach would destroy any hope of achieving optimality. To see this, just think about computing and storing $O(m/2^i)$ cost matrices, each of size $n \times O(2^i)$ at $(i+1)$th merge stage of Phase 2, where each matrix corresponds to an $O(2^i) \times (n+1)$-grid DAG. Obviously, this stage alone costs at least $O(mn)$ operations. The fact that there are $O(\log m)$ merge stages in Phase 2 implies that at least $O(mn \log m)$ operations are needed, which is larger than our desired bound, mn. For this reason, a very efficient data structure is adopted for D_G in our optimal algorithm.

A. The Data Structure for D_G

We use common vectors and remnants to represent cost matrix D_G. Specifically, for every consecutive $k+1$ rows of $m \times n$ cost matrix D_G, say, D_G^i for $1 \leq i \leq k+1$, we use an 1-D array to represent their common row-vector L, and use $k+1$ 1-D arrays to represent $k+1$ remnants $R[D_G^i]$'s. Besides, position functions $P[D_G^i, L]$'s and $P[D_G^i, R[D_G^i]]$'s defined in the previous section are also used in order to access any entry of D_G on this data structure very fast. It is not hard to see that with the help of position functions and binary search, any read or write operation on D_G can be simulated in $O(\log k)$ time sequentially on this data structure. As an example, when $k = 2$, the first three row-vectors of D_G, which are $D_G^1 = (2,3,4,5,13,\infty,\infty,\infty)$, $D_G^2 = (3,4,5,11,13,\infty,\infty,\infty)$, and $D_G^3 = (4,5,6,11,13,\infty,\infty,\infty)$, will be represented by the following data structure:

$$L = (4,5,13,\infty,\infty,\infty),$$
$$\text{in which } L_1 = (4,5), L_2 = (13,\infty,\infty,\infty);$$
$$R[D_G^1] = (2,3), \quad R[D_G^2] = (3,11), \quad R[D_G^3] = (6,11);$$
$$P[D_G^1, L] = (3,5), \quad P[D_G^1, R[D_G^1]] = (1);$$
$$P[D_G^2, L] = (2,5), \quad P[D_G^2, R[D_G^2]] = (1,4);$$
$$P[D_G^3, L] = (1,5), \quad P[D_G^3, R[D_G^3]] = (3).$$

Throughout this section, we presume that any cost matrix is represented by the above data structure. So, by computing D_G, we mean computing the common row-vectors, corresponding remnants, and position functions.

Now we would like to make a short comment on why the above representation of D_G can help us to achieve our objective. Note that by Theorem 2, $k+1$ consecutive rows of D_G with size $n \times m$ can be represented by their common vector L with a size of at most m, and $k+1$ remnants, each with size of at most k; hence, $O(mn/k + nk)$ space suffices to store all information in D_G. By this representation, not only is the redundant information in D_G removed, and thus storage space is cut a great deal, but also the number of entries to be computed is greatly reduced. Two simple facts about the position function are stated as follows.

Proposition 4: Let L_h be the hth group of L, the common row-vector of D_G^1's for $i_1 \leq i \leq i_1 + k$.

1) $Pos[D_G^{i_1+k}, L_h] \leq Pos[D_G^i, L_h] \leq Pos[D_G^{i_1}, L_h]$.
2) $Pos[D_G^{i_1+k}, L_h] \geq Pos[D_G^i, L_h] - k$.

Proof: Let $\text{Pos}[D_G^{i_1}, L_h] = j_1$, $\text{Pos}[D_G^i, L_h] = j_2$, and $\text{Pos}[D_G^{i_1+k}, L_h] = j_3$. In other words, $D_G(i_1, j_1) = D_G(i, j_2) = D_G(i_1 + k, j_3) = L_k(1)$. According to Proposition 1(2), we have $j_1 \leq j_2 \leq j_3$, because $i_1 \leq i \leq i_1 + k$. As for Proposition 4(2), noticing $D_G(i_1 + 1, j_1 - 1) \leq D_G(i_1, j_1 - 1) = L_h(1)$ (see Proposition 1(3); hence, we have $\text{Pos}[D_G^{i_1+1}, L_h] \geq j_1 - 1$. Applying Proposition 1(3) repeatedly, we get $\text{Pos}[D_G^{i_1+k}, L_h] \geq j_1 - k$, which is equivalent to $\text{Pos}[D_G^{i_1+k}, L_h] \geq \text{Pos}[D_G^{i_1}, L_h] - k$. □

B. The Algorithm for Computing D_G

We first establish the time bound for computing D_G from D_{G_U} and D_{G_L}, and then provide the proof in the next subsection.

