Parallel Algorithms for the Longest Common Subsequence Problem

K.Nandan Babu

Computer Science & Engineering
Indian Institute of Technology,
Kanpur, INDIA 208016

Sanjeev Saxena

Computer Science & Engineering
Indian Institute of Technology,
Kanpur, INDIA 208016

Abstract

The longest common subsequence problem is to find a substring that is common to two given strings and is atleast as long as any other such string. If m and n are the lengths of the two strings ($m \leq n$), we obtain $O(\log m)$ time parallel algorithm with mn processors and an $O(\log^2 n)$ time optimal parallel algorithm. Serial complexity on decision tree model is $\Theta(mn)$.

1 Introduction

Given a string, S, a *subsequence* S' of string S can be obtained by deleting some or no symbols of S; the symbols which are deleted need not be consecutive. If a string S_1 is a subsequence of both S_2 and S_3, then S_1 is said to be a *common subsequence* of S_2 and S_3. If string S_1 is at least as long as any other common subsequence of S_2 and S_3 then S_1 is called *longest common subsequence* (LCS) of S_2 and S_3. The longest common subsequences for two given strings may not be unique. Given two strings S_1 and S_2 of lengths m and n respectively, $m \leq n$, the LCS problem is to find a longest common subsequence of S_1 and S_2. Longest common subsequence problem finds applications in genetic engineering, data compression, editing, error correction, and syntactic pattern recognition[4, 7].

Aho et.al. [1] have obtained a lower bound of $\Omega(mn)$ on time for the LCS problem on the decision tree model (i.e, all decisions are of type whether two positions have the same symbol) if the number of distinct symbols that can appear in the strings is infinite.

LCS problem is related to the problem of finding the *maximum cost* path on a directed grid graph [6, 3]. Apostolico et.al. [3] have obtained an $O(\log n(\log\log m)^2)$ time algorithm with $O(mn/\log\log m)$ processors on CRCW model. Lu and Lin [6] obtain two algorithms: first is an $O(\log^2 m + \log n)$ time algorithm with $mn/\log m$ processors and other is an $O(\log^2 m \log\log m)$ time optimal algorithm with a processor-time product of $O(mn)$ on CREW model. In this paper we improve these algorithms on the stronger CRCW model. We describe an $O(\log m)$ time algorithm with mn processors and an $O(\log^2 n)$ time optimal parallel algorithm.

2 LCS Problem and Grid Directed Acyclic Graph

An $m \times n$ grid directed acyclic graph (DAG) is a graph with vertices as points of an $m \times n$ grid; the edges emanating from vertex (i, j) (for $i < m, j < n$) are to vertices $(i, j+1), (i+1, j)$ and $(i+1, j+1)$; vertices (n, j) will only have edge $(n, j+1)$ (for $j < m$) and vertices (i, m) can only have edge $(i+1, m)$. Vertex $(1, 1)$ is the source and vertex (m, n) is the sink. Costs are assigned to these edges as follows: if in the given strings A and B ith symbol in A and jth symbol in B match (are same) then the edge from (i, j) to $(i+1, j+1)$ is assigned a cost 1 and all other edges are assigned cost 0.

To solve the LCS problem, we need to find the maximum cost path beginning at the source and ending at the sink on the DAG G; we solve a more general problem of finding the maximum cost paths from every vertex on the top row to every vertex on the bottom row. We divide the $(m+1) \times (n+1)$ grid DAG, G into two $(m/2+1) \times (n+1)$ grid DAGs, G_u the upper half and G_l lower half; and recursively find the maximum cost paths on G_u and G_l and merge the paths.

A vertex v on the bottom row is defined as the *jth breakout vertex* with respect to vertex $(1, i)$ if there is a path of cost j from vertex $(1, i)$ to v and v is the leftmost of such vertices[6].

