A BSP/CGM Algorithm for the All-Substrings Longest Common Subsequence Problem

C. E. R. Alves [*]
Univ. São Judas Tadeu
São Paulo, Brazil
prof.carlos_r_alves@usjt.br

E. N. Cáceres[†]
Univ. Fed. Mato Grosso do Sul
Campo Grande, Brazil
edson@dct.ufms.br

S. W. Song [‡]
Universidade de São Paulo
São Paulo, Brazil
song@ime.usp.br

Abstract

Given two strings X and Y of lengths m and n, respectively, the all-substrings longest common subsequence (ALCS) problem obtains the lengths of the subsequences common to X and any substring of Y. The sequential algorithm takes $O(mn)$ time and $O(n)$ space. We present a parallel algorithm for ALCS on a coarse-grained multicomputer (BSP/CGM) model with $p < \sqrt{m}$ processors that takes $O(mn/p)$ time and $O(n\sqrt{m})$ space per processor, with $O(\log p)$ communication rounds. The proposed parallel algorithm also solves the well-known LCS problem. To our knowledge this is the best BSP/CGM algorithm for the ALCS problem in the literature.

1. Introduction

Given two strings, obtention of the longest subsequence common to both strings is an important problem with applications in DNA sequence comparison, data compression, pattern matching, etc. In this paper we consider the more general all-substring longest common subsequence problem and present a time and space efficient parallel algorithm.

Consider a string of symbols from a finite alphabet. A *substring* of a string is any contiguous fragment of the given string. A *subsequence* of a string is obtained by deleting zero or more symbols from the original string. A subsequence can thus have noncontiguous symbols of a string. Given the string *lewiscarroll*, an example of a substring is *scar* and an example of a subsequence is *scroll*. Given two strings X and Y, the *longest common subsequence* (LCS) problem finds the length of the longest subsequence that is common to both strings. If X = *twasbrillig* and Y = *lewiscarroll*, the length of the longest common subsequence is 5 (e.g. *warll*).

The *all-substring longest common subsequence* (ALCS) problem finds the lengths of the longest common subsequences between X and *any* substring of Y. Given strings X and Y of lengths m and n, respectively, we present a parallel algorithm for ALCS on a coarse-grained multicomputer (BSP/CGM) with p processors. The LCS and ALCS problems can be solved through a grid directed acyclic graph (GDAG). The proposed algorithm finds the lengths of the best paths between all pairs of vertices with the first vertex on the upper row of the GDAG and the second vertex on the lower row. On a BSP/CGM with $p < \sqrt{m}$ processors, the proposed parallel algorithm takes $O(mn/p)$ time and $O(n\sqrt{m})$ space per processor, with $O(\log p)$ communication rounds. To our knowledge this is the best BSP/CGM algorithm for this problem in the literature.

Solving the ALCS problem we obviously solve also the less general LCS problem. However, even considering the more general problem, we managed to obtain a time complexity of $O(mn/p)$, giving linear speedup over the usual algorithms for the LCS problem. We explore the properties of totally monotone matrices and the similarity between rows of the D_G matrix as well as between consecutive $MD[i]$ matrices. Thus the amount of information to be computed is reduced through the elimination of redundancy. Another concern of importance is the effort to use compact data structures to store the necessary information and to reduce the size of messages to be communicated among processors.

Sequential algorithms for the LCS problem are surveyed in [4, 8]. PRAM algorithms for LCS and ALCS are presented in [7]. The ALCS problem can be solved on a PRAM [7] in $O(\log n)$ time with $mn/\log n$ processors, when $\log^2 m \log\log m \leq \log n$.

[*] Doctorate student at Universidade de São Paulo

[†] Partially supported by CNPq Proc. No. 52.2028/02-9 and FINEP-PRONEX-SAI Proc. No. 76.97.1022.00. Visiting Professor at the Universidade de São Paulo

[‡] Partially supported by FAPESP Proc. No. 99/07390-0, CNPq Proc. No. 52.3778/96-1, 46.1230/00-3, 521097/01-0 and 52.2028/02-9.

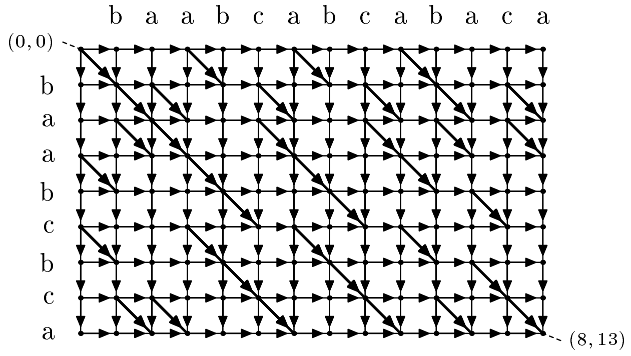

Figure 1. GDAG for the ALCS problem, with $X =$ baabcbca and $Y =$ baabcabcabaca.

2 The BSP/CGM Model

In this paper we use the *Coarse Grained Multicomputer* (BSP/CGM) [5, 6, 10] model. A BSP/CGM consists of a set of p processors $P_1, \ldots, P_p$ with $O(N/p)$ local memory per processor, where N is the space needed by the sequential algorithm. Each processor is connected by a router that can send messages in a point-to-point fashion. A BSP/CGM algorithm consists of alternating local computation and global communication rounds separated by a barrier synchronization. In the BSP/CGM model, the communication cost is modeled by the number of communication rounds. The main advantage of BSP/CGM algorithms is that they map very well to standard parallel hardware, in particular Beowulf type processor clusters [5]. Our goal is to minimize the number of communication rounds and achieve a good speedup.

