

110-62-CR

217897

318

---

## Efficient Parallel Algorithms for String Editing and Related Problems

*Alberto Apostolico*  
*Mikhail J. Atallah*  
*Lawrence Larmore*  
*H. S. McFaddin*

September 1988

Research Institute for Advanced Computer Science  
NASA Ames Research Center

RIACS Technical Report 88.26

NASA Cooperative Agreement Number NCC 2-387

(NASA-CR-185410) EFFICIENT PARALLEL  
ALGORITHMS FOR STRING EDITING AND RELATED  
PROBLEMS (Research Inst. for Advanced  
Computer Science) 31 p

N89-26449

C S C L 09B

Unclas

G3/62 0217897

# RIACS

Research Institute for Advanced Computer Science

---

# Efficient Parallel Algorithms for String Editing and Related Problems

Alberto Apostolico\*  
Mikhail J. Atallah<sup>†</sup>  
Lawrence Larmore<sup>‡</sup>  
H. S. McFaddin<sup>§</sup>

Research Institute for Advanced Computer Science  
NASA Ames Research Center

RIACS Technical Report 88.26  
September 1988

The string editing problem for input strings  $x$  and  $y$  consists of transforming  $x$  into  $y$  by performing a series of weighted edit operations on  $x$  of overall minimum cost. An edit operation on  $x$  can be the deletion of a symbol from  $x$ , the insertion of a symbol in  $x$  or the substitution of a symbol  $x$  with another symbol. This problem has a well known  $O(|x| |y|)$  time sequential solution [25]. We give the efficient PRAM parallel algorithms for the string editing problem. If  $m = (|x|, |y|)$  and  $n = \max(|x|, |y|)$ , then our CREW bound is  $O(\log m \log n)$  time with  $O(mn / \log m)$  processors. In all algorithms, space is  $O(mn)$ .

*Key words and phrases:* Strint-to-string correction, edit distances, spelling correction, longest common subsequence, shortest paths, grid graphs, analysis of algorithms, parallel computation, cascading divide-and-conquer

*AMS subject classification:* 68Q25

---

\*Dept. of Computer Science, Purdue University, West Lafayette, IN 47907. Current address: Dept. of Pure and Applied Mathematics University of L'Aquila, Italy. This author's research was supported in part by the Italian Ministry of Education and by the Italian National Research Council through IASI-CNR, Rome, Italy.

†Dept. of Computer Science, Purdue University, West Lafayette, IN 47907. This author's research was supported in part by Cooperative Agreement NCC 2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA) while the author was a Visiting Scientist at the Center for Advanced Architectures at RIACS, and by the Office of Naval Research under Grants N0014-84-K-0502 and N00014-86-K-0689, and the National Science Foundation under Grant DCR-8451393, with matching funds from AT&T.

‡Dept. of Computer Science, University of California, Irvine, CA 92717. This author's research was supported by the University of California, Irvine.

§Dept. of Computer Science, Purdue University, West Lafayette, IN 47907. This author's research was supported in part by Cooperative Agreement NCC 2-387 between the National Aeronautics and Space Administration (NASA) and the Universities Space Research Association (USRA) while the author was visiting the Center for Advanced Architectures at RIACS.

# Efficient Parallel Algorithms for String Editing and Related Problems

Alberto Apostolico\*  
Purdue University

Mikhail J. Atallah†  
Purdue University

Lawrence L. Larmore‡  
U. of California, Irvine

H. S. McFaddin§  
Purdue University

## Abstract

The string editing problem for input strings  $x$  and  $y$  consists of transforming  $x$  into  $y$  by performing a series of weighted edit operations on  $x$  of overall minimum cost. An edit operation on  $x$  can be the deletion of a symbol from  $x$ , the insertion of a symbol in  $x$  or the substitution of a symbol of  $x$  with another symbol. This problem has a well known  $O(|x||y|)$  time sequential solution [25]. We give efficient PRAM parallel algorithms for the string editing problem. If  $m = \min(|x|, |y|)$  and  $n = \max(|x|, |y|)$ , then our CREW bound is  $O(\log m \log n)$  time with  $O(mn/\log m)$  processors. Our CRCW bound is  $O((\log n (\log \log m)^2))$  time with  $O(mn/\log \log m)$  processors. In all algorithms, space is  $O(mn)$ .