If a vertex v has jth breakout vertex w, then the maximum cost path from v on the top row to its jth breakout vertex w on the bottom row on G, must have cost j. This is because all unit cost (non-zero) edges appear on diagonal only. Indeed, if there is a path between vertex v and w with cost greater than j, then vertex w can not be a jth breakout vertex of v, because in that case, 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. We are interested only in the left most maximum cost path between any pair of vertices.

We store information about breakout vertices in a matrix called *cost matrix*. We know that the maximal possible length of the common subsequence is m, and hence maximal possible cost of a path of an $(m+1) \times (n+1)$ grid DAG is m. Therefore any vertex on the top row can have at most m breakout vertices.

Now a cost matrix D_G associated with grid DAG G, is defined as follows [6]: Each row of the matrix corresponds to a vertex on the top row of the DAG; and ith row jth column of the matrix stores the index of the jth breakout from the ith vertex on top row.
Definition: for $1 \leq i \leq n$ and $1 \leq j \leq m$, D_G(i,j)=k if vertex (m+1,k) is jth breakout from vertex $(1, i)$.

3 The main structure of Algorithm

High level description of algorithm is [6]:

1. Find a cost matrix for every two consecutive rows of D_G i.e., for a graph G_i 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 of G.
4. Identify the Longest Common Subsequence that corresponds to the maximum cost path found in Step 3.

We first describe implementation of Phase 1 which takes $O(1)$ time with mn processors in Section 3.1. Then in Section 3.2, we show that Phase 4 can be implemented in $O(\log n)$ time with $n/\log n$ processors, assuming implementation of other two phases. An $O(\log m)$ time implementation with mn processors of Phase 2 is described in Section 4 and an $O(\log^2 n)$ time implementation with $mn/\log^2 n$ processors is described in Section 5. Implementation of Phase 3 is in Section 3.3.

3.1 Implementation of Phase 1

We use divide-and-conquer strategy for solving the LCS problem. Given two strings P and Q, to compute cost matrix D_G of G associated with P and Q, as a basis for merge, we first need to compute m cost matrices, each being associated with a $2 \times (n+1)$ grid DAG.

Any vertex on the top row of G_h can have at most one breakout vertex. Let us consider computing cost matrix for two consecutive rows for G_h, the DAG in this case consists of hth and $(h+1)$th rows of G. Let q_i denote the ith symbol of Q and P_j the jth symbol of P. Let us assume that symbols $q_{j1}, q_{j2}, \ldots, q_{jr}$ are all identical to p_h for $j_1 < j_2 < \ldots < j_r$. A cost matrix P_{G_h} can now be constructed as follows:

1. Initialize $D_{G_h}(1,i) = 0$, for $1 \leq i \leq r$; Let $D_{G_h}(1,1) = j_1 + 1$ and let $D_{G_h}(j_{k-1}+1, 1) = j_k$ for $1 \leq k \leq r$.
2. Compute prefix maxima from $D_{G_h}(1,1)$ to $D_{G_h}(j_r+1, 1)$.
3. Assign ∞ to entries of D_{G_h} from $D_{G_h}(j_r+2, 1)$ to $D_{G_h}(n, 1)$.

By assigning a processor to each of the vertex on the top row of G_h, we can identify $j_1, j_2, \ldots j_r$ in constant time. For this step we need n processors. Since there are m such matrices, with mn processors, this step can be implemented in $O(1)$ time

3.2 Implementation of Phase 4

Let $\mathcal{P} = v_1, v_2, \ldots, v_n$ be the maximum cost path obtained in Phase 3. The symbol p_i in $\mathcal{P}$ has to be marked if edge $e = (v_k, v_{k+1})$ has cost 1 and vertex v_k has column index i. Now getting the marked symbols together gives LCS. So, LCS can be obtained by *ranking* the marked symbols. Since the maximum cost path can have atmost $n+m$, edges marking can be done in constant time with $m+n$ processors. The ranking can be done in $O(\log n)$ time with $n/\log n$ processors using a standard technique [5].