3 The Grid Directed Acyclic Graph (GDAG)

As in the string editing problem [2, 9], the all-substrings longest common subsequence (ALCS) problem can be modeled by a *grid directed acyclic graph* (GDAG). Consider two strings X and Y of lengths m and n, respectively. To illustrate the main ideas of this paper, we use the following example. Let $X =$ baabcbca and $Y =$ baabcabcabaca. The corresponding GDAG has $(m+1) \times (n+1)$ vertices (see Figure 1). We number the rows and columns starting from 0. All the vertical and horizontal edges have weight 0. The edge from vertex $(i-1, j-1)$ to vertex (i, j) has weight 1 if $x_i = y_j$. If $x_i \neq y_j$, this edge has weight 0 and can be ignored.

The vertices of the *top row* of G will be denoted by $T_G(i)$, and those of the *bottom row* of G by $B_G(i)$, $0 \leq i \leq n$. Given a string Y of length n with symbols y_1 to y_n, denote by Y_i^j the substring of Y consisting of symbols y_i to y_j.

We now define the *cost* or *total weight* of a path between two vertices.

Definition 1 (Matrix C_G) *For $0 \leq i \leq j \leq n$, $C_G(i,j)$ is the cost or total weight of the best path between vertices $T_G(i)$ and $B_G(j)$, representing the length of the longest common subsequence between X and the substring Y_{i+1}^j. If $i \geq j$ (Y_{i+1}^j is empty or nonexistent), $C_G(i,j) = 0$.*

$\mathrm{C_G}$	0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	1	2	3	4	5	6	6	7	8	8	8	8	8
1	0	0	1	2	3	4	5	5	6	7	7	7	7	7
2	0	0	0	1	2	3	4	4	5	6	6	6	6	7
3	0	0	0	0	1	2	3	3	4	5	5	6	6	7
4	0	0	0	0	0	1	2	2	3	4	4	5	5	6
5	0	0	0	0	0	0	1	2	3	4	4	5	5	6
6	0	0	0	0	0	0	0	1	2	3	3	4	4	5
7	0	0	0	0	0	0	0	0	1	2	2	3	3	4
8	0	0	0	0	0	0	0	0	0	1	2	3	3	4
9	0	0	0	0	0	0	0	0	0	0	1	2	3	4
10	0	0	0	0	0	0	0	0	0	0	0	1	2	3
11	0	0	0	0	0	0	0	0	0	0	0	0	1	2
12	0	0	0	0	0	0	0	0	0	0	0	0	0	1
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 2. $\mathrm{C_G}$ of the given GDAG.

Values of $C_G(i,j)$ are shown in Figure 2. For example, $C_G(0,9) = 8$. This means the length of the longest common subsequence between $X =$ baabcbca and Y_1^9= baabcabca is 8. However, note that $C_G(0,10)$ is also 8. That is, if we take one more symbol of Y, the length of the longest common subsequence is still the same. This leads to the next definition of D_G that deals with this leftmost position (in the example, 9 and not 10) to achieve a fixed length value (in the example 8).

The values of $C_G(i,j)$ have the following property. For a fixed i, the values of $C_G(i,j)$ with $0 \leq j \leq n$ form a nondecreasing sequence that can be given implicitly by only those values of j for which $C_G(i,j) > C_G(i,j-1)$. This fact has been used in several sequential algorithms for LCS [8] and in the PRAM algorithm presented in[7] which is the basis for our algorithm and for the following definition.

Definition 2 (Matrix D_G) *Consider G the GDAG for the ALCS problem for the strings X and Y. For $0 \leq i \leq n$, $D_G(i,0) = i$ and for $1 \leq k \leq m$, $D_G(i,k)$ indicates the value of j such that $C_G(i,j) = k$ and $C_G(i,j-1) = k-1$. If there is no such a value, then $D_G(i,k) = \infty$.*

Implicit in this definition is the fact that $C(i,j) \leq m$. For convenience, we define D_G as a matrix with indices starting from 0. We denote by D_G^i the row i of D_G, that is, the row vector formed by $D_G(i,0)$, $D_G(i,1)$, …, $D_G(i,m)$. As an example, we again consider the GDAG of Figure 1. The values of $D_G(i,k)$ are shown in Figure 3.

The algorithm we propose deals directly with this representation. To understand the D_G matrix, consider $D_G(i,k)$.

$\mathbf{D_G(i,j)}$	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	8	9
1	1	2	3	4	5	6	8	9	∞
2	2	3	4	5	6	8	9	13	∞
3	3	4	5	6	8	9	11	13	∞
4	4	5	6	8	9	11	13	∞	∞
5	5	6	7	8	9	11	13	∞	∞
6	6	7	8	9	11	13	∞	∞	∞
7	7	8	9	11	13	∞	∞	∞	∞
8	8	9	10	11	13	∞	∞	∞	∞
9	9	10	11	12	13	∞	∞	∞	∞
10	10	11	12	13	∞	∞	∞	∞	∞
11	11	12	13	∞	∞	∞	∞	∞	∞
12	12	13	∞	∞	∞	∞	∞	∞	∞
13	13	∞	∞	∞	∞	∞	∞	∞	∞

Figure 3. $\mathbf{D_G}$ of the given GDAG.