Theorem 7: Given D_{G_U} and $D_{G_L}, O(\log m \log\log m)$ time with $m\log m$ processors suffices for computing $\log^4 m$ consecutive row-vectors in cost matrix D_G of size $n \times m$.

Corollary 3: Given D_{G_U} and $D_{G_L}, O(\log m \log \log m)$ time with $mn/\log^3 m$ processors suffices to generate cost matrix D_G of size $n \times m$.

Similarly to our first algorithm, D_G is partitioned into $n/\log^4 m$ submatrices, each with $\log^4 m$ consecutive rows of D_G, and $m \log m$ processors are assigned to each submatrix. Those submatrices are required to be generated in $O(\log m \log\log m)$ time. Differently from our first algorithm, this time we take advantage of not only the k-variant property of $M[D_G^i]$'s but also the k-variant property of D_G^i. For the sake of simplicity, we explain only how the submatrix, which consists of the first $\log^4 m$ rows of D_G, is computed; other submatrices are handled in exactly the same way. The computation of the submatrix contains three steps.

1) Compute the common row-vector L of D_G^is for $1 \leq i \leq \log^4 m$ as follows:
    a) Compute D_G^1 and $D_G^{\log^4 m}$;
    b) Compute common row-vector L of $D_G^1, D_G^{\log^4 m}$;
    c) Compute corresponding position functions $P[D_G^1, L]$ and $P[D_G^{\log^4 m}, L]$;
2) Compute remnants $R[D_G^i]$, for $1 \leq i \leq \log^4 m$;
3) Compute position functions $P[D_G^i, L]$ and $P[D_G^i, R[D_G^i]]$, for $1 \leq i \leq \log^4 m$.

To avoid possible confusion on Step 1, we would like to remind readers the fact that common row-vector L of $D_G^1, D_G^{\log^4 m}$ is the common row-vector of the $\log^4 m$ rows (see Propostion 3(1)).

C. The Proof of Theorem 7

We prove Theorem 7 by providing implementation and corresponding analysis for each of these three steps. Step 1 can be implemented similarly to Step 1 of ColMin. The only difference is the data structure of D_G. Keeping this in mind, one should have no difficulty in seeing that Step 1 can be done in $O(\log m \log\log m)$ time with $m \log m$ processors. We leave this for readers to calculate as an exercise.

Step 2, which generates the remnants, is the most complicated part of the whole algorithm. Let $R[D_G^i] =$

$(R_1^i, \cdots, R_{r+1}^i)$ be the remnant of D_G^i. We shall show only how to compute remnant group R_b^a, the bth group of $R[D_G^a]$, where $1 \leq a \leq \log^4 m$ and $1 \leq b \leq r+1$; other remnant groups can be handled in exactly the same way. We assume that L contains at least one entry that is not ∞; the case in which L contains only ∞'s should be easy to deal with. Two facts should be noted: Any finite entry in R_b^a, $b \geq 2$, exists also in $R_b^{\log^4 m}$ (see Proposition 3(3)), and ∞ entries are always on the right side of finite entries in D_G^a. Thus, removing ∞ entries from $R_b^{\log^4 m}$ will not affect generating finite entries in R_b^a. Without loss of generality, we assume that there is no ∞ entry in $R_b^{\log^4 m}$.

Note that since D_G^1, $D_G^{\log^4 m}$ and L have already been computed, it should be easy to compute $R[D_G^1]$ and $R[D_G^{\log^4 m}]$. Other remnants are computed from them. There are two cases to be considered: $b = 1$ and $b \geq 2$. By Proposition 3(3), when $b \geq 2 R_b^a$ is a subvector of $R_b^{\log^4 m}$, which, together with the monotonality of D_G^a, implies that R_b^a must be the largest common subvector of D_G^a and $R_b^{\log^4 m}$. Therefore, if D_G^a is given, then R_b^a can be obtained by identifying the largest common subvector of D_G^a and $R_b^{\log^4 m}$. Unfortunately, D_G^a is unknown. Instead of D_G^a, we are going to compute a subrow of D_G^a, denoted as $SD[a, b]$, such that 1) R_b^a is a subvector of $SD[a, b]$, and 2) the size of it is bounded by $O(\log^4 m)$. Leaving the question of how to get $SD[a, b]$ for now, we first examine the computation time for R_b^a, supposing that $SD[a, b]$ is available. Notice that the size of $R[D_G^{\log^4 m}]$ is bounded by $\log^4 m$, and so is the size of $R_b^{\log^4 m}$. Hence, the method we used in Step 1(a) of procedure ColMin can be used to compute R_b^a, and the following time bound should be quite clear.

