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 $mn / \log m$ 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 $A$ 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 "like" and "alligator." In general, there may exist more than one longest subsequences for two strings. Given two strings $A$ and $B$ with length $m$ and $n$, 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
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 \( i \)th row and the \( j \)th 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 \( A = a_1, a_2, \ldots, a_m \) and string \( B = b_1, b_2, \ldots, b_n \), 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.

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>7</th>
<th>9</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>9</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>9</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>9</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>9</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>9</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 2. Examples of cost matrix.**

A vertex \( v \) on the bottom row is the \( j \)th 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 \( j \)th breakout vertex, say \( w \), on the bottom row on \( G \) must have cost \( j \), if \( v \) does have the \( j \)th 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 \( j \)th 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 \( j \)th breakout vertex of vertex \((1, i)\), and \( D_G(i, j) = \infty \) if vertex \((1, i)\) does not have a \( j \)th 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 \( D_G \), we denote the \( i \)th 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 \( D_G \), for \( 1 \leq i \leq m \), where \( G_i \) is a \( 2 \times (n+1) \)-grid DAG consisting of the \( i \)th 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 2, 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 \( h \)-th and \( (h + 1) \)-th rows of G. Let \( b_{j_1}, b_{j_2}, \ldots, b_{j_k} \) be the \( j, k \)-th symbols in B, where \( j_1 < j_2 < \cdots < j_k \). 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 \((1, j)\), the \( j \)-th vertex on the top row of \( G_h \), has no breakout vertex at all; and, finally, any other vertex \((1, j)\), \( 1 < j < j_r \), 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 \( j_k - 1 < j < j_k \). In other words, the values of entries from \( D_{G_h}(1, 1) \) to \( D_{G_h}(j_1, 1) \) are \( j_1 \), the values of entries from \( D_{G_h}(1, 1) \) to \( D_{G_h}(j_k, 2) \) are \( j_k + 1 \), and so on. For those entries \( D_{G_h}(j, 1) \) where \( j_r < j < n \), the value is \( \infty \). 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 \( D_{G_1} = (4, 4, 4, 5, 6, 8, 8, \infty, \infty, \infty, \infty, \infty)^T \).

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 \( D_{G_h}(k, 1) = \sum_{j=1}^{k} D_{G_h}(j, 1) \), for \( 1 < k < j_r + 1 \).
3) Assign \( \infty \) 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 \( p = (v_1, v_2, \ldots, v_t) \) 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 \( D_{G_1} \) and \( D_{G_L} \). Before proposing a formula for computing \( D_G \) from \( D_{G_1} \) and \( D_{G_L} \), we would like to first examine their relations through the grid DAG.

Consider vertex \((1, i)\), its \( j \)-th breakout vertex, say, \((m + 1, i_2)\), and the maximum-cost path, say, \( p \), between them. Clearly, the cost of \( p \) is \( i_2 \) and \( i_2 \) is the value of entry \( D_{G_1}(i, j) \). 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 \((m + 1, i_2)\) 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 \( k \)-th breakout vertex of \((1, i)\) on \( G_U \), whereas vertex \((m + 1, i_2)\) on \( G \) is the \((k + 1)\)-th breakout vertex of \((1, i)\) on \( G_L \), respectively. In other words, \( D_{G_U}(i, k) = i_q \) and \( D_{G_L}(i, k - j) = i_2 \), when \( k \neq 0 \) and \( k \neq 1 \). From these two equations, plus \( D_{G_1}(i, j) = i_1 \), we have \( D_{G_1}(i, j) = D_{G_U}(i_q, k - j) \). 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, \( i_q = r \). Consequently, \( p_2 \), with a cost of \( j \), must be the maximum-cost path from \((m/2 + 1, i)\) to \((m + 1, i_2)\), implying that \( D_{G_L}(i, j) = i_2 \). Therefore, we have \( D_{G_1}(i, j) = D_{G_L}(i, j) \). Similarly, one can prove that when \( j = k \), we have \( D_{G_1}(i, j) = D_{G_U}(i, j) \).

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

\[
D_{G_1}(i, j) = \min_{1 \leq k \leq j} \{ D_{G_U}(i, j), D_{G_L}(i, j), D_{G_1}(D_{G_U}(i, k), j - k) \},
\]

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

As an application of Theorem 1, we can calculate \( D_{G_1}(1, 3) \) from \( D_{G_L} \) and \( D_{G_U} \), shown in Fig. 2, as follows.

\[
D_{G_1}(1, 3) = \min \{ D_{G_U}(1, 3), D_{G_L}(1, 3), D_{G_1}(D_{G_U}(1, 1), 2) \} = 4.
\]
Proof: The correctness of Theorem 1 in the normal case in which the \(j\)th 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 \(j\)th breakout vertex. To do this, it suffices to show that \(D_G(i,j)\) will be assigned \(\infty\) under this circumstance.