Index i is the starting index of the Y string at the top row of G. The value k is the desired length of the common subsequence between X and the string Y starting at i. Consider the GDAG of Figure 1. If we start from position i of the top row and proceed to the bottom row at the position given by $D_G(i,k)$ then we can get a path of total weight k. Actually $D_G(i,k)$ gives the leftmost position that gives the total weight k. Let us illustrate this with an example. Since the length of string X is 8, the maximum value we can expect for k is therefore 8. Let us consider $D_G(0,8) = 9$. This means the following: in the GDAG of Figure 1, start from the index 0 of the top row and take edges at either of the three directions: by taking the diagonal we get a weight 1 while by taking the horizontal or vertical edges we get weight 0. Now if we wish to have a total weight of 8, then the leftmost position at the bottom row will be 9. Thus we have $D_G(0,8) = 9$. If we make i greater than 0 then we compare X with the string Y starting at position i.

The following property was proven in [7] and is important to our results. This property suggests the definition of V_G.

Property 1 *For $0 \le i \le n-1$, row D_G^{i+1} can be obtained from row D_G^i by removing the first element ($D_G^i(0) = i$) and inserting just one new element (that can be ∞).*

Definition 3 (Vector V_G) *For $1 \le i \le n$, $V_G(i)$ is the value of the finite element that is present in row D_G^i but not present in row D_G^{i-1}. If such a finite element does not exist, then $V_G(i) = \infty$.*

For example, V_G for the GDAG of Figure 1 is

$$(\infty, 13, 11, \infty, 7, \infty, \infty, 10, 12, \infty, \infty, \infty, \infty).$$

So we have an economical way of storing and communicating D_G with $O(m+n)$ space. We need only to store and transmit the first row of D_G, i.e. D_G^0, of size $O(m)$ and a vector V_G of size $O(n)$.

Due to space limitation, we state the following result without proof. Details can be obtained in [3].

Theorem 1 *Given two strings X and Y of lengths m and n, respectively, it is possible to solve the ALCS problem sequentially in $O(mn)$ time and $O(n)$ space.*

Using a result by Schmidt[9] for *all highest scoring paths* in GDAGs with unit weights we can solve the ALCS problem with the above complexity. The sequential algorithm for ALCS is important since it will be used in each processor, as seen in the following. The time complexity for this algorithm is equal to the complexity of the LCS when solved by the classic dynamic programming algorithm, except for a small multiplicative constant.

4 Basic Strategy of the BSP/CGM Algorithm

We will now present a BSP/CGM algorithm for the ALCS problem of two given strings X and Y of lengths m and n, respectively. For simplicity, we consider the number of processors p to be a power of 2 and m to be a multiple of p.

The algorithm divides string X into p substrings of length $\frac{m}{p}$ that do not overlap. The GDAG of the original problem is divided horizontally into *strips*, thereby obtaining p GDAGs of $\frac{m}{p}+1$ rows each. Two such contiguous strips share a common row. For $0 \le i < p$, processor P_i solves sequentially the ALCS problem for the strings $X_{mi/p+1}^{m(i+1)/p}$ and Y, and computes the local D_G. From Theorem 1, the time necessary for the p processors to solve the ALCS subproblem in parallel is $O(mn/p)$.

Then we use $\log p$ rounds to join the results, in which pairs of partial solutions (for two neighboring strips) are joined to give a single solution for the union of the two strips. At each union step, the number of processors associated to each strip doubles. After $\log p$ rounds we have the solution of the original problem. The sum of the times of all the union steps is $O(n\sqrt{m}(1+\frac{\log m}{\sqrt{p}}))$, as will be seen. Figure 4 illustrates the union process, with $p = 8$. In each strip of the GDAG G we indicate the processors used in the solution of the GDAG of the strip.

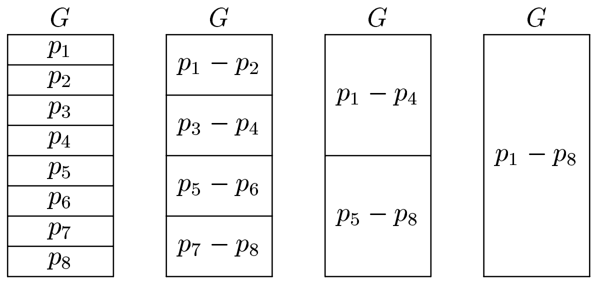

Figure 4. Joining the partial solutions of ALCS.

Thus the BSP/CGM algorithm for ALCS consists of the two phases:

1. Each of the p processors runs the sequential ALCS algorithm on its local *strip* and computes the local D_G.

2. In $\log p$ rounds pairs of the contiguous partial solutions are joined together successively to obtain the solution of the original problem.

The most difficult part of the algorithm involves the union of two contiguous strips. In particular we need to pay special attention to the storage of D_G using a compact data structure as well as the size of messages to be communicated among processors in each union step. This is dealt with in the next section. The union operation proper is presented in Section 6.

5 Compact Data Structures for D_G

For an $(n+1) \times (m'+1)$ GDAG G, the two representations of D_G we have seen so far are not adequate. The *direct* representation as matrices (as in Figure 3) presents redundancy and takes much space ($O(m'n)$), though the obtention of any individual value of $D_G(i,k)$, given i and k, can be done in O(1) time. On the other hand, the *incremental* representation through the use of the vectors D_G^0 and V_G uses only $O(m'+n)$ space but does not allow quick querying of values of D_G.