Lemma 6: Suppose that $\mathrm{SD}[a, b]$ is given, where $b \geq 2$, and that the size of $\mathrm{SD}[a, b]$ is bounded by $O(\log^4 m)$. R_b^a can be computed in $O(\mathrm{loglog} m)$ time with $\log^4 m$ processors.

When $b = 1$, it can be shown by applying Proposition 1(3) that the size of R_1^a is bounded by $\log^4 m$. So, we define $SD[a, 1]$ as the first $\log^4 m$ entries of D_G^a. Let k be the size of R_1^a; then R_1^a consists of the first k entries of $SD[a, 1]$, and the $(k+1)$th entry of $SD[a, 1]$ is identical to $L_1(1)$. From this, we draw the following conclusions.

Lemma 7: Suppose that $\mathrm{SD}[a, 1]$ and $L(1)$ are given, then $\log^4 m$ processors suffice to identify R_1^a in constant time.

From Lemmas 6 and 7, together with the fact that r, the number of groups of L, is bounded by $\log^4 m$, we further draw the following conclusions.

Corollary 4: Suppose that $\mathrm{SD}[i, j]$ and $L(1)$ are given, and that the size of $\mathrm{SD}[i, j]$ is bounded by $\log^4 m$. Then R_j^i's, for $1 \leq i \leq \log^4 m$ and $1 \leq j \leq r$, can be obtained in $O(\mathrm{loglog} m)$ time by using $\log^{12} m$ processors.

Since $L_1(1)$ has already been computed in Step 1(b), in the remainder of this section, we concentrate on the critical problem of how to find $SD[a, b]$ with $b \geq 2$ such that R_b^a is a subvector of it and such that its size is bounded by $O(\log^4 m)$.

The basic formula described in Corollary 1 is applied to generate $SD[a, b]$. (Remember that $SD[a, b]$ is nothing but a subrow of D_G^a.) We first discuss the issue of which entries of D_G^a should be included in $SD[a, b]$. A vector, $\mathrm{Ind}[SD[a, b]]$, is used to record their original positions in D_G^a for entries in $SD[a, b]$, as follows:

$$\mathrm{Ind}[SD[a, b]](i) = l \text{ if } SD[a, b](i) = D_G^a(l).$$

Understanding the following relation between $\mathrm{Pos}[D_G^a, R_b^a]$ and $\mathrm{Pos}[D_G^{\log^4 m}, R_b^{\log^4 m}]$, i.e., the relation between the position of the first entry of R_b^a in D_G^a and the position of the first entry of $R_b^{\log^4 m}$ in $D_G^{\log^4 m}$, is the key to figure out how to find $\mathrm{Ind}[SD[a, b]]$,

$$l \leq \mathrm{Pos}[D_G^a, R_b^a] \leq l + \log^4 m,$$

where $l = \mathrm{Pos}[D_G^{\log^4 m}, R_b^{\log^4 m}]$. This inequality is derived from Proposition 4. Remember that there are no more than $\log^4 m$ entries in R_b^a, so we can choose the subrow of D_G^a, from entry $D_G(a, l)$ to entry $D_G(a, l + 2\log^4 m - 1)$, as $SD[a, b]$. Surely, R_b^a must be contained in $SD[a, b]$ according to the above inequality. $\mathrm{Ind}[SD[a, b]]$ therefore can be found as follows. For $1 \leq i \leq 2\log^4 m$,

$$\mathrm{Ind}[SD[a, b]](i) = \mathrm{Pos}[D_G^{\log^4 m}, R_b^{\log^4 m}] + i - 1. \qquad (8)$$