Obviously, when vertex \((1,i)\) does not have a \(j\)th breakout on \(G\), neither vertex \((1,i)\) on \(G_U\) nor vertex \((1,i)\) on \(G_1\) has its \(j\)th breakout vertex. So, the first two items, \(D_G(i,j)\) and \(D_G(i,j)\), in the above formula must be \(\infty\), which leaves us to show that \(D_{G_2}(i,j,k), j-k = \infty\) for \(1 \leq k \leq j\). For a contradiction, we assume the existence of \(k\), \(1 \leq k \leq j\), such that \(D_{G_2}(i,j,k) = i_1\) and \(i_1 = \infty\). Recalling the definitions in Theorem 1, we see that \(D_{G_2}(i,j)\) must also be finite. Let \(D_{G_2}(i,k) = i_2\), then \(D_{G_2}(i_2,j) = i_1\). 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 \(j\)th 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 \(D_G\) and \(D_{G_1}\) is nothing more than identifying the minima among \(O(m)\) entries, which can be done in \(O(\log m)\) time using \(m^2/\log m\) 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 \(O(\log^2 m)\) 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 \(\infty\) (see Fig. 3). Given \(D_{G_U}\) and \(D_{G_1}\), the \(n \times m/2\) cost matrices associated with \(G_U\) and \(G_1\), we define \(n\) matrices \(M[D_G]\), for \(1 \leq i \leq n\), as follows (see also Fig. 4).

1. The size of \(M[D_G]\) 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 \(j\)th row of \(M[D_G]\) is \(CY[j+1, D_{G_U}(i,1)\) (here \(D_{G_U}(i,1))\) is the \(j\)th row of \(D_{G_U}\) with \(l = D_{G_U}(i,1))\).

As an example, let \(D_{G_U}\) and \(D_{G_1}\) be two matrices shown in Fig. 2. Fig. 5 shows matrices \(M[D_G]\) and \(M[D_G]\), respectively. Using matrix \(M[D_G]\), Theorem 1 can be rewritten as follows.

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

\[
DG(i,j) = \min_{1 \leq k \leq l} \{ DG_U(i,j), DG_1(i,j), M[D_G](k,j) \},
\]

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 \(M[D_G]\).

III. PROPERTIES OF \(D_G\) AND \(M[D_G]\)

The objective of this section is to illustrate the information redundancy in \(D_G\) and \(M[D_G]\), called \(k\)-variant, as well as another property of \(M[D_G]\), 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. \(DG(i,j) < DG(i,j_2)\) if \(j_1 < j_2\) and \(DG(i,j_1) \neq \infty\).
2. \(DG(i,j) \leq DG(i,j_2)\) if \(i_1 < i_2\).
3. \(DG(i+1,j) \leq DG(i,j+1)\).
4. If \(DG(i,j_1) = k\) and \(k \neq \infty\), then there exists \(j_2\) such that \(DG(i,j_2) = k\) for any \(i_1 < i_2 < k\).
5. If \(DG(i,j_1) = DG(i,j_2) = k_1\), then there exists \(j\) such that \(DG(i,j) = k_1\) for any \(i, i_1 < i_2, k_1 < k_2\).

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 \(M[D_G]\) are nothing but copies of rows in \(D_G\), they also suggest the similarity among \(M[D_G]\)'s. The following definitions are helpful in formalizing the information redundancy in \(D_G\) and in \(M[D_G]\). 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, \(W_1, W_2, \ldots, W_t\), 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 \(D_G\) and \(D_G\) shown in Fig. 2 are 1-variant.
Fig. 5. $M[D_1^0]$ and $M[D_2^0]$.

Proposition 2:
1) $D_2^i$ and $D_2^{i+1}$, the $i$th and $(i+1)$th row of $D_C$, are 1-variant for $1 \leq i \leq n$.
2) Any $k+1$ consecutive rows of $D_C$ are $k$-variant.

Proof: Proposition 2(2), so we prove only the latter. Without loss of generality, we assume that $\exists$ exists in both $D_2^i$ and $D_2^{i+1}$.
We also assume that there are two numbers, $l_1$ and $l_2$, such that $D_2(l_1, l_1) \neq \infty$, $D_2(l_1, l_1 + 1) = \infty$, $D_2(l_1 + 1, l_2) \neq \infty$, and $D_2(l_1 + 1, l_2 + 1) = \infty$. To show that $D_2^i$ and $D_2^{i+1}$ 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.

The existence of those $R[i,j]$ is certain by Proposition 1(5).

The third case could not occur, according to Proposition 2(1). Proposition 2(1) holds in the first case. In fact, $D_2^i$ is identical to $D_2^{i+1}$ when $D_2(i, 1) = D_2(i+1, 1)$. Indeed, for instance, if $D_2(i, 2)$ is not $\infty$, then there exists $j_2$ such that $D_2(i + 1, j_2) = D_2(i, 2)$ according to Proposition 1(4), where $j_2$ must be less than or equal to 2. On the other hand, if $j_2$ should not be 1, because $D_2(i + 1, 1) = D_2(i, 1) \neq D_2(i, 2)$.

Therefore, we conclude that $j_2 = 2$; i.e., $D_2(i + 1, 2) = D_2(i, 2)$. Similarly, one can show that if $D_2(i, 3)$ is not $\infty$, then $D_2(i + 1, 3) = D_2(i, 3)$, and so on. As for the second case, except for $D_2(i, 1)$, all entries in $D_2^i$ must also be in $D_2^{i+1}$. Indeed, since $i + 1 \leq D_2(i, 1) = D_2(i + 1, j)$ for $2 \leq j \leq l_1$ (by Proposition 1(1)), entry $D_2(i, j)$ can be found in $D_2^{i+1}$ according to Proposition 1(4).