*Key words and phrases:* String-to-string correction, edit distances, spelling correction, longest common subsequence, shortest paths, grid graphs, analysis of algorithms, parallel computation, cascading divide-and-conquer

*AMS subject classification:* 68Q25

---

\*Dept. of Computer Science, Purdue University, West Lafayette, IN 47907. Current Address: Dept. of Pure and Applied Mathematics, University of L'Aquila, Italy. This author's research was supported in part by the Italian Ministry of Education and by the Italian National Research Council through IASI-CNR, Rome, Italy.

†Dept. of Computer Science, Purdue University, West Lafayette, IN 47907. This author's research was supported by the Office of Naval Research under Grants N00014-84-K-0502 and N00014-86-K-0689, and the National Science Foundation under Grant DCR-8451393, with matching funds from AT&T. Part of this research was carried out while this author was visiting the Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, California.

‡Dept. of Computer Science, University of California, Irvine, CA 92717. This author's research was supported by the University of California, Irvine.

§Dept. of Computer Science, Purdue University, West Lafayette, IN 47907. This author's research was carried out while he was at RIACS, NASA Ames Research Center, California.

# 1. Introduction

One of the major goals of parallel algorithm design for PRAM models is to come up with parallel algorithms that are both *fast* and *efficient*, i.e. that run in polylog time while the product of their time and processor complexities is within a polylog factor of the time complexity of the best sequential algorithm for the problem they solve. This goal has been elusive for many simple problems that are trivially in the class NC (recall that NC is the class of problems that are solvable in  $O(\log^{O(1)} n)$  parallel time by a PRAM using a polynomial number of processors). For example, topological sorting of a DAG and finding a breadth-first search tree of a graph are problems that are trivially in NC, and yet it is not known whether either of them can be solved in polylog time with  $n^2$  processors.

This paper gives parallel algorithms for the string editing problem that are both fast and efficient in the above sense. We give a CREW-PRAM algorithm that runs in  $O(\log m \log n)$  time with  $O(mn/\log m)$  processors, where  $m$  (resp.  $n$ ) is the length of the shorter (resp. longer) of the two input strings. We also give a CRCW-PRAM algorithm that runs in  $O(\log n (\log \log m)^2)$  time with  $O(mn/\log \log m)$  processors. In both algorithms, space is  $O(mn)$ .

In related work, Ranka and Sahni [17] have designed a hypercube algorithm for  $m = n$  that runs in  $O(\sqrt{n \log n})$  time with  $n^2$  processors, and have considered time/processor tradeoffs. In independent work, Mathies [15] has obtained a CRCW-PRAM algorithm for the edit distance that runs in  $O(\log n \log m)$  time with  $O(mn)$  processors if the weight of every edit operation is smaller than a given constant integer.

Recall that the CREW-PRAM model of parallel computation is the synchronous shared-memory model where concurrent reads are allowed but no two processors can simultaneously attempt to write in the same memory location (even if they are trying to write the same thing). The CRCW-PRAM differs from the CREW-PRAM in that it allows many processors to write simultaneously in the same memory location: in any such common-write contest, only one processor succeeds, but it is not known in advance which one.

The rest of this introduction reviews the problem, its importance, and how it can be viewed as a shortest-paths problem on a special type of graph.

Let  $x$  be a string of  $|x|$  symbols on some alphabet  $I$ . We consider three *edit operations* on  $x$ , namely, *deletion* of a symbol from  $x$ , *insertion* of a new symbol in  $x$  and *substitution* of one of the symbols of  $x$  with another symbol from  $I$ . We assume that each edit operation

has an associated nonnegative real number representing the *cost* of that operation. More precisely, the cost of deleting from  $x$  an occurrence of symbol  $a$  is denoted by  $D(a)$ , the cost of inserting some symbol  $a$  between any two consecutive positions of  $x$  is denoted by  $I(a)$  and the cost of substituting some occurrence of  $a$  in  $x$  with an occurrence of  $b$  is denoted by  $S(a, b)$ . An *edit script* on  $x$  is any consistent (i.e., all edit operations are viable) sequence  $\sigma$  of edit operations on  $x$ , and the cost of  $\sigma$  is the sum of all costs of the edit operations in  $\sigma$ .