The next issue is how to compute $SD[a, b]$ from D_{G_U} and D_{G_U}. Clearly, we need to identify only the $(\mathrm{Ind}[SD[a, b]](i))$th column minima in $M[D_G^a]$ for $1 \leq i \leq 2\log^4 m$. Let $M[SD[a, b]]$ be a submatrix of $M[D_G^a]$ obtained by removing those columns that are not in $SD[a, b]$. Readers may have already noticed the following important fact about $M[SD[a, b]]$: First, $M[SD[a, b]]$ is totally monotone. Second, the size of $M[SD[a, b]]$ is very small, which in fact is bounded by $(m/2) \times (2\log^4 m)$. Unfortunately, the size of $M[SD[a, b]]$ is still not small enough for us to identify the column minima of $M[D_G^a]$ by directly applying Corollary 1 on it. For further reducing the computation, we introduce another matrix $M'[SD[a, b]]$ that has the same column minima as $M[SD[a, b]]$, but with far fewer rows. (Remember that we play the same game in our first algorithm.) For a formalized description of $M'[SD[a, b]]$, a few more notations are needed.

Let $L' = (L'_1, \cdots, L'_{r_1})$ be the common row-vector of $D_{G_U}^i$'s for $1 \leq i \leq \log^4 m$. Let $M = (M_1, \cdots, M_{r_1})^T$ be the common matrix of $M[D_G^i]$'s, for $1 \leq i \leq \log^4 m$, constructed following the rule in Theorem 3 (that is, $M_j = CY[i, D_{G_L}^{L'_j(i)}]$). Again, let $X[i, k]$ be the submatrix of $M[D_G^i]$, corresponding to M_k (i.e., $X[i, k] = S[K(i, k), M_k]$ for some $K(i, k)$). By $SM[SD[i, j], k]$, we refer to the submatrix of both $M[SD[i, j]]$ and $X[i, k]$ such that $SM[SD[i, j], k]$ has the maximum number of columns and rows. See Fig. 6. Matrix $M'[SD[a, b]]$ is defined as the matrix obtained from $M[SD[a, b]]$ by replacing matrices $SM[SD[a, b], k]$ with $\mathrm{Cmin}[SM[SD[a, b], k]]$'s for $1 \leq k \leq r_1$. The following facts about $M'[SD[a, b]]$ are obvious.

Proposition 5:

1) $\mathrm{Cmin}[M'[\mathrm{SD}[a, b]]] = \mathrm{Cmin}[M[\mathrm{SD}[a, b]]]$.
2) The size of $M'[\mathrm{SD}[a, b]]$ is bounded by $(2\log^4 m) \times (2\log^4 m)$.

Corollary 5: Suppose that we have already obtained $M'[\mathrm{SD}[i, j]]$'s, for $1 \leq i \leq \log^4 m$ and $1 \leq j \leq r$, which are represented by arrays, then $\mathrm{Cmin}[M[\mathrm{SD}[i, j]]]$'s, for $1 \leq i \leq \log^4 m$ and $1 \leq j \leq r$, can be computed

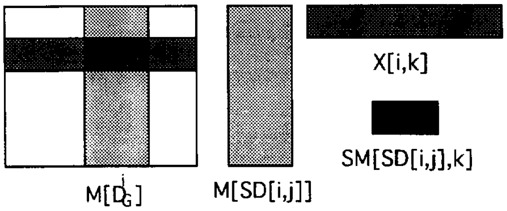


Fig. 6. Illustrating $sm[sd[i,j],k]$.

in $O(\text{loglog}\, m)$ time using polylogarithmic number of processors.

Now we state the objective for the rest of the section.

Lemma 8: SD$[a,b]$ can be generated in $O(\log m \text{loglog}\, m)$ time using polylogarithmic number of processors.

Noticing that once Cmin$[M[SD[a,b]]]$ is obtained, $SD[a,b]$ can then be easily generated by performing just a few more comparisons (see Corollary 1), to show Lemma 8, we shall show that $M'[SD[a,b]]$ can be generated in $O(\log\log m)$ time using polylogarithmic number of processors. Recall that $M'[SD[a,b]]$ is defined as the matrix obtained from $M[SD[a,b]]$ by replacing its submatrices $SM[SD[a,b],k]$'s with Cmin$[SM[SD[a,b],k]]$'s for $1 \leq k \leq r$. The numbers of rows and columns of $M'[SD[a,b]]$ are bounded by $(2\log^4 m) \times (2\log^4 m)$ (see Proposition 5(2)). Each entry of $M'[SD[a,b]]$, except those entries belonging to Cmin$[SM[SD[a,b],k]]$, can be obtained from D_{G_U} and D_{G_L} in $O(\log\log m)$ time using polylogarithmic number of processors. Therefore, we can make the following observation.