To capture the useful information in $D_2^i$ and in $M[D_2^i]$, we introduce several concepts. Consider $k+1$ consecutive row-vectors in $D_C$, say, $D_2^i = \{i, i+1, \ldots, i+k\}$. Let $L$ be a common row-vector of them. $L$ can be partitioned into groups, $L_1, L_2, \ldots, L_r$, such that entries in each group are consecutive entries in every $D_2^i$. The remnant of $D_2^i$ with respect to $L$ (the remnant of $D_2^i$ for short) is a row-vector $R[D_2^i] = (R_1^i, R_2^i, \ldots, R_{i+k}^i)$ such that $D_2^i = (R_1^i, L_1, R_2^i, \ldots, L_r, R_{i+k}^i)$. Note that there may be no entry in $R_j^i$. The size of $R[D_2^i]$ is defined as the sum of sizes of $R_i^i$’s for $1 \leq j \leq r + 1$. 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 $i', i_1 \leq i' \leq i_1 + k$, such that these two entries are not consecutive in $D_2^i$. Clearly, under this assumption, the partition for groups is unique. Consider the first and second rows of $D_C$ in Fig. 2 as an example.

The common row-vector of them is $L = (7, 9, 12)$, and the remnants of them are $R[D_2^1] = (2)$ and $R[D_2^2] = (3)$, respectively.

Proposition 3:
1) For $k+1$ consecutive row-vectors $D_2^i$ of $D_C$, $i_1 \leq i \leq i_1 + k$, common row-vector $L$ of $D_2^i$ and $D_2^{i+k}$ is a common row-vector of the $k+1$ row-vectors.
2) $L$ can be partitioned into at most $2k$ groups.
3) Let $R[D_2^i] = (R_1^i, \ldots, R_{i+k}^i)$ be the corresponding remnant of $D_2^i$, where $r > 0$. Then $R_1^i$ is a subvector of $R_2^i$ when $i_1 < i_2$ and $2 \leq j \leq r + 1$.

Proof: To show that $L$ is the common row-vector of $D_2^i$’s, for $i_1 \leq i \leq i_1 + k$, we need to show only the following:
1) Any “finite” entry in $L$ appears in every $D_2^i$.
2) The number of “$\infty$” entries in $L$ are no more than that in any $D_2^i$’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 $D_2(i_1 + 1, j_2) = D_2(i_1, j_2) = w$. Indeed, $D_2(i_1 + 1, j_2) = w$, then for any $i, i_1 < i < i_1 + k$, there exists $j$ such that $D_2(i, j) = w$ by (Proposition 1(5)). Statement 2) is true because, in fact, Proposition 1(2) suggests that there are at least as many $\infty$ entries in $D_2^i$ as in $D_2^i$ for $i_1 < i$.

$L$ can be obtained from either $D_2^i$ or $D_2^{i+k}$ 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 $D_2^i$ and $D_2^{i+k}$. Now, to show Proposition 3(2), we shall show that the partition is valid for $D_2^i$’s for $i_1 < i < i_1 + k$. That is, if $w_1, w_2, \ldots, w_{h}$ are $h$ consecutive entries in both $D_2^i$ and $D_2^{i+k}$ under this partition, then they are also consecutive in any $D_2^i$’s for $i_1 < i < i_1 + k$.

The existence of those entries in $D_2^i$’s is certain by Proposition 1(5). To show the consecutivity, we assume that there exists entry $w$ in $D_2^i$ such that $w_1 < w < w_{i+k+1}$, $1 \leq l < h$, for a contradiction. Since $w_1$ is an entry in $D_2^{i+k}$, we have $i + k + 1 < w$. Because of $i < i_1 + k$, $i_1 + k < w_1$, and $w_1 < w$, we have $i < i_1 + k < w$.

By Proposition 1(4), $w$ must be an entry also in $D_2^{i+k}$. The assumption that $w$ is between $w_1$ and $w_{i+k+1}$, thus implying that $w_1$ and $w_{i+k+1}$ are not consecutive in $D_2^{i+k}$, 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 $D_2^i$ of any $k+1$ consecutive rows of $D_C$ 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 $M[D_2^i]$

The information redundancy in matrix $M[D_2^i]$, along with another important property of $M[D_2^i]$, called totally monotone, is illustrated in this section.
Let \( X = (X^1, X^2, \ldots, X^m) \) and \( Y = (Y^1, Y^2, \ldots, Y^m) \) be two matrices of the same size, where \( X^i \) and \( Y^i \) are the \( i \)th 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 \( X = (Y^{m+k-1}, Y^1, Y^{m-k+1}, Y^{m-k}) \). Let \( Y_1, Y_2, \ldots, Y_k \) be matrices, all with same number of columns. By \( (Y_1, Y_2, \ldots, Y_k)^T \), we denote the following matrix in this paper: 

\[
\begin{pmatrix}
Y_1 \\
Y_2 \\
\vdots \\
Y_k
\end{pmatrix}
\]

\( M \) is a common matrix of \( M_1, M_2, \ldots, M_r \), denoted by \( M_{D2'} \), if there exists a common matrix \( M \) with all \( M_i \)'s, but also the rows in \( M \) are not in the common matrix. We can see that the former is obtained from \( M_{D2} \), Comparing these two copies \( M_{D2} \) by \( k \)-shift. Because of the lack of space, it can be handled similarly. Indeed, by the definition of \( M[D_{2'}] \), the \( w_1 \)th row of \( M[D_{2'}] \) is \( CY[w_1 + 1, D_{Gz}(i_1, w_1)] \). On the other hand, the first row of \( M_j \) is \( CY[1, D_{Gz}(i_1, w_1)] \), as defined in the theorem, which is equivalent to \( CY[1, D_{Dz}(i_1, w_1)] \), because \( L_1(1) = D_{Gz}(i_1, w_1) \). Comparing these two copies of \( D_{Gz}(i_1, w_1) \), one can see that the former is obtained from the latter by the \( w_1 \)th-shift.