We now define a *compact* representation of D_G that takes $O(n\sqrt{m'})$ space and allows reads of D_G in $O(1)$ time. It can be constructed from D_G^0 and V_G in $O(n\sqrt{m'})$ time. This structure is essential to our algorithm. The construction is incremental by adding each row of D_G at a time. The values of D_G are stored in a vector called RD_G (*reduced* D_G) of size $O(nl)$, where $l = \lceil\sqrt{m'+1}\rceil$.

Before we describe the construction of RD_G, we give an overview of how a row of D_G is represented. Row D_G^i ($0 \leq i \leq n$), of size $m'+1$, is divided into at most l sub-vectors, all of size l with the possible exception of the last one. These sub-vectors are stored in separate locations of RD_G, and we need an additional vector of size l to indicate the location of each sub-vector. This additional vector is denoted by Loc^i. The $(n+1) \times l$ matrix formed by all the vectors Loc^i (one for each D_G^i) is called Loc. The indices of Loc start at 0 (see Figure 5). It can be easily shown that the value of $D_G(i,k)$ can be obtained with Loc and RD_G in $O(1)$ time.

Now we will show how to construct RD_G and Loc. First we include D_G^0. Each sub-vector of D_G^0 of size l is allocated in a fragment of RD_G of size $2l$. The extra space will be used to allocate the next rows of D_G. Thus each sub-vector of D_G^0 is followed by an empty space of size l. Since we have a lot of redundancies, the inclusion of the next row

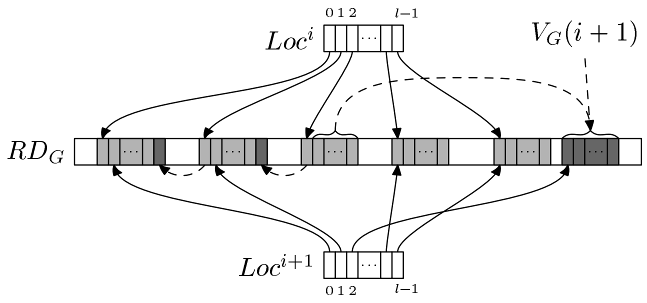

Figure 5. Storing D_G^i and D_G^{i+1} in RD_G.

of D_G can be done in a clever way. The vector RD_G (together with *Loc*) can contain all the data of D_G in only $O(nl)$ space due to the fact that the sub-vectors of different rows of D_G can overlap. With D_G^i already present, to include D_G^{i+1}, we can use most of the data already in the data structure. More precisely, the difference is that D_G^{i+1} does not have the value $D_G^i(0) = i$ but can have a new value given by $V_G(i+1)$. The inclusion of D_G^{i+1} (see Figure 5) consists of:

1. Determine the sub-vector of D_G^i to insert $V_G(i+1)$. Let v be the index of this sub-vector.

2. Determine the position in this sub-vector to insert $V_G(i+1)$.

3. All the sub-vectors of D_G^{i+1} of index *larger* than v are equal to those of D_G^i, thus already present in RD_G. It suffices to make $Loc(i+1,j) = Loc(i,j)$ for $v < j < l$.

4. All the sub-vectors of D_G^{i+1} of index *smaller* than v are equal to those of D_G^i, but for a left shift and for the inclusion of a new element to the right (precisely the element *thrown out* from the next sub-vector). The sub-vectors can be shifted to the left easily by making $Loc(i+1,j) = Loc(i,j)+1$ for $0 \leq j < v$. The new element of each sub-vector can be written immediately to the right of the sub-vector, given that there is empty space for this in RD_G. Otherwise we allocate a new fragment of size $2l$ and do the necessary bookkeeping.

5. The sub-vector of index v is modified such that a new sub-vector must be allocated in RD_G, with the inclusion of $V_G(i+1)$. $Loc(i+1,v)$ indicates the position of this new sub-vector.

In Figure 5 the element $V_G(i+1)$ must be inserted in the sub-vector of index 2. The darker areas represent the data written in DR_G in this inclusion step. The dashed arrows indicate copying of data. The inclusion of D_G^{i+1} involves the determination of a new row of *Loc* ($O(nl)$ time and space) and some new sub-vectors in RD_G for steps 4 and 5 ($O(nl)$ time and space in an amortized analysis).

We summarize the results of this section in the following.

Theorem 2 *Consider the GDAG G of the ALCS problem for the strings X and Y. From D_G^0 and V_G we can reconstruct a representation of D_G in $O\left(n\sqrt{m'}\right)$ time and space such that any value of D_G can be read in $O(1)$ time.*

6 The Basic Union Operation of Two Partial Solutions

The strategy of Section 4 utilizes one basic operation, namely the union of two contiguous strips to form a larger strip. After the last union operation we obtain the D_G matrix corresponding to the original GDAG. Let us consider the union of two GDAGs U and L of $m' + 1$ rows each, resulting in a GDAG G of $2m' + 1$ rows. We use the given example to illustrate the operation.

Consider D_U corresponding to the *upper* half of the GDAG (first 5 rows, from row 0 through row 4) and D_L corresponding to the *lower* half of the GDAG (rows 4 through row 8). D_U and D_L are shown in Figure 6 (the meaning of $\bullet$ and $\circ$ will be explained later). We first show how we can obtain D_G using D_U and D_L. The basic idea is the same as in [7].