3.3 Implementation of Phase 3

We identify vertices on the maximum cost path $\mathcal{P}$ between the source and sink of the $(m+1) \times (n+1)$ grid DAG given D_G. In the maximum cost path there can be more than one vertex in the same row, of G as $m \leq n$. A vertex v_i on $\mathcal{P}$ is a *cross vertex* if v_i is the leftmost vertex on that row. Phase 3 is implemented in two stages (as in [6]). First Identify cross vertices on $\mathcal{P}$ and then identify other vertices on $\mathcal{P}$. Lu and Lin [6] have shown that Phase 3 can be implemented in $O(\log n)$ time with n processors.

4 Implementation of Phase 2

In this phase we recursively calculate D_G from D_{G_u}, the matrix corresponding to the upper half DAG and D_{G_l} the matrix corresponding to lower half DAG. Consider vertex $(1, i)$ and its jth breakout vertex, say, $(m+1, i_v)$, and the maximum cost path, say $\mathcal{P}$, between them. Clearly $D_G(i,j) = i_v$. Now let us assume that $\mathcal{P}$ intersect common boundary of D_{G_u} and D_{G_l} at $w = (\frac{m}{2}+1, i_q)$ (say). Thus vertex w divides the maximum cost path $\mathcal{P}$ into two sub-paths P_1 and P_2, where sub-path P_1 is from vertex $(1, i)$ to w and sub-path P_2 is from vertex w to $(m+1, i_v)$. Let us assume that w is the kth breakout vertex of $(1, i)$ on D_{G_u} (for some k and i). Therefore $(m+1, i_v)$ will be $(k-j)$th breakout vertex of $(1, i_q)$ on D_{G_l}. 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 $k \neq j$. As $D_G(i,j) = i_v$, we get $D_G(i,j) = D_{G_l}(D_{G_u}(i,k), k-j)$.

As we are interested only in left most paths, when $k = 0$, sub-path P_1 must go straight down from $(1, i)$ to w; thus $i_q = i$. Consequently, sub-path P_2, having cost j, must be the maximum cost path from $(m/2 + 1, i)$ to $(m + 1, i_v)$, or $D_G(i, j) = i_v$. Hence $D_G(i, j) = D_{G_l}(i, j)$. Similarly if $k = j$ we get $D_G(i, j) = D_{G_u}(i, j)$.

Theorem 1 **[6]** *For* $1 \leq i \leq n$ *and* $1 \leq j \leq m$, $D_G(i, j) = \min_{1 \leq k \leq j} (D_{G_u}(i, j), D_{G_l}(i, j), D_{G_l}(D_{G_u}(i, k), j - k))$ *where both* $D_{G_u}(i, j)$ *and* $D_{G_l}(i, j)$ *are* ∞ *for* $j > \frac{m}{2}$ *and* $D_{G_l}(D_{G_u}(i, k), j - k)$ *is also* ∞ *for* $D_{G_u}(i, k) = \infty$. ∎

By Theorem 1, for the leftmost path we have to search for a vertex with minimum value that is at a distance j from $(1, i)$. For this we define a new matrix, $M[D^i_G]$ (refer [6]) a matrix corresponding to ith row of D_G; we have n such matrices ($1 \leq i \leq n$). The size of $M[D^i_G]$ is $l \times m$, where l is the number of breakout vertices of vertex $(1, i)$ on G_u. For $1 \leq j \leq l$ the jth row of $M[D^i_G]$ is defined as follows:

Definition: $M[D^i_G](j, k) = D_{G_l}(D_{G_u}(i, j), k - j)$, for $j + 1 \leq k \leq \frac{m}{2} + j + 1$ and the remaining $m/2$ entries of $M[D^i_G]$ are ∞.

Clearly from definition of $M[D^i_G]$, Theorem 1 can be rewritten as follows.