To prove that at most \( k \) rows of \( M[D_{2'}] \) are not in \( (M_1, M_2, \ldots, M_r)^T \), we need to notice only the fact that there are at most \( k \) entries that are in \( D_{Gz}(i_1, w_1) \), but not in \( L \).

**Corollary 2:** Any \( k + 1 \) consecutive matrices, \( M[D_{Gz}], M[D_{Gz}^{-1}], \ldots, M[D_{Gz}^{-k}] \), for \( 1 \leq i \leq n - k \), 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 \( M[D_{2'}] \) can be seen in the next two sections, where we shall show how we can get rid of redundant information in \( M[D_{Gz}] \) and thus create a new matrix that has the same column minima as \( M[D_{2'}] \), but contains far fewer rows. It is worth pointing out that in both of our algorithms, matrix \( M[D_{2'}] \) is never physically generated for the obvious reason: It costs too much. Even matrix \( M \), the common matrix of \( M[D_{Gz}] \), 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 \( M[D_{2'}] \). A 2-D matrix \( Z \) is **monotone** if the minimum value in its \( i \)th 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 \( 2 \times 2 \) submatrix of it is monotone [2]. As an example, matrix

\[
\begin{pmatrix}
4 & 6 & 4 \\
5 & 4 & 3
\end{pmatrix}
\]

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:** \( M[D_{Gz}] \) is totally monotone.

**Proof:** Consider an arbitrary \( 2 \times 2 \) submatrix of \( M[D_{Gz}] \):

\[
\begin{pmatrix}
M[D_{Gz}](i_1, j_1) & M[D_{Gz}](i_1, j_2) \\
M[D_{Gz}](i_2, j_1) & M[D_{Gz}](i_2, j_2)
\end{pmatrix}
\]

where \( i_1 < i_2 \) and \( j_1 < j_2 \). To prove that \( M[D_{Gz}] \) is totally monotone, it suffices to prove that the above submatrix is monotone, that is, the following:

\[
M[D_{Gz}](i_1, j_1) > M[D_{Gz}](i_2, j_1) \Rightarrow M[D_{Gz}](i_1, j_2) > M[D_{Gz}](i_2, j_2)
\]  
(1)

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:

\[ M[D_G^i](i_1,j_1) > M[D_G^i](i_2,j_1) \]
\[ \Rightarrow M[D_G^i](i_1,j_1 + 1) > M[D_G^i](i_2,j_1 + 1). \]  

(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:

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

(3)

Let \( v_1, v_2, \ldots, v_{m/2} \) be the entries of \( D_{G_0} \) from the first one to the \((m/2)\)th one. Then the \( i \)th row of \( M[D_G^i] \) is \( CY[i_1 + 1, D_G^i] \) by definition. Keeping this in mind, one can see that (3) is equivalent to the following:

\[ DG_1(v_{i_1}, j_1 - i_1) > DG_1(v_{i_2}, j_1 - i_2), \]  

(4)

and

\[ DG_1(v_{i_1}, j_1 - i_1 + 1) \leq DG_1(v_{i_2}, j_1 - i_2 + 1). \]  

(5)

\( v_{i_2} < DG_1(v_{i_1}, j_1 - i_1) \) follows immediately from (4), because \( v_{i_2} \) is always strictly less than \( DG_1(v_{i_2}, j_1 - i_2) \). Thus, there exists \( j' \) such that the following is true:

\[ DG_2(v_{i_1}, j') = DG_1(v_{i_1}, j_1 - i_1). \]  

(6)

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

\[ DG_1(v_{i_1}, j_1 - i_1) < DG_1(v_{i_2}, j_1 - i_2 + 1). \]  

(7)

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

\[ DG_1(v_{i_1}, j_1 - i_2) < DG_1(v_{i_2}, j') < DG_1(v_{i_1}, j_1 - i_2 + 1), \]

which contradicts the fact that \( DG_1(v_{i_2}, j_1 - i_2) \) and \( DG_1(v_{i_1}, j_1 - i_2 + 1) \) are two consecutive entries of \( D_{G_1}^{i_1}. \)

Now we consider the cases in which some entries in (2) are \( \infty \). In order to be consistent with the definition of monotone matrix, we have to distinguish \( \infty \) entries. For any two \( \infty \) 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 \( \infty \) 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 \( \infty \), 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). \( \square \)

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 \( D_{G_0} \) 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 \( D_{G_0} \) and \( D_{G_L} \)

Recall the basic formula for computing \( D_G \) from \( D_{G_0} \) 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 \( D_G \) 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 \( D_{G_0} \) 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 \( D_{G_0} \) 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 \( D_{G_0}(i,j), D_{G_L}(i,j) \) and \( j \)th column minimum of \( M[D_G^i] \).

To show Theorem 5, we need to show only the following.