Now, let  $x$  and  $y$  be two strings of respective lengths  $|x|$  and  $|y|$ . The *string editing problem* for input strings  $x$  and  $y$  consists of finding an edit script  $\sigma'$  of minimum cost that transforms  $x$  into  $y$ . The cost of  $\sigma'$  is the *edit distance from  $x$  to  $y$* . In various ways and forms, the string editing problem arises in many applications, notably, in text editing, speech recognition, machine vision and, last but not least, molecular sequence comparison. For this reason, this problem has been studied rather extensively in the past, and forms the object of several papers (e.g. [13,14,16,18,20,19,25], to list a few). The problem is solved by a serial algorithm in  $\Theta(|x||y|)$  time and space, through dynamic programming (cf. for example, [25]). Such a performance represents a lower bound when the queries on symbols of the string are restricted to tests of equality [2,26]. Many important problems are special cases of string editing, including the *longest common subsequence* problem and the problem of *approximate matching* between a pattern string and text string (see [11,21,23] for the notion of approximate pattern matching and its connection to the string editing problem). Needless to say that our solution to the general string editing problem implies similar bounds for all these special cases.

The criterion that subtends the computation of edit distances by dynamic programming is readily stated. For this, let  $C(i, j)$ , ( $0 \leq i \leq |x|$ ,  $0 \leq j \leq |y|$ ) be the minimum cost of transforming the prefix of  $x$  of length  $i$  into the prefix of  $y$  of length  $j$ . Let  $s_k$  denote the  $k$ -th symbol of string  $s$ . Then:

$$C(i, j) = \min\{C(i-1, j-1) + S(x_i, y_j), C(i-1, j) + D(x_i), C(i, j-1) + I(y_j)\},$$

for all  $i, j$ , ( $1 \leq i \leq |x|$ ;  $1 \leq j \leq |y|$ ). Hence  $C(i, j)$  can be evaluated row-by-row or column-by-column in  $\Theta(|x||y|)$  time [25]. Observe that, of all entries of the  $C$ -matrix, only the three entries  $C(i-1, j-1)$ ,  $C(i-1, j)$  and  $C(i, j-1)$  are involved in the computation of the final value of  $C(i, j)$ . Such interdependencies among the entries of the  $C$ -matrix induce an  $(|x| + 1) \times (|y| + 1)$  *grid* directed acyclic graph (grid DAG for short) associated with the string editing problem.

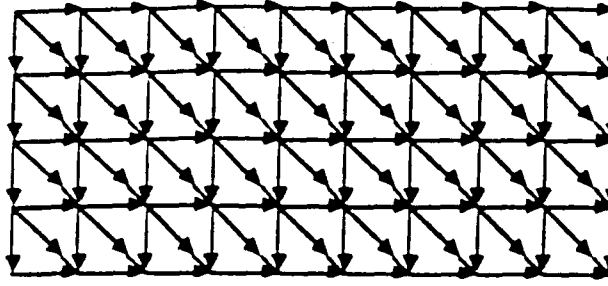


Figure 1. Example of a  $5 \times 10$  grid DAG.

**Definition 1** An  $m \times n$  grid DAG is a directed acyclic graph whose vertices are the  $mn$  points of an  $m \times n$  grid, and such that the only edges from grid point  $(i, j)$  are to grid points  $(i, j+1)$ ,  $(i+1, j)$  and  $(i+1, j+1)$ .

Figure 1 shows an example of a grid DAG and also illustrates our convention of drawing the points such that point  $(i, j)$  is at the  $i$ -th row from the top and  $j$ -th column from the left. Note that the top-left point is  $(1, 1)$  and has no edge coming into it (i.e. is a *source*), and that the bottom-right point is  $(m, n)$  has no edge leaving it (i.e. is a *sink*).