Observation 3: Suppose Cmin[SM[SD$[a,b],k]]$'s, for $1 \leq k \leq r_1$, are given; then Cmin$[M[\text{SD}[a,b]]]$ can be obtained in the required time with polylogarithmic number of processors.

Now we narrow our discussion further down to how to generate Cmin$[SM[SD[a,b],k]]$'s in $O(\log\log m)$ time using polylogarithmic number of processors. A little calculation will show that it is unaffordable to compute Cmin$[SM[SD[a,b],k]]$'s by identifying the column minima of $SM[SD[a,b],k]$s individually. The following two facts help us to further reduce the computational time: Cmin$[SM[SD[a,b],k]]$ is a subvector of Cmin$[X[a,k]]$; Cmin$[X[a,k]]$ can be obtained from Cmin$[M_k]$ by $K(a,k)$-shift for $1 \leq k \leq r_1$, where M_k is the kth group of common matrix M (since $X[a,k]$ is obtained from M_k by $K(a,k)$-shift).

Because of these two facts, instead of generating all Cmin$[SM[SD[a,b],k]]$'s individually, we generate all Cmin$[X[a,k]]$'s. Moreover, instead of generating all Cmin$[X[a,k]]$'s individually, we generate all Cmin$[M_k]$'s for $1 \leq k \leq r_1$. Although the computation has been generally reduced so far, it is still too expensive. Indeed, there could be $O(\log^4 m)$ M_k's each with size $(m/\log^4 m) \times m$. It is impossible to compute all column minima of them in $O(\log m \log\log m)$ time just by using $m \log m$ processors. Fortunately, not all entries of Cmin$[M_k]$'s are useful for generating $M'[SD[a,b]]$. We define a submatrix SM_k of M_k by removing those columns of M_k that are irrelevant to generate Cmin$[SM[SD[a,b],k]]$'s. Let Ind$[SM_k]$ be the vector that records their original positions in M_k for each column in SM_k; that is, Ind$[SM_k](h) = l$ if $SM_k(h) = M_k(l)$.

Ind$[SM_k]$ is dependent on Ind$[SD[a,b]]$ and $K(a,k)$. The procedure that generates Ind$[SM_k]$ is described as follows.

Procedure Ind$[SM_k]$

Suppose that $K(a,k)$ is the coefficient such that $X[a,k] = S[K(a,k), M_k]$.

1) For $1 \leq h \leq 2\log^4 m$ processor P_h is assigned to compute Ind$[SM_k](h)$:

$$\text{Ind}[SM_k](h) = \begin{cases} \text{Ind}[SD[a,b]](h) - K(a,k), \\ \qquad\qquad \text{if } \text{Ind}[SD[a,b]](h) - K(a,k) \geq 1, \\ m - K(a,k) + \text{Ind}[SD[a,b]](h), \qquad \text{otherwise.} \end{cases}$$

2) Sort all entries in Ind$[SM_k]$ for $1 \leq k \leq r_1$, and mark an entry if it is the leftmost one among those that have the same value.

3) Ind$[SM_k]$ is obtained by ranking those marked entries.

Step 1 in this procedure guarantees that all those columns in M_k that may be used in generating $SD[a,b]$ are identified. Step 2 and Step 3 are used to get rid of those entries that should not belong to Ind$[SM_k]$. Step 1 takes constant time using $O(\log^4 m)$ processors, whereas Step 2 and Step 3 take $O(\log\log m)$ time using $O(\log^4 m)$ processors, because the size of Ind$[SM_k]$ is bounded by $O(\log^4 m)$. Hence, to compute Ind$[SM_k]$'s, for $1 \leq k \leq r_1, O(\log\log m)$ time suffices with polylogarithmic number of processors.