Lemma 1: Given cost matrix \( D_{G_0} \) and \( D_{G_L} \), the column minima of \( M[D_G^1], M[D_G^2], \ldots, M[D_G^{\log^2 m}] \) 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 \( D_{G_0} \) and \( D_{G_L} \). Procedure ColMin takes advantage of the information redundancy in \( M[D_G^i] \). According to Corollary 1, matrices \( M[D_G^1], M[D_G^2], \ldots, M[D_G^{\log^2 m}] \) 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 \(M[D^c_G]\) 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 \(M = (M_1, \ldots, M_r)^T\) be the common matrix of \(M[D^c_G]\) 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 \(M[D^c_G]\), corresponding to \(M_j\), i.e., \(X[i,j] = S[K(i,j), M_j]\). By vector \(C_{min}[Z]\), we denote the column minima of matrix \(Z\), i.e., \(C_{min}[Z](i)\) is referred to as the minimum of the \(i\)th column of matrix \(Z\).

In procedure ColMin, matrix \(M[D^c_G]\) is reduced to a new matrix, denoted by \(M'[D^c_G]\), by replacing its submatrix \(X[i,j]\) with \(C_{min}[X[i,j]]\). It should be clear that the new matrix shares the same column minima with the original one, i.e., \(C_{min}[M[D^c_G]] = C_{min}[M'[D^c_G]]\). 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 \(M'[D^c_G]\)'s is bounded by \(3\log^2 m\), instead of \(m/2\) for the original ones. Second, to generate \(M'[D^c_G]\)'s for \(1 \leq i \leq \log^2 m\), we need to compute \(C_{min}[M_j]\)'s only for \(1 \leq j \leq r\), instead of computing \(C_{min}[X[i,j]]\)'s for \(1 \leq i \leq \log^2 m\) and \(1 \leq j \leq r\), independently. This is because \(C_{min}[X[i,j]]\) can be obtained from \(C_{min}[M_j]\) by \(K(i,j)\)-shift, because \(X[i,j] = S[K(i,j), M_j]\). Finally, \(M'[D^c_G]\)'s are totally monotone. Next additional notations are given before procedure ColMin is presented.

Let \(L = (L_1, L_2, \ldots, L_r)\) be a common row-vector of row-vectors \(W_i\), for \(1 \leq i \leq l\), where \(L_j\) is the \(j\)th 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 \(L_j(1)\). \(P[W_i, L_j]\) is defined as a function such that \(P[W_i, L_j]\) can be identified in \(O(1)\) time with \(\log m\) processors. Similarly, let \(Y = (Y_1, Y_2, \ldots, Y_T)^T\) be a common matrix of \(Z_i\)'s for \(1 \leq i \leq l\), where \(Y_j\) is the \(j\)th 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 of \(Y_j\) by certain shifting. \(P[Z_i, Y_j]\) is defined as a function such that \(P[Z_i, Y_j]\) can be identified in \(O(1)\) time with \(\log m\) processors.

Procedure ColMin

\[
\text{Input: cost matrix } D^G_G, \text{ and } D^L_G. \\
\text{Output: } C_{min}[M[D^c_G]]'s, \text{ for } 1 \leq i \leq \log^2 m. \\
1) \text{Identify common row-vector } L \text{ of } D^G_G \text{ for } 1 \leq i \leq \log^2 m: \\
\quad a) \text{Compute } L = (L_1, \ldots, L_r); \text{ and } \\
\quad b) \text{Compute } P[D^G_G, L], \text{ for } 1 \leq i \leq \log^2 m. \\
2) \text{Identify common matrix } M \text{ of } M[D^c_G] \text{ for } 1 \leq i \leq \log^2 m: \\
\quad a) \text{Compute } P[M[D^c_G], M], \text{ for } 1 \leq i \leq \log^2 m; \text{ and } \\
\quad b) \text{Compute } K(i,j) \text{ such that } X[i,j] = S[K(i,j), M_j], \text{ for } 1 \leq i \leq \log^2 m \text{ and } 1 \leq j \leq r. \\
3) \text{Compute } C_{min}[M_j], \text{ for } 1 \leq j \leq r; \\
4) \text{Compute } C_{min}[M[D^c_G]] \text{ for } 1 \leq i \leq \log^2 m. \\
\]

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

\[
P[D^G_G, L] \text{ is to be computed by identifying the position of } L_j(1) \text{ in } D^G_G, \text{ for } 1 \leq j \leq r. \\
\text{Since the number of groups } r \text{ is bounded by } \log^2 m, \text{ (see Corollary 1), and since there are } \log^2 m, \text{ such functions to be computed, we use } \log^4 m. \\
\text{Lemma 3: Given common row-vector } L = (L_1, \ldots, L_r) \text{ of } D^G_G, \text{ for } 1 \leq i \leq \log^2 m, \text{ the corresponding position functions } P[M[D^c_G], M]'s \text{ and } K(i,j)'s \text{ for } 1 \leq i \leq \log^2 m \text{ and } 1 \leq j \leq r \text{ can be identified in } O(1) \text{ time with } \log^4 m, \text{ processors.} \\