We associate an  $(|x|+1) \times (|y|+1)$  grid DAG  $G$  with the string editing problem in the obvious way: the  $(|x|+1)(|y|+1)$  vertices of  $G$  are in one-to-one correspondence with the  $(|x|+1)(|y|+1)$  entries of the  $C$ -matrix, and the *cost* of an edge from vertex  $(k, l)$  to vertex  $(i, j)$  is equal to  $I(y_i)$  if  $k = i$  and  $l = j-1$ , to  $D(x_i)$  if  $k = i-1$  and  $l = j$ , to  $S(x_i, y_j)$  if  $k = i-1$  and  $l = j-1$ . We can restrict our attention to edit scripts which are not wasteful in the sense that they do no obviously inefficient moves like: inserting then deleting the same symbol, or changing a symbol into a new symbol which they then delete, etc. More formally, the only edit scripts considered are those that apply at most one edit operation to a given symbol occurrence. Such edit scripts that transform  $x$  into  $y$  or vice versa are in one to one correspondence to the weighted paths in  $G$  that originate at the source (which corresponds to  $C(0, 0)$ ) and end on the sink (which corresponds to  $C(|x|, |y|)$ ). Thus, in order to establish the complexity bounds claimed in this paper, we need only establish them for the problem of finding a shortest (i.e. least-cost) source-to-sink path in an  $m \times n$  grid DAG  $G$ . Throughout, the *left boundary* of  $G$  is the set of points in its leftmost column. The *right*, *top*, and *bottom* boundaries are analogously defined. The *boundary* of  $G$  is the union of its left, right, top and bottom boundaries.

The rest of the paper is organized as follows. Section 2 gives a preliminary CREW-PRAM algorithm for computing the length of a shortest source-to-sink path, assuming  $m = n$ . Section 3 gives an algorithm that uses a factor of  $\log m$  fewer processors than

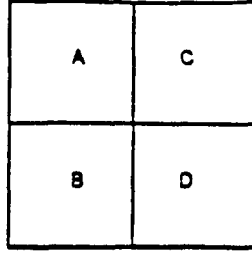


Figure 2. Illustrating how the problem is partitioned.

the previous one and that will be later needed in our best CREW algorithm (given in Section 6). Section 4 sketches how to extend the previous algorithm to the case  $m \leq n$ . Section 5 considers computing the path itself rather than just its length. Section 6 gives our best CREW-PRAM algorithm. Section 7 gives the CRCW-PRAM algorithm. Section 8 concludes.

## 2. A preliminary algorithm

Throughout this section,  $m = n$ , i.e.  $G$  is an  $m \times m$  grid DAG. Let  $DIST_G$  be a  $(2m) \times (2m)$  matrix containing the lengths of all shortest paths that begin at the top or left boundary of  $G$ , and end at the right or bottom boundary of  $G$ . In this section we establish that the matrix  $DIST_G$  can be computed in  $O(\log^3 m)$  time,  $O(m^2)$  space, and with  $O(m^2/\log m)$  processors by a CREW-PRAM. The preliminary algorithm that achieves this is intended as a “warmup” for the better algorithms that follow in later sections. The preliminary algorithm works as follows: divide the  $m \times m$  grid into four  $(m/2) \times (m/2)$  grids  $A, B, C, D$ , as shown in Figure 2. In parallel, recursively solve the problem for each of the four grids  $A, B, C, D$ , obtaining the four distance matrices  $DIST_A, DIST_B, DIST_C, DIST_D$ . Then obtain from these four matrices the desired matrix  $DIST_G$ . The main problem is how to perform this last step efficiently.

The performance bounds we claimed for this preliminary algorithm would immediately follow if we can show that, for any integer  $q \leq m$  of our choice,  $DIST_G$  can be obtained from  $DIST_A, DIST_B, DIST_C, DIST_D$  in time  $O((q + \log m) \log m)$  and with  $O(m^2/q)$  processors. This is because the time and processor complexities of the overall algorithm would then obey the following recurrences:

$$T(m) \leq T(m/2) + c_1(q + \log m) \log m$$

$$P(m) \leq \max(4P(m/2), c_2 m^2/q),$$

with boundary conditions  $T(\sqrt{q}) = c_3 q$  and  $P(\sqrt{q}) = 1$ , where  $c_1, c_2, c_3$  are constants. The solutions are  $T(m) = O((q + \log m) \log^2 m)$  and  $P(m) = O(m^2/q)$ . Choosing  $q = \log m$  would then establish the desired result. Therefore in the rest of this section, we merely concern ourselves with showing that  $DIST_G$  can be obtained from  $DIST_A, DIST_B, DIST_C, DIST_D$  in time  $O((q + \log m) \log m)$  and with  $O(m^2/q)$  processors.