$\mathbf{D_U}$	0	1	2	3	4	$\mathbf{D_L}$	0	1	2	3	4
0	0	1	2	3	4	0	0	1	2	6	9
1	1	2	3	4	10	1	1	2	5	6	9
2	2	3	4	7	10	2	2	3	5	6	9°
3	3	4	6	7	10	3	3	4	5	6°	$9^\bullet$
4	4	6	7	10	∞	4	4	5	6°	$8^\bullet$	9
5	5	6	7	10	∞	5	5	6	8	9	13
6	6	7	9	10	∞	6	6	7	8	9	13
7	7	9	10	13	∞	7	7	8	9	11	13
8	8	9	10	13	∞	8	8	9	11	13	∞
9	9	10	11	13	∞	9	9	10	11	13	∞
10	10	11	13	∞	∞	10	10	11°	$13^\bullet$	∞	∞
11	11	13	∞	∞	∞	11	11	12	13	∞	∞
12	12	13	∞	∞	∞	12	12	13	∞	∞	∞
13	13	∞	∞	∞	∞	13	13	∞	∞	∞	∞

Figure 6. D_U and D_L corresponding to the upper half and lower half of the GDAG.

Recall first $D_G^i(k) = D_G(i, k)$ represents the smallest value of j such that $C_G(i, j)$, the total weight of the best path between $T_G(i)$ (vertex i of the top row of GDAG G) and $B_G(j)$ (vertex j of the bottom row of GDAG G), is k. All the paths from $T_G(i) = T_U(i)$ to $B_G(j) = B_L(j)$ have to cross the common boundary $B_U = T_L$ at some vertex and the total weight of the path is the sum of the weights of the interval in U and in L. So if we are interested in determining $D_G(i, k)$ we need to consider paths that cross U with total weight l and then cross L with total weight $k - l$, for all l from 0 to m'.

Having fixed a certain value of l, the smallest value of j such that there is a path from $T_G(i)$ to $B_G(j)$ with weight l in U and weight $k - l$ in L is given by $D_L(D_U(i, l), k - l)$, since $D_U(i, l)$ is the first vertex at the boundary that is at a distance of l from $T_G(i)$ and $D_L(D_U(i, l), k - l)$ is the first vertex that is at a distance of $k - l$ from the vertex at the boundary.

By the above considerations we have:

$$D_G(i, k) = \min_{0 \le l \le m'} \{D_L(D_U(i, l), k - l)\} \qquad (1)$$

Observe that if we keep i fixed and vary k, the rows of D_L used are always the same. The variation of k changes only the element that must be consulted in each row. For each row of D_L consulted, a different element is taken, due to the term $-l$. This shift operation and the following observation suggest the following definitions of *shift*, *diag* and *MD*. Before giving these definitions let us consider the obtention of, say $D_G(1, 6)$ and $D_G(1, 5)$, by using Equation 1.

$$D_G(1, 6) = \min_{0 \le l \le m'} \{D_L(D_U(1, l), 6 - l)\}$$

This involves the minimum of the values marked with $\bullet$ in Figure 6, giving the value 8. On the other hand, to obtain

$$D_G(1, 5) = \min_{0 \le l \le m'} \{D_L(D_U(1, l), 5 - l)\},$$

we have to compute the minimum of the values marked with $\circ$, which is 6.

Notice that if we shift the appropriate rows of D_L (rows 1, 2, 3, 4, 10) to the right, with each row shifted one to the right with respect the the previous row, all the values whose minimum needs to be computed are aligned conveniently on the same column, with all new positions filled with ∞ (Figure 7). We see that all the minimum values of each respective column give the entire row 1 of D_G. The layout of Figure 7 is called $MD[1](i, j)$, which is formalized by the definitions of *shift* and *diag*. Note the white and black bullets are all aligned vertically. $MD[1]$ can be used to obtain row 1 of D_G.

$MD[\mathbf{1}](\mathbf{i}, \mathbf{j})$	0	1	2	3	4	5	6	7	8
0	1	2	5	6	9	∞	∞	∞	∞
1	∞	2	3	5	6	9°	∞	∞	∞
2	∞	∞	3	4	5	6°	$9^\bullet$	∞	∞
3	∞	∞	∞	4	5	6°	$8^\bullet$	9	∞
4	∞	∞	∞	∞	10	11°	$13^\bullet$	∞	∞
	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$	$\downarrow$
Minimum	1	2	3	4	5	6	8	9	∞

Figure 7. The matrix $MD[1]$.

Definition 4 (*shift*$[l, W, c]$) *Given a vector W of length $s+1$ (indices from 0 to s), for all l ($0 \le l \le c - s$) define* *shift*$[l, W, c]$ *as the vector of length* $c + 1$ *(indices from* 0 *to* c*) such that:*

$$\textit{shift}[l, W, c](i) = \begin{cases} \infty & \textit{if}\, 0 \leq i < l \\ W(i+l) & \textit{if}\, l \leq i \leq s+l \\ \infty & \textit{if}\, s+l < i \leq c \end{cases}$$

In other words, *shift*$[l, W, c]$ *is the vector* W *shifted to the right* l *positions and completed with* ∞.