Corollary 1 **[6]** $D_G(i, j) = \min_{1 \leq k \leq l} \left(D_{G_u}(i, j), D_{G_l}(i, j), M[D^i_G](k, j)\right)$ *for* $1 \leq i \leq n$ *and* $1 \leq j \leq m$. *Here* l *is the number of the breakout vertices of vertex* $(1, i)$ *on* G_u. ∎

4.1 K-variant property of D_G and $M[D^i_G]$

We next illustrate some properties of D_G and $M[D^i_G]$ that are useful in the algorithm.

Given a row vector, R, a *subrow vector* R' of R can be obtained by deleting some or no entries of R; the entries which are deleted need not be consecutive. If a row vector is a subrow vector of more than two row vectors, say, $W_1, W_2, ...W_l$ then it is a *common row vector* of them.

A pair of row vectors of size m is said to be *k-variant* either the rows differ in atmost k elements; in other words their *common row vector* is of size atleast $m - k$. Two matrices of size $m \times n$ are said to be *k-variant* if there exists a sub-matrix of size $(k - m) \times n$ which is common to both.

Proposition 1 **[6]** *If value* k *is found in a row of cost matrix* D_G, *then it will also be found in next* $(k - 1)$ *rows; i.e., 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 *between* i_1 *and* k $(i_1 \leq i_2 \leq k)$ ∎

Theorem 2 **[6]** *If* $D^{i_1}_{G_u}$ *and* $D^{i_2}_{G_u}$ *are k-variant, then* $M[D^{i_1}_G]$ *and* $M[D^{i_2}_G]$ *are also k-variant.* ∎

Proofs of the Proposition 1 and Theorem 2 can be found in [6]. From the Proposition 1 it is clear that, any $k + 1$ rows of D_G are $k - variant$.

Now, let $L = (L_1, L_2, ..., L_r)$ be a common row-vector of $D^{i_1}_{G_u}$ and $D^{i_2}_{G_u}$, where L_j is the jth group of L. Then a common matrix $M = (M_1, M_2, ..., M_r)^T$ of $M[D^{i_1}_G]$ and $M[D^{i_2}_G]$ can be constructed from the corresponding common row vectors as follows: for every element of a row of D_{G_u}, there exists a row in the corresponding $M[D^i_G]$(see definition). Since L contains common elements of the two rows $D^{i_1}_{G_u}$ and $D^{i_2}_{G_u}$, the matrix $M[D^i_G]$ constructed from L has rows common to both $M[D^{i_1}_G]$ and $M[D^{i_2}_G]$. Recall $D^{i_1}_{G_u}$ and $D^{i_2}_{G_u}$ are $i_1 th$ and i_2th rows of D_{G_u} respectively. Thus a matrix $M = (M_1, M_2, ..., M_r)^T$, constructed from $L = (L_1, L_2, ..., L_r)$ using definition is a common matrix to both $M[D^{i_1}_G]$ and $M[D^{i_2}_G]$. Here all M_is have same number of columns as $M[D^i_G]$ and each M_i has some consecutive rows of $M[D^i_G]$.

4.2 Totally monotone property of D_G and $M[D^i_G]$

If in a matrix Z, minimum value of items stored in the ith column is in jth row, then we define $\zeta_Z(i) = j$. The matrix Z is said to be *monotone* if $\zeta_Z(i) \leq \zeta_Z(i + 1)$. Matrix Z is said to be *totally monotone* if every 2×2 sub-matrix of Z is also monotone. Lu and Lin [6] have shown that the matrix $M[D^i_G]$ is totally monotone. We can compute the cost matrix D_G by assuming that the matrix D_G has been partitioned into $\frac{n}{k}$ sub-matrices each having k consecutive subrows of D_G. We compute these sub-matrices independently. Let us concentrate on the computation of a submatrix, which consists of the first k rows of D_G. Remaining submatrices can be computed similarly. The entry $D_G(i, j)$ is minimum of a) $D_{G_u}(i, j)$, b) $D_{G_l}(i, j)$ and c) jth column minima of $M[D^i_G]$. As only the computation of column minima of $M[D^i_G]$ is non-trivial, we concentrate on computing column minima. Recall that any k consecutive rows of D_{G_u} are $k - 1$ variant, and hence $M[D_{G1}], M[D_{G2}], \ldots, M[D_{Gk}]$ are also $k-1$ variant. Hence each of these can be represented by a common matrix together with at most $k-1$ other row vectors. Let $M = (M_1, M_2, ..., M_r)^T$ be the common matrix of $M[D^i_G]$.