Let  $DIST_{A \cup B}$  be the  $(3m/2) \times (3m/2)$  matrix containing the lengths of shortest paths that begin on the top or left boundary of  $A \cup B$  and end on its right or bottom boundary. Let  $DIST_{C \cup D}$  be analogously defined for  $C \cup D$ . The procedure for obtaining  $DIST_G$  performs the following Steps 1–3:

1. Use  $DIST_A$  and  $DIST_B$  to obtain  $DIST_{A \cup B}$ .
2. Use  $DIST_C$  and  $DIST_D$  to obtain  $DIST_{C \cup D}$ .
3. Use  $DIST_{A \cup B}$  and  $DIST_{C \cup D}$  to obtain  $DIST_G$ .

We only show how Step 1 is done, since the procedures for Steps 2 and 3 are very similar. First, note that the entries of  $DIST_{A \cup B}$  that correspond to shortest paths that begin and end on the boundary of  $A$  (resp.  $B$ ) are already available in  $DIST_A$  (resp.  $DIST_B$ ), and can therefore be obtained in  $O(q)$  time. Therefore we need only worry about the entries of  $DIST_{A \cup B}$  that correspond to paths that begin on the top or left boundary of  $A$  and end on the right or bottom boundary of  $B$ . Assign to every point  $v$  on the top or left boundary of  $A$  a group of  $m/q$  processors. The task of the group of  $m/q$  processors assigned to  $v$  is to compute the lengths of all shortest paths that begin at  $v$  and end on the right or bottom boundary of  $B$ . It suffices to show that it can indeed do this in time  $O((q + \log m) \log m)$ . Observe that:

$$DIST_{A \cup B}(v, w) = \min\{Dist_A(v, p) + Dist_B(p, w) \mid p \text{ is on the boundary common to } A \text{ and } B\} \quad (1)$$

Using (1) to compute  $DIST_{A \cup B}(v, w)$  for a given  $v, w$  pair is trivial to do in time  $O(q + \log(m/q))$  by using  $O(m/q)$  processors for each such pair, but that would require an unacceptable  $O(m^3/q)$  processors. We have only  $m/q$  processors assigned to  $v$  for computing  $DIST_{A \cup B}(v, w)$  for all  $w$  on the bottom or right boundary of  $B$ . Surprisingly, these  $m/q$  processors are enough for doing the job in time  $O((q + \log(m/q)) \log m)$ . The procedure is given below.





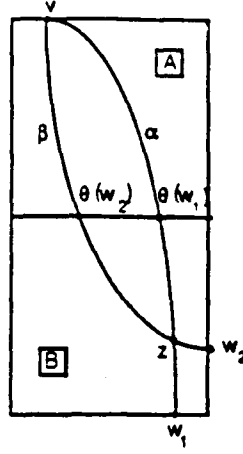


Figure 4. Illustrating the proof of property (2).

The procedure returns  $\theta(w)$  for every  $a <_B w <_B c$ . If  $r = 1$  then there is only one such  $w$  and there are enough processors to compute  $\theta(w)$  in time  $O(q + \log(\rho/q))$ . If  $r > 1$  then all of the  $\max\{1, (\rho + r)/q\}$  processors get assigned to the median of the  $a$ -to- $c$  range and compute, for that median (call it point  $b$ ), the value  $\theta(b)$  in time  $O(q + \log(\rho/q))$ . Because of (2), it is now enough for the procedure to recursively call itself on the  $a$ -to- $b$  range and (in parallel) the  $b$ -to- $c$  range. The first (resp. second) of these recursive calls gets assigned  $\max\{1, (\rho_1 + r/2)/q\}$  (resp.  $\max\{1, (\rho_2 + r/2)/q\}$ ) processors, where  $\rho_1$  (resp.  $\rho_2$ ) is the number of points between  $\theta(a)$  and  $\theta(b)$  (resp. between  $\theta(b)$  and  $\theta(c)$ ). Because  $\rho_1 + \rho_2 = \rho$ , there are enough processors available for the two recursive calls. (See Figure 3.) In the initial call to the procedure, it is given (i) the whole list  $L_B$ , (ii) the  $\theta$  of the first and last point of  $L_B$ , and (iii)  $3m/2q$  processors. The depth of the recursion is  $\log m$ , at each level of which the time taken is no more than  $O(q + \log(m/q))$ . Therefore the procedure takes time  $O((q + \log(m/q)) \log m)$  with  $O(m/q)$  processors. We conclude that the preliminary solution would immediately follow if we establish (2).