\text{As a matter of fact, once } L = (L_1, L_2, \ldots, L_r), \text{ obtained, } M = (M_1, \ldots, M_r)^T \text{ can be constructed following the rule in Theorem 3. As for the computation of } P[M[D^c_G], M], \text{ recall that the } (Pos[D^G_G, L_j]) \text{th row of } M[D^c_G] \text{ is } CY[Pos[D^G_G, L_j] + 1, D^G_G] \text{ (from the definition of } M[D^c_G]), \text{ and also recall that the first row of } M_j \text{ is } CY[1, D^G_G] \text{ (from Theorem 2). So, the former can be obtained from the latter by } Pos[D^G_G, L_j] \text{-shift, implying } Pos[M[D^c_G], M_j] = Pos[D^G_G, L_j]. \text{ Thus, in general, we have } P[M[D^c_G], M] = P[D^G_G, L]. \text{ Since } P[D^G_G, L] \text{ 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_G, L_j]. \text{ Constant time with } \log^4 m \text{ processors should suffice to identify the common matrix of } M[D^c_G]'s. \]
Lemma 4: Given $D_G$, $D_L$, $P[M[D_G]]$, $M$, and $K(i,j)$'s for $1 \leq i \leq \log_2 m$ and $1 \leq j \leq r$, $C_{\min}[M_j]$'s can be computed in $O(\log n)$ 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 $C_{\min}[M_j]$. For the worst case, we assume that it is of size $(m/\log^2 m) \times (m/\log^2 m)$ 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 $C_{\min}[Z]$. 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 $M$ can be computed in $O(\log m)$ time, also using $m/\log m$ processors.

In total, all $C_{\min}[M_j]$'s for $1 \leq j \leq r$ can be computed in $O(\log m)$ time with $m \log m$ processors.

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

Instead of computing $C_{\max}[M[D_G]]$ directly, we compute $C_{\min}[M[D_G]]$, a much smaller matrix. Recall that $M'[D_G]$ consists of row-vectors $C_{\max}[X[i,j]]'$s for $1 \leq j \leq \log^2 m$ and at most $\log^2 m$ other row-vectors from $M[D_G]$. Also recall that $C_{\max}[X[i,j]]'$ is obtained from $C_{\min}[M_j]$ by $K(i,j)$-shift. The construction of $M'[D_G]$'s should be easy under the assistance of $P[M[D_G]]$, $M$. Since $M'[D_G]$'s are totally monotone, the similar approach we just used for computing $C_{\min}[M_j]$'s can be used to compute $C_{\min}[M'[D_G]]$'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, $p = (v_1, v_2, \ldots, v_l)$, 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 $i$th row of $G$ by $v[i]$ 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 $y_j$ 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 between $G_U$ 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 $D_G(1,j), D_G(i,j), D_G(i-1,k) + j - k$'s, for $1 \leq k \leq j$. If $D_G(i,j) = D_G(1,j)$, then obviously $q$ is vertex $(m/2 + 1, D_G(i,j))$ on $G$. If $D_G(i,j) = D_G(i,j)$, then $q$ is vertex $(m/2 + 1, i)$ on $G$. Otherwise, suppose that $D_G(i,j) = D_G(i-1,k) + j - k$; then $q$ is the $(m/2 + 1, D_G(i,k))$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 $(i,j)$ and vertex $(i+1,j)$ on $G$, respectively. Then vertices on the $i$th row of $G$ from $(i,j+1)$ to $(i,j+2)$ 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 $mn/\log n$ 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 \( O(m^2) \) cost matrices, each of size \( n \times O(2^k) \) at \((i+1)\)th merge stage of Phase 2, where each matrix corresponds to an \( O(2^k) \times (n + 1) \) 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 \( L_i \), and use \( k + 1 \) 1-D arrays to represent \( k + 1 \) remnants \( R[D_G^i] \)’s. Besides, position functions \( P[D_G^i, L_i] \)’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:

\[
L = (4, 5, 13, \infty, \infty, \infty),
\]

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

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 row vector \( L \) with a size of at most \( m \), and \( k + 1 \) remnants, each with size of at most \( n \); hence, \( O(mn/k + nk) \) space suffices to store all information in \( D_G \). By this representation, not only 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 \( h \)th group of \( L \), the common row-vector of \( D_G^i \)’s for \( i_1 \leq i \leq i_1 + k \).

1. \( \text{Pos}[D_G^{i_1 + k}, L_h] \leq \text{Pos}[D_G^{i_1}, L_h] \leq \text{Pos}[D_G^{i_1 + k}, L_h] \).
2. \( \text{Pos}[D_G^{i_1 + k}, L_h] \geq \text{Pos}[D_G^{i_1}, L_h] + k \).

B. The Algorithm for Computing \( D_G \)

We first establish the time bound for computing \( D_G \) from \( D_G^1 \) and \( D_G^2 \), and then provide the proof in the next subsection.

Theorem 7: Given \( D_G^1 \) and \( D_G^2 \), \( O(\log m \log \log m) \) time with \( m \log m \) processors suffices for computing \( \log^3 m \) consecutive row-vectors in cost matrix \( D_G \) of size \( n \times m \).

Corollary 3: Given \( D_G^1 \) and \( D_G^2 \), \( O(\log(m \log \log m)) \) 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 \( m \log^4 m \) 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 \( D_G^i \)’s for \( 1 \leq i \leq \log^4 m \) as follows:
   a. Compute \( D_G^1 \) and \( D_G^2 \).
   b. Compute common row-vector \( L \) of \( D_G^1 \), \( D_G^2 \).
   c. Compute corresponding position functions \( P[D_G^1, L] \) and \( P[D_G^2, L] \).
2. Compute remnants \( R[D_G^1] \), for \( 1 \leq i \leq \log^4 m \).
3. Compute position functions \( P[D_G^1, L] \) and \( P[D_G^2, R[D_G^1]] \), 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 \( D_G^1 \), \( D_G^2 \) is the common row-vector of the \( \log^4 m \) rows (see Proposition 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 \( m \log^4 m \) 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 \( R[D_G^1] = \)