Since M_k is totally monotone, so is SM_k. In order to calculate the time bound for computing Cmin$[SM_k]$ by applying Aggarwal and Park's algorithm, we bring readers' attention to the size of SM_k's. The number of columns in each SM_k is bounded by $O(\log^4 m)$; the total number of rows in all SM_k's for $1 \leq k \leq r_1$ is bounded by $m/2$. Without loss of generality, we can assume that there are at most $(m/2)/\log^4 m$ rows in SM_k; otherwise, we can divide the larger SM_k into several smaller ones, each consisting of at most $(m/2)/\log^4 m$ rows, and the total number of SM_k's will not be more than $2r_1$. According to Observation 2, $m \log m$ processors suffice to compute Cmin$[SM_k]$'s for $1 \leq k \leq r_1$ in $O(\log m \log\log m)$ time.

Once Cmin$[SM_k]$ is generated, any entry of Cmin$[SM[SD[a,b],k]]$ can then be obtained from it in $O(\log\log m)$ time. Indeed, the hth entry of Cmin$[SM[SD[a,b],k]]$ is the Ind$[SD[a,b]](h)$th entry of Cmin$[X[a,k]]$, i.e., the $(\text{Ind}[SD[a,b]](h) - K(a,k))$th entry of Cmin$[M_k]$ if Ind$[SD[a,b]](h) - K(a,k) \geq 1$, or, otherwise, is the $(m - K(a,k) + Ind[SD[a,b]](h))$th entry of Cmin$[M_k]$. With the help of Ind$[SM_k]$, we can locate the hth entry of Cmin$[SM[SD[a,b],k]]$ in Cmin$[SM_k]$ through a binary search. Since the number of columns in SM_k is bounded by $O(\log^4 m)$, such a search takes $O(\log\log m)$ time.

Once Cmin$[SM[SD[a,b],k]]$'s are computed, Cmin$[M[SD[a,b]]]$ can then be computed in the required time, as mentioned by Observation 3. Thus, we have completed the proof of Lemma 8. Corollary 4, together with Lemma 8, suggests that Step 2(b) can be done in required time using the required number of processors. Finally, once $R[D_G^i]$, for $1 \leq i \leq \log^4 m$, are computed, there will not be any difficulty in computing $P[D_G^i, L]$'s and $P[D_G^i, R[D_G^i]]$'s, for $1 \leq i \leq \log^4 m$, because the number of groups of L and $R[D_G^i]$ are bounded by a polylogarithm. Thus, we have completed the proof of Theorem 7.

D. The Implementation of Phase 3

Like our first algorithm, Phase 3 in our optimal algorithm is implemented by two stages:

1) Identify cross vertex $v[i]$ on p, for $1 \leq i \leq m+1$.
2) Identify other vertices on p.

Obviously, the discussion about the second stage in our first algorithm is still applicable. Unfortunately, the discussion about the first stage in that algorithm is no longer applicable here. Remember that in the first algorithm, all cross-vertices are computed and stored as the side effect of Phase 2. Since the method of computing cost matrices in our optimal algorithm is totally different, the side effect has not been preserved. A new method is needed. In what follows, we first state the result.

Theorem 8: Suppose that all cost matrices are given. Then cross-vertices on p, $v[i]$, for $1 \leq i \leq m+1$, can be identified in $O(\log^2 m)$ time with n processors.

The claim is to be shown by presenting a procedure called *CrossVertex*$(v[i_1], v[i_2])$. Procedure *CrossVertex*$(v[i_1], v[i_2])$ takes cost matrices obtained from Phase 2 as input, and returns cross-vertices on p between $v[i_1]$ and $v[i_2]$.

```
Procedure CrossVertex $(v[i_1], v[i_2])$
1) $v[(i_1+i_2)/2] \leftarrow \theta(v[i_1], v[i_2])$.
2) Call CrossVertex $(v[i_1], v[(i_1+i_2)/2])$ and CrossVertex
   $(v[(i_1+i_2)/2], v[i_2])$ if $i_1 \neq i_2$.
```

Function $\theta(v[i_1], v[i_2])$ calculates the location of the cross-vertex that is in the middle of $v[i_1]$ and $v[i_2]$. Initially, we call CrossVertex $(v[1], v[m+1])$. To explain how $\theta()$ works, we take only $\theta(v[1], v[m+1])$ as an example, without loss of generality.

Lemma 9: $O(\log m)$ time using n processors suffices to compute $\theta(v[1], v[m+1])$ (i.e., to identify cross-vertex $v[m/2+1]$).