We prove (2) by contradiction: Suppose that, for some  $w_1, w_2 \in L_B$ , we have  $w_1 <_B w_2$  and  $\theta(w_1)$  is to the right of  $\theta(w_2)$ , as shown in Figure 4. By definition of the function  $\theta$  there is a shortest path from  $v$  to  $w_1$  going through  $\theta(w_1)$  (call this path  $\alpha$ ), and one from  $v$  to  $w_2$  going through  $\theta(w_2)$  (call it  $\beta$ ). Since  $w_1 <_B w_2$  and  $\theta(w_1)$  is to the right of  $\theta(w_2)$ , the two paths  $\alpha$  and  $\beta$  must cross at least once somewhere in  $B$ : let  $z$  be such an intersection point. See Figure 4. Let *prefix*( $\alpha$ ) (resp. *prefix*( $\beta$ )) be the portion of  $\alpha$  (resp.  $\beta$ ) that goes from  $v$  to  $z$ . We obtain a contradiction in each of two possible cases:

- Case 1. The length of *prefix*( $\alpha$ ) differs from that of *prefix*( $\beta$ ). Without loss of

generality, assume it is the length of  $prefix(\beta)$  that is the smaller of the two. But then, the  $v$ -to- $w_1$  path obtained from  $\alpha$  by replacing  $prefix(\alpha)$  by  $prefix(\beta)$  is shorter than  $\alpha$ , a contradiction.

- **Case 2.** The length of  $prefix(\alpha)$  is same as that of  $prefix(\beta)$ . In  $\alpha$ , replacing  $prefix(\alpha)$  by  $prefix(\beta)$  yields another shortest path between  $v$  and  $w_1$ , one that crosses the boundary common to  $A$  and  $B$  at a point to the left of  $\theta(w_1)$ , contradicting the definition of the function  $\theta$ .

This completes the proof of (2).

A referee pointed out that ideas similar to those in this Section were independently found by Baruch Schieber and Uzi Vishkin.

### 3. Using fewer processors

This section gives an algorithm that has same time complexity as that of the previous section, but whose processor complexity is a factor of  $\log m$  better. This is more than a mere “warmup” for our best CREW algorithm of Section 6: the algorithm of Section 6 will actually use the technical result, given in this section, that  $DIST_{A \cup B}$  can be obtained from  $DIST_A$  and  $DIST_B$  with  $O(m^2)$  total work.

We establish the following lemma.

**Lemma 1** *Let  $G$  be an  $m \times m$  grid DAG. Let  $DIST_G$  be a  $(2m) \times (2m)$  matrix containing the lengths of all shortest paths that begin at the top or left boundary of  $G$ , and end at the right or bottom boundary of  $G$ . The matrix  $DIST_G$  can be computed in  $O(\log^3 m)$  time,  $O(m^2)$  space, and with  $O(m^2 / \log^2 m)$  processors by a CREW-PRAM.*

We prove the above lemma by giving an algorithm whose processor complexity is a  $\log m$  factor better than that of the preliminary solution of Section 2. We illustrate the method by showing how  $DIST_{A \cup B}$  can be obtained from  $DIST_A$  and  $DIST_B$  in  $O(\log^2 m)$  time and  $O(m^2 / \log^2 m)$  processors. The preliminary procedure for computing  $DIST_{A \cup B}$  can be seen to do a total amount of work which is  $O(m^2 \log m)$ . Our strategy will be to first give a procedure which has same time and processor complexities as the preliminary one, but which does a total amount of work which is only  $O(m^2)$ . Our claimed bounds for the computation of  $DIST_{A \cup B}$  from  $DIST_A$  and  $DIST_B$  will then follow from this improved procedure and from Brent’s theorem [5]:

**Theorem 1 (Brent)** *Any synchronous parallel algorithm taking time  $T$  that consists of a total of  $W$  operations can be simulated by  $P$  processors in time  $O((W/P) + T)$ .*

**Proof.** See [5]. •

There are actually two qualifications to Brent's theorem before one can apply it to a PRAM: (i) at the beginning of the  $i$ -th parallel step, we must be able to compute the amount of work  $W_i$  done by that step, in time  $O(W_i/P)$  and with  $P$  processors, and (ii) we must know how to assign each processor to its task. Both (i) and (ii) will trivially hold in our framework.