With this definition we can rewrite Equation 1:

$$D_G(i, k) = \min_{0 \leq l \leq m'} \{\textit{shift}[D_L^{D_U(i,l)}, l, 2m'](k)\} \qquad (2)$$

By taking all the rows relative to a certain value of i we can obtain the matrix $MD[i]$, through the following definitions, to find *all* the elements of D_G^i.

Definition 5 **(**$\textit{Diag}[W, M, l_0]$**)** *Let* W *be a vector (starting index* 0*) of integers in increasing order such that the first* $\overline{m} + 1$ *elements are finite numbers and* $W(\overline{m}) \leq n$*. Let* M *be an* $(n+1) \times (m'+1)$ *matrix (starting indices* 0*).* $\textit{Diag}[W, M, l_0]$ *is an* $(\overline{m}+1) \times (2m'+1)$ *matrix such that its row of index* l *is* *shift*$[M^{W(l)}, l + l_0, 2m']$.

$\textit{Diag}[W, M, l_0]$ has its rows copied from a matrix M. The selection of which rows are copied is done by the vector W. Each row copied is shifted one column to the right in relation to the previous row. The amount to shift the first row copied is indicated by l_0.

Definition 6 **(**$MD[i]$**)** *Let* G *be a GDAG for the ALCS problem, formed by the union of the* U *(upper) and* L *(lower) GDAGs. Then* $MD[G, i] = \textit{Diag}[D_U^i, D_L, 0]$*. When* G *is clear in the context, we will use only the notation* $MD[i]$*.*

Figure 7 shows $MD[1]$ that can be used to obtain D_G^1. Thus by obtaining the minimum of each column of $MD[i]$ we get D_G^i. If we denote by $\textit{Cmin}[M]$ the values of the minimum of the respective column of matrix M, then we can write:

$$D_G^i(k) = \textit{Cmin}[MD[i]](k) \qquad (3)$$

A matrix is called *monotone* if the minimum of a column is below or to the right of the minimum of its right neighbor column. If two or more elements have the minimum, take the upper element. A matrix is called *totally monotone* if every one of its 2×2 sub-matrices is monotone [1].

Theorem 3 *Matrix* $MD[i]$ *is totally monotone.*

The proof can be found in [3].

Given that all the matrices $MD[i]$ are totally monotone, the union of GDAGs can be solved through a search of the minimum of columns for all the matrices, through an algorithm based on [1] that takes only $O(m)$ time for each of the $n + 1$ matrices (since the matrices have height m).

Even with this algorithm, however, the total time is still $O(nm)$. This is not good enough. To solve the union problem in less time we observe that, given the similarity between adjacent rows of D_G, matrices $MD[i]$ are also similar for values close to i. This will be explored now.

7 Redundancy Elimination by Exploring Similarities

We explore the property that $r + 1$ consecutive rows of D_U are r-variant [7], i.e., to obtain any row from any other row we need only to remove at most r elements and insert at most r other elements. More importantly, with $r + 1$ consecutive rows of length $m' + 1$, it is possible to obtain a vector of elements that are common in all the $r + 1$ rows of size $m' + 1 - r$, that we call *common vector*. The elements of the common vector may be noncontiguous. We use the notation $D_U^{i_0,r}$ to indicate the common vector of the rows from $D_U^{i_0}$ to $D_U^{i_0+r}$ for a GDAG U. We use $r = \lceil\sqrt{m'}\rceil$.

It can be shown that all the elements of a vector $D_U^{i_0}$ are also present in the vector $D_U^{i_0+r}$, except those that are smaller than $i_0 + r$. For example, consider D_U of Figure 6, we have $m' = 4$ and $r = 2$. The elements common to $D_U^0 = (0, 1, 2, 3, 4)$ and $D_U^2 = (2, 3, 4, 7, 8)$ are 2, 3, 4 (the last ones of D_G^0).

It can also be shown that all the elements present in both vectors $D_U^{i_0}$ and $D_U^{i_0+r}$ are also present in vectors D_U^i for $i_0 < i < i_0 + r$. So we have a simple form of determining the common vector for a group of rows: we just need to take the matrix D_U and read all the elements of $D_U^{i_0}$ ignoring those that are smaller than $i_0 + r$. This takes $O(m')$ time. In the previous example, we have $D_U^{0,2} = (2, 3, 4)$.

Furthermore, it is important to determine which elements are adjacent in all the rows and form the *indivisible pieces* that will be called *common fragments*. In the example, taking rows 6, 7, 8, $D_U^{6,2} = (9, 10, \infty)$ and the common fragments are $(9, 10)$ and (∞).

Determination of the common fragments can be done using the vector V_U. From $V_U(i_0 + 1)$ to $V_U(i_0 + r)$ we have the elements that do not appear in the common vector and can be used to divide it into the common fragments. This takes $O(\log m')$ time for each element and a total time of $O(r \log m')$. The common fragments are numbered from 0 to r and the fragment t will be denoted $D_U^{i_0,r}[t]$.