Let us now consider the problem of finding column minima. We are given D_{G_u} and D_{G_l} and we have to find $Colmin[M[D^i_G]]$. We avoid the generation of the matrix $M[D^i_G]$ since it is costly. We proceed in following steps.

Step 1 Find common row vector $L = (L_1, ..., L_r)$ of

$D^i_{G_u}$.
Consider any two rows R_{i_1} and R_{i_2}, $i_1 < i_2$, then all the rows R_i $i_1 \leq i \leq i_2$ are $i_2 - i_1 - 1$ variant. If we find the common row vector of R_{i1} and R_{i2}, it is also the common row vector of the rows R_i for $i_1 \leq i \leq i_2$. So for finding the common row vector of the first k rows of D_{G_u} it is enough to find the common row of the first and the kth row. *The numbers present in the matrix are less than or equal to* n, the size of the longer string. This follows from the definition of the cost matrix.

Now proceed as follows: Choose m/k rows at intervals of k. Assign n processors to each selected row. Find the common row vector of $(2i+1)k$th and $(2i+2)k$th rows, (for $i = 0, ..., (m/2k)-1$), in constant time. This is done using an array of size n. Initialization can be avoided by using the method of back pointers (see Ex 2.12 of [2]). Since the elements are in the range 1 to n and we have n processors it is easy to find the common row vector in constant time. Now, we have a common row vector for each of the k rows. It remains to find the remnants (i.e., the elements of the row other than those in common row vector) of each row among the k rows. Assign one processor to each of the elements of the k rows and obtain the remnants in constant time. Thus we need mn processors to get m/k common row vectors and m corresponding remnants in constant time.

Let $L = (L_1, L_2, ...L_r)$ be a common row vector of row vectors $D^i_{G_u}$ 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 $D^i_{G_u}$, denoted by $Pos[D^i_{G_u}, L_j]$, is defined as the index of the entry in $D^i_{G_u}$, which is identical to $L_j(1)$. Now a Position function $P[D^i_{G_u}, L]$ is defined such that $P[D^i_{G_u}, L](j) = Pos[D^i_{G_u}, L_j]$.

We should also compute position function $P[D^i_{G_u}, L]$ $i = 1, \ldots, k$. This can be done by identifying the position of $L_j(1)$s in D^i_{Gu}. As a matter of fact, we can do this while computing L_j itself.

Step 2 Find common matrix M of $M[D^i_G]$.
We use the definition to calculate M_js of M; position function is used to get positions of M_js in $M[D^i_G]$s in $O(1)$ time.

The position of M_js in the matrix $M[D^i_G]$s, denoted by $Pos[M[D^i_G, M]]$, is defined as the index of the entry in $M[D^i_G]$, which is the left most element of the matrix that is similar to M_j. Now a Position function $P[M[D^i_G], M]$ is defined such that $P[M[D^i_G], M](j) = Pos[M[D^i_G], M_j]$. We should calculate a position function $P[M[D^i_G, M]$ for $i = 1, \ldots, k$. We have $P[M[D^i_G, M]] = P[D^i_{G_u}, L]$ (for proof see [6]). The function $P[D^i_{G_u}, L]$ is calculated in the previous step.