Proof: Let \( \text{Pos}[D_G^{i_1 + k}, L_h] = j_1 \), \( \text{Pos}[D_G^{i_1}, L_h] = j_2 \), and \( \text{Pos}[D_G^{i_1 + k}, L_h] = j_3 \). In other words, \( D_G(i_1, j_1) = D_G(i_1, j_2) = D_G(i_1 + k, j_3) = L_h(1) \). 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 \( \text{Pos}[D_G^{i_1 + k}, L_h] \geq j_1 - 1 \). Applying Proposition 1(3) repeatedly, we get \( \text{Pos}[D_G^{i_1 + k}, L_h] \geq j_1 - k \), which is equivalent to \( \text{Pos}[D_G^{i_1 + k}, L_h] \geq \text{Pos}[D_G^1, L_h] - k \).
(R_{1}, \ldots, R_{r+1}) be the remnant of D_{G}. We shall show only how to compute remnant group R_{a}, the bth group of R[D_{G}], where 1 \leq a < \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 \infty; the case in which L contains only \infty\'s should be easy to deal with. Two facts should be noted: Any finite entry in R_{a}, b \geq 2, exists also in R_{b}^{\log^{4} m} (see Proposition 3(3)), and \infty entries are always on the right side of finite entries in D_{G}. Thus, removing \infty entries from R_{b}^{\log^{4} m} will not affect generating finite entries in R_{a}. Without loss of generality, we assume that there is no \infty entry in R_{b}^{\log^{4} m}.

Note that since D_{G}, D_{G}^{\log^{4} m} and L have already been computed, it should be easy to compute R[D_{G}] and R[D_{G}^{\log^{4} m}]. Other remnants are computed from them. There are two cases to be considered: b = 1 and b < 2. By Proposition 3(3), when b \geq 2R_{a} is a subvector of R_{b}^{\log^{4} m}, which, together with the monotonality of D_{G}, implies that R_{b}^{R_{a}} must be the largest common subvector of D_{G} and R_{b}^{\log^{4} m}. Therefore, if D_{G} is given, then R_{b} can be obtained by identifying the largest common subvector of D_{G}^{\log^{4} m} and R_{b}^{\log^{4} m}. Unfortunately, D_{G}^{\log^{4} m} is unknown. Instead of D_{G}, we are going to compute a subrow of D_{G}^{\log^{4} m}, denoted as SD[a, b], such that 1) \{R_{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}, 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}, and the following time bound should be quite clear.

Lemma 6: Suppose that SD[a, b] is given, where b \geq 2, and that the size of SD[a, b] is bounded by O(\log^{4} m). R_{b} can be computed in O(\log^{4} 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} is bounded by log^{4} m. So, we define SD[a, 1] as the first log^{4} m entries of D_{G}. Let k be the size of R_{1}, then R_{1} 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}. From this, we draw the following conclusions.

Lemma 7: Suppose that SD[a, 1] and L(1) are given, then log^{4} m processors suffice to identify R_{1} 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 SD[i, j] and L(1) are given, and that the size of SD[i, j] is bounded by log^{4} m. Then R_{i}S, for 1 \leq i < \log^{4} m and 1 \leq j \leq r, can be obtained in O(\log^{4} m) time by using log^{4} 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} 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}.) We first discuss the issue of which entries of D_{G} should be included in SD[a, b]. A vector, Ind[SD[a, b]], is used to record their original positions in D_{G} for entries in SD[a, b], as follows:

Ind[SD[a, b]](i) = l if SD[a, b](i) = D_{G}(l).

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

l \leq Pos[SD[a, b], R_{b}] \leq l + \log^{4} m,

where l = Pos[SD[a, b], 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}^{\log^{4} m} entries in R_{b}. Suppose that SD[a, b] consists of the first log_{4} m\cdot \log_{4} m entries in D_{G}, that is given, then Ind[SD[a, b]], therefore can be found as follows. For 1 \leq i \leq 2 \log^{4} m,

Ind[SD[a, b]](i) = Pos[SD[a, b], R_{b}^{\log^{4} m}, R_{b}^{\log^{4} m}] + i + 1. (8)

The next issue is how to compute SD[a, b] from D_{G} and D_{G}. Clearly, we need to identify only the (Ind[SD[a, b]](i))th column minima in M[D_{G}] for 1 \leq i \leq 2 \log^{4} m. Let M[SD[a, b]] be a submatrix of M[D_{G}] 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 \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}] 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}', \ldots, L_{r}') be the common row-vector of \{D_{G}^{i}'\)'s for 1 \leq i \leq 2 \log^{4} m. Let M = (M_{1}, \ldots, M_{r}) be the common matrix of M[SD[a, b]]'s, for 1 \leq i \leq 2 \log^{4} m, constructed following the rule in Theorem 3 (that is, M_{j} = C[j, i, D_{G}^{j}(i)], j = 1, \ldots, r). Again, let X[i, k] be the submatrix of M[SD[a, b]] corresponding to M_{j}. i.e., X[i, k] = S[K(i, k), M_{j}] for some K(i, k). By SM[SD[a, j], k], we refer to the submatrix of both M[SD[i, j]] and X[i, k] such that SM[SD[a, 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 Cmin[SM[SD[a, b], k]]'s for 1 \leq k \leq r. The following facts about M'[SD[a, b]] are obvious.

