1 Introduction

An alphabet is a finite set of symbols. A string over an alphabet is a sequence of symbols from the alphabet. Given a string, a subsequence of the string can be obtained from the string by deleting none or some symbols (not necessarily consecutive ones). If string $C$ is a subsequence of both string $A$ and string $B$, then $C$ is a common subsequence (CS) of $A$ and $B$. String $C$ is the 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 subsequence. For example, string “tcaggt” is the longest subsequence of strings “tcaggtatctag” and “ttatgttcaggt”. Note that, in general, there 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 the longest common subsequence of $A$ and $B$. Fast solutions of this problem are requested very often in genetic engineering [Manelevsky84], data compression [AHU76][CKK72], editing error correction [HD80] and syntactic pattern recognition [CF81].

The lower bound on the time complexity of this problem has been exploited by Aho et al in [AUH76]. They showed that under the decision tree model, in which all decisions are whether or not two positions have or do not have the same symbol, the lower bound of the LCS problem is $mn$, unless the alphabet size is fixed. That is, any algorithm using “equal-unequal” comparisons for this problem will consume at least $mn$ times comparisons. There are some sequential algorithms reached this bound [Hirschberg75], [Hirschberg76] and [HD84].

In recent years, exploiting the parallelism of this problem has attracted many research interests and several parallel algorithms have been designed [AP88],[Mathies88],[AALM90],[Lu90],[LLA91]. Among them, Aggarwal and Park in [AP88], and Apostolico et al in [AALM90] have independently shown that this problem (and the more general problem called string-editing problem) can be solved in $O(\log mn/\log m)$ time using $mn/\log m$ processors on CREW-PRAM. Their approach is to relate the string-editing problem with the problem of recognizing the shortest path from source to sink on a $(m+1) \times (n+1)$ grid directed graph and use a divide-and-conquer scheme to compute the “distance matrix” which records the shortest distance from every vertices on the left (or top) boundary of the grid directed graph to every
vertices on the bottom (or right) boundary. Following this approach but more concentrating on exploiting the nice properties of the LCS problem, Lin et al in [LLA91] have defined a different distance matrix which is more compact than the previous one, and shown that, on CREW-PRAM model, \( O(\log^2 m \cdot \log n) \) time using \( mn/\log n \) processors suffices to solve the LCS problem. Since in general \( m \leq n \), this result is better than the previous ones, though in the worst case they are the same. However, in terms of the product of the time bound and the number of processors used, none of them achieve the lower bound \( mn \).

In this paper, we propose an optimal algorithm for the LCS problem in the sense that the \( time \times processors \) matches the lower time complexity bound of the problem. The computation model we used is the concurrent-read and exclusive-write (CREW) parallel random access machine (PRAM). A PRAM employs synchronous processors all having access to a common memory. A CREW-PRAM allows simultaneous access by more than one processors to the same memory location for read but not for write purpose. Our algorithm takes \( O(\log^2 m \cdot \log \log m) \) time with \( mn/\log^2 m \cdot \log \log m \) processors when \( \log^2 m \cdot \log \log m > \log n \), or otherwise takes \( O(\log n) \) time with \( mn/\log n \) processors.

This is continuation of the papers referenced above. The improvement made in this paper based on the following observation. In [LLA91], like in [AP88] and [AALM90], at each stage of the \( \log n \) stages of "conquer" we are to compute the distance matrix which records the information about the longest paths (or shortest path in [AP88] and [AALM90]) between the vertices on the upper boundary and the vertices on the bottom boundary of the graph. For a sub-grid directed graph of size \((n+1) \times (m_1+1)\), the size of the corresponding distance matrix is \( m_1 \times n \). So suppose that at the \( i \)-th "conquer stage" the \((m_1+1) \times (n+1)\) grid directed graph has been divided into \( O(m/2^i) \) sub-grid directed graphs of size \( 2^i \times (n+1) \), we have to compute \( O(mn) \) entries of these matrices. Totally, we have to compute \( O(mn \cdot \log m) \) entries. This suggests that any attempt of computing and recording the distance matrices directly will destroy the hope of achieving the optimal algorithm. By using some nice properties of the distance matrix, we invent an efficient data structure to represent the distance matrix such that any entry of the distance matrix can be obtained from the data structure very fast and the size of the data structure representing an \( n \times m \) distance matrix is much smaller than that of an \( n \times m \) 2-dimensional array representing the same distance matrix.

The remainder of the paper is organized as follows: In section 2, we show how the LCS problem can be viewed as the longest path problem on the grid directed graph, establish the main structure of our algorithm, and introduce some basic ideas used in "conquer stage". Section 3 concentrates on exploiting the properties of the distance matrix. In section 4, these properties are applied to achieve an optimal algorithm.