Now let us go back the union process of the algorithm. We obtain the common vector and common fragments of the matrix D_U. Instead of constructing $MD[i]$ for each individual i, we use rows $D_U^{i_0}$ to $D_U^{i_0+r}$ to construct matrices $MD[i_0]$ to $MD[i_0 + r]$ and, as already mentioned, the similarities between rows close to each other imply similarities

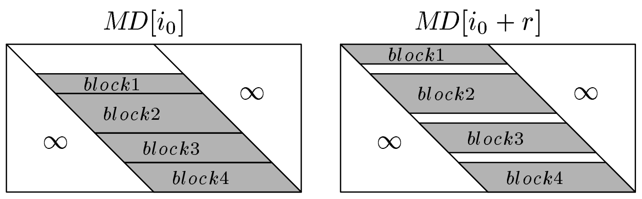

Figure 8. Structure of the matrices $MD[i_0]$ and $MD[i_0+r]$, showing four common blocks.

between matrices $MD[i]$ for values close to i. The common vector $D_U^{i_0,r}$ contains the indices of the rows of D_L that are present in all the matrices of $MD[i_0]$ to $MD[i_0+r]$. Each fragment $D_U^{i_0,r}[t]$ indicates a set of rows of D_L, called *common blocks*, that can be used in *adjacent* rows in all these matrices.

Consider a common fragment $D_U^{i_0,r}[t]$, $0 \le t \le r$. All the matrices $MD[i]$ with $i_0 \le i \le i_0 + r$ will contain $Diag[D_U^{i_0,r}[t], D_L, l_{i,t}]$ as a set of contiguous rows from the row $l_{i,t}$, where $l_{i,t}$ varies from matrix to matrix and from block to block. This is illustrated in Figure 8.

As in each matrix it is necessary to solve the problem of the column minima, we can avoid the repetitive computation by determining first the column minima of the common blocks. We have thus the following definition:

Definition 7 ($ContBl[i_0,r,t]$)

$$ContBl[i_0,r,t] = Cmin[Diag[D_U^{i_0,r}[t], D_L, 0]]$$

In other words, $ContBl[i_0,r,t]$ is a vector of the minima of the columns of the block t common to the matrices $MD[i_0]$ to $MD[i_0+r]$.

Consider now the idea of *row contraction* of a (totally) monotone matrix.

Definition 8 *[Contraction of rows of a (totally) monotone matrix] Let M be a (totally) monotone matrix. A row contraction applied to a set of contiguous rows is the substitution of all these rows in M by a single row. The element of column i of this new row is the minimum of the elements present in column i of the original substituted rows.*

It can be shown that after contraction the matrix continues to be (totally) monotone.

If for each matrix $MD[i]$ we do successive *contraction of rows*, one for each one of the common blocks, the result will be a matrix that we call $ContMD[i]$, such that $Cmin[MD[i]] = Cmin[ContMD[i]]$.

The contraction of each block can be done by an algorithm presented in [1] adapted for matrices with very few rows. A block t of m_t rows is contracted in $O(m_t \log(m'/m_t))$ time. For each row i in a particular block, this algorithm indicates the range of columns that have their minima in this row. This indirect representation of the column minima can be queried in $O(\log m')$ time for each element of $ContBl[i_0,r,t]$. In the following analysis, the $\log m'$ factor in the time complexities comes from this querying time. The contraction of all the blocks can be done in $O(m' \log m')$ time.

The common blocks appear in the matrices $MD[i]$ in different positions, but the result of the search for the minima of the columns of these blocks can be used in all the matrices. More precisely, if the common block t appears starting from row $l_{i,t}$ of the matrix $MD[i]$, the contraction of this block in this matrix is done by simply substituting of the block by the vector $shift[l_{i,t}, ContBl[i_0,r,t], 2m]$.

In addition to the $r+1$ rows obtained from the contraction of the common blocks, each matrix $ContMD[i]$ contains at most r additional rows. Recall that $r = \lceil\sqrt{m'}\rceil$. So the contraction of the matrices reduces the height of these matrices to $O(\sqrt{m'})$.

The construction of the matrices $ContMD[i_0]$ to $ContMD[i_0+r]$ can be done by simple pointer manipulations. In fact, to build $ContMD[i+1]$, we can use $ContMD[i]$, remove the first row and insert a new one. A structure similar to the one for D_G of Section 5 can be use here.

We use the algorithm from Aggarwal et al. [1] to find all the column minima of the first matrix of the range, $ContMD[i_0]$, obtaining row $D_G^{i_0}$ in $O(m' \log m')$ time. For row D_G^i, $i_0 < i \le i_0 + r$, using Property 1, we just need to determine $V_G(i)$ from D_G^{i-1}. We do this by taking from $ContMD[i]$ $O(\sqrt{m'})$ columns, spaced by $\lceil\sqrt{m'}\rceil$ and finding their minima in $O(\sqrt{m'}\log m')$ time. This gives us a $\lceil\sqrt{m'}\rceil$-sample of D_G^i, which can be compared to D_G^{i-1} to give us an interval of size $O(\sqrt{m'})$ where $V_G(i)$ can be found. To find $V_G(i)$ we do another determination of column minima in $O(\sqrt{m'}\log m')$ time. Doing so for r values of i, we spend $O(r\sqrt{m'}\log m') = O(m' \log m')$ time. The rows of D_G can be kept in the data structure described in Section 5 for comparing adjacent rows (the data for rows already used can be discarded).

Since there are a total of $(n+1)/(r+1)$ groups of $r+1$ matrices $MD[i]$ to process, the total time to determine D_G^0 and V_G from D_U e D_L is $O((nm'/r)\log m') = O(n\sqrt{m'}\log m')$. We can divide this work by using q processors, through the division of D_U among the processors. Each block of r rows of D_U can be used to determine the r rows of D_G, the processing of each block being independent of the other blocks. With this, the time becomes $O((n\sqrt{m'}\log m')/q)$.