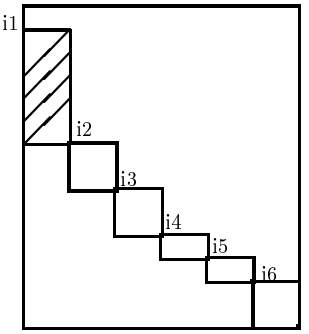

Figure 1: Monotone Matrix corresponding to the problem

Step 3 Compute column minima of $M[D^i_G]$
Since M_js are submatrices of $M[D^i_G]$, we first find column minima of M_js. We have seen that M_js are monotone arrays. We use the procedure described in the next subsection to find their column minima. M_j have same number of columns as $M[D^i_G]$ but have fewer rows. Replace the matrix M_j with its column minima in $M[D^i_G]$s. Observe that the resulting $M[D^i_G]$s are still monotone and are much smaller than original $M[D^i_G]$. Finally, we have to find the column minima among them again. The procedure for searching monotone arrays is next described.

4.3 Searching Monotone Arrays for Column Minima

Consider a monotone matrix of size $m \times n$ whose elements are integers less than or equal to n. We basically reduce problem space by selecting k columns at equal intervals in the matrix and find minimums among them using "small universe algorithm". this step requires mk processors and can be done in constant time.

Let the minimums in the selected columns be in the rows i_1, i_2, etc,. Consider the unselected columns between the first two selected columns. The minimums among them are bound to be found in the rectangle shown in Figure 1. That is so because as column minima in the first selected column is in i_1th row, the column minima of the columns after it should be found below or to the right of it, and since the column minima of the second selected column is in i_2th row, the column minima of the columns preceding it should be found to the left of it or above it. Hence it follows that the search space for column minima of the columns between the first two selected columns is only the rectangle shown.

It remains to find the column minima in the remaining columns in the rectangles shown. The rectangles can be placed one below the other resulting in n/k columns and m rows. Applying the "small universe algorithm" again to find the column minima in constant time we need $n/k \times m$ processors. Choose $k = \sqrt{n}$; Then this step requires $\sqrt{n}m$ operations.

Thus this stage takes atmost mn operations and can be done in constant time. Since there are $\log m$ merge stages in phase 2 this phase can be completed with $O(mn \log m)$ operations with $O(\log m)$ time.

4.4 Complexity of algorithm

The algorithm consists of four phases. In Section 3, we saw that the first phase is executable in constant time with mn operations and the fourth phase can be implemented in $O(\log n)$ time and n operations. Further $O(\log n)$ time and n processors suffice for Phase 3. As, second phase can be implemented in $O(\log m)$ time with $O(mn \log m)$ operations, the entire algorithm takes $O(\log m)$ time with mn processors.

5 An Optimal Algorithm for LCS Problem

The main difference between the algorithm of previous sections and this one is the data structure used for representing the cost matrices. In this algorithm, Common vectors and Remnants are used to represent cost matrix D_G. For every $k+1$ consecutive rows of the cost matrix a row vector is used to represent the common row and $k+1$ arrays are used to represent $k+1$ remnants one each for every row. Let remnant for ith row be denoted by $R[D^i_G]$. Further position functions defined in the previous algorithm are used to facilitate "random access" of any entry of this data structure. As any $k+1$ consecutive rows of D_G with size $n \times m$, can be represented by a common row vector of size atmost m, and $k+1$ remnants, each with size of atmost k, hence $O(mn/k + nk)$ space suffices to store all information of D_G[6].

Observation 1 *Let L_h be the hth group of L, the common row-vector of D^i_Gs for $i_1 \le i \le i_1 + k$. Then the position of the first element of the L_h in pth row is less than its position in qth row, if $i + k > p > q > i$. i.e., $Pos[D^{i_1+k}_G, L_h] \le Pos[D^i_G, L_h] \le Pos[D^{i_1}_G, L_h]$.* ∎

The above observation follows from the definition of cost matrix. To be precise $Pos[D^{i_1+k}_G, L_h] \le Pos[D^i_G, L_h] - k$ because, an element found in a row can shift to left in the next row by atmost one position (it cannot shift to right).