$v[1]$ is the source, and $v[m+1]$ is a vertex on the bottom row of G. We assume that l is the cost of the maximum-cost path between them. According to our discussion in Section II-B, cross-vertex $v[m/2+1]$ should be a breakout vertex of $v[1]$ on G_U, and $v[m+1]$ should be the $(l-k)$th breakout vertex of $v[m/2+1]$ on G_L if the maximum cost from $v[1]$ to $v[m/2+1]$ is k. To identify $v[m/2+1]$, we assign one processor to each breakout vertex of $v[1]$ on G_U. Processor P_i, which is assigned to the ith breakout vertex of $v[1]$, does the following. With the help of the position functions, P_i first reads entry $D_{G_U}(1, i)$ in $O(\log\log m)$ time. Then, in another $O(\log\log m)$ time, P_i checks whether vertex $(m+1, D_{G_L}(D_{G_U}(1, i), l-i))$ is identical to cross-vertex $v[m+1]$, and marks vertex $(m/2+1, D_{G_U}(1, i))$ on G if it is. Finally, the leftmost one among those marked vertices is identified in $O(\log m)$ time with m processors. Thus, CrossVertex$(v[1], v[m+1])$ can be computed in $O(\log m)$ time using m processors.

As for the time bound, $T(m)$, of *CrossVertex*$(v[1], v[m+1]), T(m) = O(\log^2 m)$ is suggested by Lemma 9, which can be found by the similar recurrence we used in Section IV-E. The number of processors needed is bounded by n. Thus, we have completed the proof of Theorem 8.

E. The Complexity of the Algorithm

Theorem 9: The LCS problem can be solved in $O(\log^2 m \log m)$ time with $mn/\log^2 m \log\log m$ processors, when $\log^2 m \log\log m > \log n$, or otherwise in $O(\log n)$ time with $mn/\log n$ processors.

To prove this, we need to examine the complexity for each of the four phases. We deal with only Phase 1 and Phase 2; the discussion of Phase 1 is applicable to Phase 3 and Phase 4. We have proved in Section II-C that Phase 1 can be done in $O(\log n)$ time with $mn/\log n$ processors. To be consistent with Theorem 9, we just point out that when $\log^2 m \log\log m > \log n$, instead of using $mn/\log n$ processors, we can use $mn/\log^2 m \log\log m$ processors only. By applying Brent's principle, the procedure for Phase 1 can be simulated by using $mn/\log^2 m \log\log m$ processors in $O(\log^2 m \log\log m)$ time.

As for Phase 2, we shall show that it can be done in $O(\log^2 m \log\log m)$ time with $mn/\log^2 m \log\log m$ processors. From this result, by applying Brent's principle, readers can easily see that $O(\log n)$ time suffices for Phase 2 if $mn/\log n$ processors are used, where $mn/\log n < mn/\log^2 m \log\log m$.

Consider the ith stage of $O(\log m)$ stages in Phase 2. We have $O(m/2^i)$ grid DAG's to deal with, each of size $O(2^i) \times (n+1)$. Denote $O(m/2^i)$ as g_i. By Corollary 3, for any of those DAG's, the corresponding cost matrix can be computed in t_i time by using p_i number of processors, where $t_i = O(\log(2^i)\log\log(2^i))$ and $p_i = 2^i n/\log^3(2^i)$. Since a total of P number of processors are available, where $P = mn/\log^2 m \log\log m$, we can compute s_i cost matrices simultaneously, where $s_i = P/p_i$. On the other hand, because there are a total of g_i cost matrices to compute, when $g_i > s_i$ holds (i.e., $i < \sqrt[3]{\log^2 m \log\log m/\log^3 2}$) $O((g_i/s_i)t_i)$ time suffices, and when $g_i \leq s_i$ holds, t_i time is enough for this stage. This discussion, together with the fact that there are $O(\log m)$ stages in Phase 2, results in the time bound T for Phase 2, which can be found as follows:

$$\begin{aligned} T &\leq \sum_{i=1}^{i_1} c_1(g_i/s_i)t_i + \sum_{i=i_1}^{c_2 \log m} t_i \\ &\leq c_1' \log^2 m \log\log m \sum_{i=1}^{i_1} \frac{\log i}{i^2} + \sum_{i=i_1}^{c_2 \log m} \log m \log\log m \\ &= O(\log^2 m \log\log m), \end{aligned}$$

where $i_1 = \sqrt[3]{\log^2 m \log\log m/\log^3 2}$. Thus, we have proved that $O(\log^2 m \log\log m)$ time suffices for Phase 2 using $mn/\log^2 m \log\log m$ processors.