We need to consider the time to construct the representation of D_U and D_L, as shown earlier. The representation of D_L must be available in the local memory of all the processors. This construction takes $O(n\sqrt{m'})$ time, or $O(n\sqrt{m'}/q)$ by using q processors. We thus have the following.

Lemma 1 *Let G' be a $(2m'+1) \times (n+1)$ GDAG for the ALCS problem, formed by the union of the $(m'+1) \times (n+1)$ U (upper) and L (lower) GDADs. The determination of $D^0_{G'}$ and $V_{G'}$ from D^0_U, V_U, D^0_L and V_L can be done by q processors in $O\left(n\sqrt{m'}\left(1+\frac{\log m'}{q}\right)\right)$ time and $O(n\sqrt{m'})$ space.*

8 Analysis of the Complete Algorithm

Given strings X and Y of lengths m and n, respectively, phase 1 of the BSP/CGM algorithm of Section 4 solves the ALCS for the p GDAGs defined for the p substrings of X in $O(mn/p)$ time.

We need to obtain the time complexity of phase 2. Consider a basic union operation that joins an upper GDAG U' and a lower GDAG L', say of sizes $(m'+1) \times (n+1)$ each, to produce a union GDAG G' of size $(2m'+1) \times (n+1)$. The value of m' doubles in each union step until the last one when we have the solution (when $2m' = m$).

When there are q processors to deal with GDAG G', we have $m' = \frac{mq}{2p}$. By Lemma 1, the time complexity becomes:

$$O\left(n\sqrt{m}\left(\sqrt{\frac{q}{2p}} + \frac{\log\frac{mq}{2p}}{\sqrt{2pq}}\right)\right) = O\left(\frac{n\sqrt{m}}{\sqrt{p}}\left(\sqrt{q} + \frac{\log m}{\sqrt{q}}\right)\right)$$

The amount of processors q involved in obtaining each GDAG doubles in each union process. Thus the sum of the times of all the union processes is:

$$O\left(n\sqrt{m}\left(1+\frac{\log m}{\sqrt{p}}\right)\right)$$

The details can be found in [3].

To get linear speedup we need to make this time $O(mn/p)$. This is accomplished if $p < \sqrt{m}$. The space needed for the proposed algorithm is $O(n\sqrt{m})$ per processor, due to the representation of D_L in each union process.

As for the communication requirements, with q processors performing a union, each processor determines n/q elements of $V_{G'}$ that needs to be transmitted to the other $2q-1$ processors for the next union step. This results in $O(n)$ data transferred per processor. The processor that determines $D^0_{G'}$ needs to transfer the mq/p elements of this vector to other $2q-1$ processors, resulting in a communication round where $O(mq^2/p)$ data are transmitted.

For some constant C, this can also be done in C communication rounds in which each processor transmits $O((mq^{1+1/C})/p)$ data: in the first round the processor that determined $D^0_{G'}$ does a broadcast of this vector to $\lfloor q^{1/C} \rfloor$ other processors, that then transmits to $\lfloor q^{2/C} \rfloor$ other processors and so on. The last union step is when the largest amount of data is transmitted per processor, $O(mp^{1/C}+n)$.

We thus conclude this section with the main result of this paper which is a linear speedup BSP/CGM algorithm for the ALCS problem.

Theorem 4 *Given two strings X and Y of lengths m and n, respectively, the ALCS problem can be solved by $p < \sqrt{m}$ processors in $O(mn/p)$ time, $O(n\sqrt{m})$ space per processor, and $O(C \log p)$ communication rounds, for some chosen constant C, in which $O(mp^{1/C}+n)$ data are transmitted from/to each processor.*

Corollary 1 *Given two strings X and Y of lengths m and n, respectively, the ALCS problem can be solved in the BSP model with $p < \sqrt{m}$ processors in time $O(\frac{mn}{p} + C\log pL + \log p(Cmp^{1/C}+n)g)$, where g and L are the communication throughput ratio and communication latency, respectively.*

References

[1] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.

[2] C. E. R. Alves, E. N. Cáceres, F. Dehne, and S. W. Song. Parallel dynamic programming for solving the string editing problem on a CGM/BSP. In *Proceedings of the 14th Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 275–281. ACM Press, 2002.

[3] C. E. R. Alves, E. N. Cáceres, and S. W. Song. Sequential and parallel algorithms for the all-substrings longest common subsequence problem. Technical report, Dept. of Computer Science, University of São Paulo, November 2002.

[4] A. Apostolico and C. Guerra. The longest common subsequence problem revisited. *Algorithmica*, 2:315–336, 1987.

[5] F. Dehne. Coarse grained parallel algorithms. *Special Issue of Algorithmica*, 24(3/4):173–176, 1999.

[6] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel computational geometry for coarse grained multicomputers. In *Proc. 9th Annual ACM Symp. Comput. Geom.*, pages 298–307, 1993.

[7] M. Lu and H. Lin. Parallel algorithms for the longest common subsequence problem. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):835–848, 1994.

[8] C. Rick. New algorithms for the longest common subsequence problem. Technical Report 85123–CS, Institut fr Informatik, Universitt Bonn, 1994.

[9] J. Schmidt. All highest scoring paths in weighted graphs and their application to finding all approximate repeats in strings. *SIAM J. Computing*, 27(4):972–992, 1998.

[10] L. G. Valiant. A bridging model for parallel computation. *Communication of the ACM*, 33(8):103–111, 1990.