Let us now discuss algorithm for computing D_G. Similar to our earlier algorithm, we partition D_G in to n/k submatrices, each having k consecutive rows of D_G. We explain the computation of one such matrix; other submatrices can be obtained similarly. The computation of the sub-matrices is in three steps as follows [6]:

Step 1 Compute common row vector L.
This is done by first Computing D^1_G and D^k_G using our earlier algorithm. The optimal algorithm of Lu and Lin [6] uses an algorithm which runs in $O(\log^2 m + \log n)$ time, for the above step. We use our fast algorithm (Section 4.3) which runs in $O(\log m + \log n)$ time. Then compute common row-vector L of D^1_G, D^k_G and the corresponding position functions $P[D^1_G, L] and P[D^k_G, L]$ using Step 1 of algorithm of Section 4.2.

Step 2 Compute remnants $R[D^i_G]$ $(i = 1, \ldots, k)$.

Step 3 Compute Position functions $P[D^i_G, L], P[D^i_G, R[D^i_G]]$, for $(i = 1, \cdots k)$.

Let us discuss implementation of Steps 2 and 3 in detail. In the Step 2 we have to generate the remnants. Let R^a_b be the bth remnant group of ath row. We shall discuss how to compute the remnant group R^a_b. Other remnant groups can be handled similarly.

Observation 2 *Any finite entry in R^a_b is also contained in $R^k_b (k > a)$ and infinite (∞) entries are always on the right side of finite entries in D^a_G.* ∎

Remembering that R^a_b is a subrow of D^a_G, the above observation is straight forward. As we have already computed D^1_G and D^k_G computation of $R[D^1_G]$ and $R[D^k_G]$ is easy, since we have the common row vector from step 1. Other remnants are computed from them as will be explained.

Proposition 2 *Let $R[D^i_G] = (R^i_1, \ldots, R^i_{r+1})$ be the remnant of D^i_G, then $R^{i_1}_j$ is a subvector of $R^{i_2}_j$ when $i_1 < i_2$ and $2 \le j \le r+1$.* ∎

From Proposition 2, when $b \ge 2$, R^a_b is a subvector of R^k_b. We also know that D^a_G in monotonally increasing (from definition of cost matrix). Hence R^a_b is the largest common subvector of D^a_G and R^k_b. Therefore if D^a_G is given R^a_b can be obtained. But as D^a_G is not known, we will find a sub-row of D^a_G which is sufficient for our purpose. Let us call this sub-row $SD[a, b]$, which is defined as follows.

Definition: $SD[a, b]$ contains all elements of R^a_b
The size of $SD[a, b]$ is bounded by $O(k)$.

The basic formula described in Corollary 1 of Theorem 1 is applied to generate $SD[a, b]$, as $SD[a, b]$ is a sub-row of D^a_G. We next try to identify columns of $M[D^i_G]$ which should be selected to find column minima in order to get entries of $SD[a, b]$. Let $l = Pos[D^k_G, R^k_b]$ be the position of the first element of bth remnant of kth row (in kth row). Then the po-

sition of first element of bth remnant of ath row in D^a_G can be atmost $l+k$, where k is the number of rows we are considering; this is because if an element is found in ath row then it can shift to left in the next row by atmost 1 (it cannot shift to right). Thus, $l \leq Pos[D^a_G, R^a_b] \leq l+k$.

Now since there are atmost k entries in R^a_b we choose the subrow of D^a_G from $D_G(a,l)$ to $D_G(a, l+2k-1)$ as SD[a,b]. By above inequality, R^a_b is contained in $SD[a,b]$.

Let array Ind store positions in D^a_G for entries in $SD[a,b]$ [6]; Then, $Ind[SD[a,b]](i) = Pos[D^k_G, R^k_b] + i - 1$, for $i = 1, \ldots, 2k$. i.e., $Ind[SD(a,b)]](i) = l$ if $SD[a,b](i) = D^a_G(l)$.