## 2 Solve the LCS problem through grid DAG

### 2.1 The grid directed acyclic graph

An \( l_1 \times l_2 \) grid DAG is a directed acyclic graph whose vertices are the \( l_1 \times l_2 \) grid points of an \( l_1 \times l_2 \) grid. The only edges from grid point \((i, j)\), referred to as vertex \((i, j)\), are to vertices \((i, j+1), (i+1, j)\), and \((i+1, j+1)\). Sometimes we refer them 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., two strings \( A = a_1, a_2, \ldots, a_m \) and \( B = b_1, b_2, \ldots, b_n \), the grid DAG, \( G \), associated with strings \( A \) and \( B \) is an \((m+1) \times (n+1)\) grid DAG such that each edge on \( G \) is associated with cost 1 if it is a diagonal edge from vertex \((i, j)\) to vertex \((i+1, j+1)\) and symbols \( a_i \) and \( b_j \) are identical, otherwise associated with cost 0. The length of a path on \( G \) is defined as the sum of costs on the path. Throughout, we presume that \( m, n \) of \( A \), is a power of 2. As an example, Figure 1 shows the grid DAG associated with strings "tcaggati" and "galattagcagg".

By this definition, edge \( e = (v_k, v_{k+1}) \) is an edge on \( p = v_1, v_2, \ldots, v_l \) on \( G \), where the location of \( v_k \) is \((i_k, j_k)\), then, edge \( e \) has cost 1 if and only if symbol \( a_{i_k} \) in \( A \) is identical to symbol \( b_{j_k} \) in \( B \). Therefore, in general, we have the following fact.

**Observation 2.1:** Any path of length \( l \) on grid DAG \( G \) associated with strings \( A \) and \( B \) corresponds to a CS with length \( l \) of \( A \) and \( B \). In particular, the longest path between the
Figure 1: The grid DAG for string “teaggtt” and “gatttactcagg”

source and the sink corresponds to the LCS of A and B.

So, given strings A and B with lengths m and n, respectively, to find the LCS of A and B we only need to find the longest path beginning at the source and ending at the sink on grid DAG G associated with A and B. To do this, we are going to solve a more general problem: Find the longest paths of G from every vertex on the top row to every vertex on the bottom row. Consider the longest paths from a vertex on the top row of G, say vertex (1, i), to vertices on the bottom row. A vertex v on the bottom row is the j-th breakout vertex with respect to (w.r.t.) vertex (1, i) if v is the left most vertex on the bottom row such that the path from vertex (1, i) to v is of length j. Sometimes we simply call v the breakout vertex w.r.t. vertex (1, i), or the breakout vertex of vertex (1, i) for short. In Figure 1, vertices (9, 2), (9, 3), (9, 4), (9, 5) and (9, 13) are the 1-st, 2-nd, 3-rd, 4-th and 5-th breakout vertices w.r.t. the source. Note that there are no 5-th breakout vertices w.r.t. some vertices, for example, (1, 8), because the longest path from vertex (1, 8) to the bottom row is 4.

\[
D_{G_U} = \begin{pmatrix}
2 & 7 & 9 & 12 \\
3 & 7 & 9 & 12 \\
4 & 7 & 9 & 12 \\
5 & 7 & 9 & 12 \\
6 & 7 & 9 & 12 \\
7 & 9 & 11 & 12 \\
8 & 9 & 11 & 12 \\
9 & 11 & 12 & \infty \\
10 & 11 & 12 & \infty \\
11 & 12 & \infty & \infty \\
12 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty \\
\end{pmatrix}
\]

\[
D_{G_L} = \begin{pmatrix}
2 & 3 & 4 & 5 \\
3 & 4 & 5 & \infty \\
4 & 5 & \infty & \infty \\
5 & 6 & \infty & \infty \\
6 & 8 & \infty & \infty \\
7 & 8 & \infty & \infty \\
8 & 11 & \infty & \infty \\
9 & 11 & \infty & \infty \\
11 & \infty & \infty & \infty \\
12 & \infty & \infty & \infty \\
\infty & \infty & \infty & \infty \\
\end{pmatrix}
\]

Figure 2: $D_{G_U}$ and $D_{G_L}$

The $n \times m$ distance matrix associated with $G$, $D_G$, is defined as follows:

\[
D_G(i, j) = \begin{cases}
  k & \text{if vertex (m + 1, k) is the j-th breakout vertex w.r.t. vertex (1, i)} \\
  \infty & \text{if vertex (1, i) does not have the j-th breakout vertex,}
\end{cases}
\]

for $1 \leq i \leq n$ and $1 \leq j \leq m$.

Throughout, by $D_{G_i}$ we denote the i-th row of $D_G$.

Note the difference between our definition of distance matrix and the definition in [AP88] and [AALM90]. An entry in $D_G$, in our definition, is really not the length of the path but the location of the vertex on the bottom row. Figure 2 is the distance matrices associated with $G_U$, the upper half of $G$ shown in Figure 1, and $G_L$, the lower half of $G$.

2.2 The main structure of the algorithm

Now we give an overview of our algorithm. Basically the algorithm consists of the following four steps:

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