VI. CONCLUSION

We gave two CREW-PRAM algorithms for the LCS problem, based upon exploiting nice properties of the problem. One algorithm is very fast and efficient, and the other is optimal in the sense of achieving the lower bound of *time* × *processors*.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewer for his/her helpful comments.

REFERENCES

[1] A. Aho, D. Hirschberg, and J. Ullman, "Bounds on the complexity of the longest common subsequence problem," *J. Assoc. Comput. Mach.*, vol. 23, no. 1, pp. 1–12, Jan. 1976.

[2] A. Aggarwal and J. Park, "Notes on searching in multidimensional monotone arrays," *Proc. 29th Ann. IEEE Symp. Foundations of Comput. Sci.*, 1988, pp. 497–512.

[3] A. Apostolico, M. Atallah, L. Larmore, and S. Mcfaddin, "Efficient parallel algorithms for string editing and related problems," *SIAM J. Computing*, vol. 19, pp. 968–988, Oct. 1990.

[4] R. Brent, "The parallel evaluation of general arithmetic expressions," *J. Assoc. Comput. Mach.*, vol. 21, pp. 201–206, 1974.

[5] V. Chavatal, D. Klarner, and D. Knuth, "Selected combinatorial research problem," Tech. Rep. STAN-CS-72-292, Stanford Univ., p. 26, 1972.

[6] R. Cole, "Parallel merge sort," *Proc. 27th Ann. IEEE Symp. on Foundations of Comput. Sci.*, 1986, pp. 511–516.

[7] A. Gibbons and W. Rytter, *Efficient Parallel Algorithms.* Cambridge, UK: Cambridge University Press, 1988.

[8] W. Hsu and M. Du, "New algorithms for the LCS problem," *J. Comput. Syst. Sci.*, vol. 29, pp. 133–152, 1984.

[9] D. Hirschberg, "Algorithms for the longest common subsequence problem," *J. Assoc. Comput. Mach.*, vol. 24, no. 4, pp. 664–675, Oct. 1977.

[10] H. Lin and M. Lu, "Solving the longest common subsequence problem on the CREW-PRAM machine," Tech. Rep. TAMU–ECE–93–09, Dept. of Elec. Eng., Texas A&M Univ., College Station, TX, 1993.

[11] M. Lu, "A parallel algorithm for longest-common-subsequence computing," *Proc. Int. Conf. Computing and Inform.*, 1990, pp. 372–377.

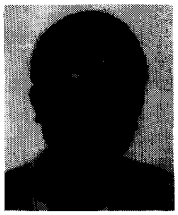
[12] T. Mathies, "A fast parallel algorithm to determine edit distance," Tech. Rep. CMU-CS-88-130, Dept. of Comput. Sci., Carnegie Mellon Univ., Pittsburgh, PA, 1988.

[13] J. Modelevsky, "Computer applications in applied genetic engineering," *Advances in Applied Microbiology*, vol. 30, pp. 169–195, 1984.

M. Lu (S'87–M'87–SM'94) received the B.S.E.E. degree from the Shanghai Institute of Mechanical Engineering, People's Republic of China, in 1981, and the M.S. and Ph.D. degrees in electrical and computer engineering from Rice University, Houston, TX, USA, in 1984 and 1987, respectively.

She joined the Department of Electrical Engineering, Texas A&M University in 1987, where she is currently an Associate Professor. Her research interests include parallel computing, distributed processing, parallel computer architectures and applications, computational geometry, and very large scale integration algorithms.

Dr. Lu has published more than 50 technical papers in her areas of research, and is a member of the IEEE Computer Society.



H. Lin received the B.S. and M.S. degrees in electrical engineering from Fudan University, People's Republic of China, in 1983 and 1986, respectively.

Beginning in 1986, he taught for three years in the Department of Electronics Engineering at Fudan University as a Lecturer, and he is currently a Ph.D. candidate in the Department of Electrical Engineering at Texas A&M University, College Station, TX, USA. His research interests include the design and analysis of parallel algorithms for combinatorial optimization problems, and the parallizing compiler.