Proposition 5:
1) Cmin[M'[SD[a, b]]] = Cmin[M[SD[a, b]]].
2) The size of M'[SD[a, b]] is bounded by (2 \log^{4} m) \times (2 \log^{4} m).

Corollary 5: Suppose that we have already obtained M'[SD[a, j]]'s, for 1 \leq i \leq 2 \log^{4} m and 1 \leq j \leq r, which are represented by arrays, then Cmin[M[SD[i, j]]] for 1 \leq i \leq 2 \log^{4} m and 1 \leq j \leq r, can be computed
in \(O(\log \log m)\) 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 \(O(\log m \log \log m)\) 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]]’s, can be obtained from D_Gu and D_Gv 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’[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\([a, k]\)] and Cmin[X\([a, k]\)] can be obtained from Cmin[Mk] by K(a, k)-shift for 1 \(\leq k \leq r_1\), where Mk is the kth group of common matrix M (since X\([a, k]\) is obtained from Mk 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[Mk]’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 \(O(\log^2 m)\) Mk’s each with size \((m/\log^2 m) \times m\). 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[Mk]’s are useful for generating M’[SD\([a, b]\)]. We define a submatrix SMk of Mk by removing those columns of Mk that are irrelevant to generate Cmin[SM’[SD\([a, b]\), k]]’s. Let Ind[Mk] be the vector that records their original positions in Mk for each column in SMk; that is, \(\text{Ind}[M_k](h) = I\) if SMk(h) = Mk(h).

Procedure Ind[Mk]

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[Mk](h):

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

2) Sort all entries in Ind[Mk] 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[Mk] is obtained by ranking those marked entries.

Step 1 in this procedure guarantees that all those columns in Mk 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[Mk]. The step 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[Mk] is bounded by \(O(\log^4 m)\). Hence, to compute Ind[Mk]’s for 1 \(\leq k \leq r_1\), \(O(\log \log m)\) time suffices with polylogarithmic number of processors.

Since Mk is totally monotone, so is SMk. In order to calculate the time bound for computing Cmin[SMk] by applying Aggarwal and Park’s algorithm, we bring readers’ attention to the size of SMk’s. The number of columns in each SMk is bounded by \(O(\log^4 m)\); the total number of rows in all SMk’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 SMk, otherwise, we can divide the larger SMk into several smaller ones, each consisting of at most \(m/2)/\log^4 m\) rows, and the total number of SMk’s will not be more than 2r1. According to Observation 2, \(m \log m\) processors suffice to compute Cmin[SMk]’s for 1 \(\leq k \leq r_1\), \(O(\log \log m)\) time.

Since Cmin[SMk] 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 kth entry of Cmin[SM’[SD\([a, b]\), k]] is the Ind[SD\([a, b]\)](h) entry of Cmin[X\([a, k]\)], i.e., the (Ind[SD\([a, b]\)](h) - K(a, k))th entry of Cmin[Mk] if Ind[SD\([a, b]\)](h) - K(a, k) \(\geq 1\), or, otherwise, is the \((m - K(a, k) + \text{Ind}[SD[a, b](h)](h))\)th entry of Cmin[Mk].

With the help of Ind[Mk]’s, we can locate the kth entry of Cmin[SM’[SD\([a, b]\), k]] in Cmin[SMk] through a binary search. Since the number of columns in SMk 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\([a, b]\)]’s 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 I[R[D_G]],
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 \( v[l] \) 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 \( p, v[l] \), for \( 1 \leq i \leq m + 1 \), can be identified in \( O(\log^2 m) \) time with \( n \) processors.

The proof of this result needs 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 \( m > \log m \), instead of using \( mn/\log n \) processors, we can use \( mn/\log^2 m \log m \) processors only. By applying Brent’s principle, the procedure for Phase 1 can be simulated by using \( mn/\log^2 m \log m \) processors in \( O(\log^2 m \log m) \) time.

As for Phase 2, we shall show that it can be done in \( O(\log^2 m \log m) \) time with \( mn/\log^2 m \log m \) 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 \( mn/\log n < mn/\log^2 m \log m \).

Consider the \( i \)th stage of \( O(\log m) \) stages in Phase 2. We have \( O(m/2) \) grid DAG’s to deal with, each of size \( O(2^i) \times (n + 1) \). Denote \( O(2^i) \) 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 \( t_i = O((\log 2^i) \log 2^i) \) and \( p_i = 2^i n^{\log 2^i} \). Since a total of \( P \) number of processors are available, \( P = mn/\log^2 m \log m \), 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., \( i < \sqrt{\log^2 m \log m / \log 2^i} \)), \( O(g_i/s_i) t_i \) time suffices, and when \( g_i \leq s_i \), \( t_i \) 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:

\[
T \leq \sum_{i=1}^{\log_2 m} g_i / s_i t_i + \sum_{i=1}^{\log_2 m} t_i \\
\leq \sum_{i=1}^{\log_2 m} \log m \log m \sum_{i=1}^{\log_2 m} \frac{i}{2^i} + \sum_{i=1}^{\log_2 m} \log m \log m \\
= O(\log^2 m \log m),
\]

where \( i_1 = \sqrt{\log^2 m \log m / \log 2^i} \). Thus, we have proved that \( O(\log^2 m \log m) \) time suffices for Phase 2 using \( mn/\log^2 m \log m \) 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 \( t \times p \) processors.

ACKNOWLEDGMENT

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

REFERENCES


M. Lu (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 parallelizing compiler.