Therefore we need to identify only the $IND[SD[a,b]](i)$th column minima in $M[D^a_G]$ for $i = 1, \ldots, 2k$. Now let $M[SD[a,b]]$ be a sub-matrix of $M[D^a_G]$ obtained by deleting those columns that are not in $SD[a,b]$. Now proceed as follows:

- Identify a common matrix of $M[SD[a,b]]$, for $1 \leq a \leq k$.
- Find the column minima of the common matrix.
- Replace the common matrix in all $M[SD[a,b]]$ by the column minima and find the column minima of $M[SD[a,b]]$ to get $SD[a,b]$.

We have already seen how to identify M_j, the common matrix of $M[D^i_G]$ in our earlier algorithm. Here we need not compute the whole matrix M_j, which is costly. For obtaining the common matrix of $M[SD[a,b]]$, it is enough to identify those columns from M_js, which are required to get the common matrix of $M[SD[a,b]]$. This is because, the common matrix of $M[SD[a,b]]$ is a submatrix of M_j with less columns. Let us call this common matrix of $M[SD[a,b]]$, SM_k. We use array IND for finding SM_k from M_k. Recall that array IND stores positions in D^a_G for entries in $SD[a,b]$. Correspondingly we can get the positions of SM_k in M_k from it.

Now, since M_ks are totally monotone so are SM_ks. To find the column minima of the common matrix, we use the procedure described in our previous algorithm to "search for column minima in monotone arrays". This step takes mn operations and can be done in constant time. Alternately we can do in $O(\log m)$ time with $mn/\log m$ processors.

For Step 3 of the above algorithm, recall that the common matrix of SM_k contains same number of columns as $M[SD[a,b]]$. Replace all the rows of SM_k in $M[SD[a,b]]$ with a single row containing the column minima of SM_k. Then the resulting matrix is smaller and is still monotone, because $M[SD[a,b]]$ is totally monotone. Now find the column minima in this resulting matrix as explained above with same time and processor limits.

Once column minima is obtained we get $SD[a,b]$ and using it R^a_b. (see definition). Now it remains to find $P[D^i_G, R[D^i_G]]$, to complete the data structure for D_G. We have shown how to identify $P[D^1_G, L]$ in earlier sections. We first identify $P[D^1_G, L]$ and from it $P[D^i_G, R[D^i_G]]$s can easily be identified as the number of groups of L and $R[D^i_G]]$ are bounded by a polylogarithm. (for proof see [6]). Phase 3 in this algorithm is to identify the maximum cost path. This is done in $O(\log m)$ time using n processors as in [6].

5.1 Analysis of Algorithm

Phase 1 of this algorithm can be done in $O(\log n)$ time with $mn/\log n$ processors, as is [6]. Phase 3 and Phase 4 are also the same Phase 2, the crucial part of the algorithm takes $O(\log^2 m)$ time with $mn/\log^2 m$ processors. In fact, the analysis given in [6] works fine for us except for the fact that each stage in the Phase 2 of that algorithm takes $O(\log m \log\log m)$, where as for us it is only $O(\log m)$. So Phase 2 of our algorithm takes $O(\log^2 m)$ as compared to $O(\log^2 m \log\log m)$ time in [6].

Acknowledgments

We thank a referee for bringing [3] to our notice.

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.V.Aho, J.E.Hopcroft and J.D.Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[3] A.Apostolico, M.J.Atallah, L.L.Larmore and S.McFaddin, *Efficient Parallel Algorithms for String Editing and Related Problems*, SIAM J. Comput, vol 19, no.5, pp 968-988, October 1990

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

[5] A. Gibbons and W. Rytter, *Efficient parallel Algorithms*, Cambridge, U.K. Cambridge University Press, 1988.

[6] Mi Lu and Hua Lin *"Parallel Algorithms for the Longest common Sub sequence problem",* IEEE tran. on Parallel and Distri. Sys. vol 5. No 8. August 1994. pp 835-847.

[7] J. Modelevsky, *"Computer applications in applied genetic Engineering"*, Advances in applied Microbiology, vol 30, pp. 169-195, 1984.