Let  $L_A$  and  $<_A$  be defined analogously to  $L_B$  and  $<_B$ , respectively. In other words,  $L_A$  is a list of the  $m$  points on the left and top boundaries of  $A$ , sorted in the order in which they are encountered by a walk that starts at the lowest point of the left boundary of  $A$  and ends at the rightmost point of the top boundary of  $A$  (i.e. sorted by increasing order according to the  $<_A$  relationship). A symmetric version of (2) holds, i.e., for any  $w \in L_B$  and any two points  $v_1$  and  $v_2$  of  $L_A$ , we have the following:

$$\text{If } v_1 <_A v_2 \text{ then } \theta(v_1, w) \text{ is not to the right of } \theta(v_2, w). \quad (3)$$

The proof of (3) is identical to that of (2) and is therefore omitted.

Let  $P$  be the  $m \times (m/2)$  submatrix of  $DIST_A$  containing the lengths of the shortest paths that begin at the top or left boundary of  $A$ , and end at its bottom boundary. Let  $Q$  be the  $(m/2) \times m$  submatrix of  $DIST_B$  containing the lengths of the shortest paths that begin at the top boundary of  $B$ , and end at its bottom or right boundary. By definition, the rows of  $P$  are indexed by the entries of  $L_A$ , the columns of  $Q$  are indexed by the entries of  $L_B$ , and the columns of  $P$  (hence the rows of  $Q$ ) are indexed by the  $m/2$  points at the common boundary of  $A$  and  $B$ , sorted from left to right. The problem we face is that of "multiplying" the  $m \times (m/2)$  matrix  $P$  and the  $(m/2) \times m$  matrix  $Q$  in the closed semiring  $(\min, +)$ . In matrix terminology,  $\theta(v, w)$  is the smallest index  $k$ ,  $1 \leq k \leq m/2$ , such that  $PQ(v, w) = P(v, k) + Q(k, w)$ . We give the procedure below for the (more general) case where  $P$  is an  $\ell \times h$  matrix, and  $Q$  is an  $h \times \ell$  matrix,  $\ell \leq 2h$ . The only structure of these matrices that our algorithm uses is the following property (4), which is merely a restatement of properties (2) and (3) using matrix terminology:

$$\forall (1 \leq v_1 < v_2 \leq \ell, 1 \leq w \leq \ell), \theta(v_1, w) \leq \theta(v_2, w), \text{ and } \theta(w, v_1) \leq \theta(w, v_2). \quad (4)$$

To compute the product of  $P$  and  $Q$  in the closed semiring  $(\min, +)$ , it suffices to compute  $\theta(v, w)$  for all  $1 \leq v, w \leq \ell$ . To compute the product  $PQ$  (i.e. the function  $\theta$ ), we use the following procedure which runs in  $O(\log \ell \log h)$  time and  $O(\ell h / \log h)$  processors and  $O(\ell h)$  total work:

1. Recursively solve the problem for the product  $P'Q'$  where  $P'$  (resp.  $Q'$ ) is the  $(\ell/2) \times h$  (resp.  $h \times (\ell/2)$ ) matrix consisting of the odd rows (resp. odd columns) of  $P$  (resp.  $Q$ ). This gives  $\theta(v, w)$  for all pairs  $(v, w)$  whose respective parities are (odd, odd). If  $Work(\ell, h)$  and  $T(\ell, h)$  denote the total work and time for this procedure, then this step does  $Work(\ell/2, h)$  work in  $T(\ell/2, h)$  time.
2. Compute  $\theta(v, w)$  for all pairs  $(v, w)$  of parities (even, odd). This is done as follows. In parallel for each odd  $w$ , assign  $h / \log h$  processors to  $w$ , with the task of computing  $\theta(v, w)$  for all even  $v$ . The fact that we already know  $\theta(v, w)$  for all odd  $v$ , together with property (4), implies that these  $h / \log h$  processors are enough to do the job in  $O(\log h)$  time. The work done is then  $O(h)$  for each such  $w$ , for a total of  $O(\ell h)$  work for this step.
3. Compute  $\theta(v, w)$  for all pairs  $(v, w)$  of parities (odd, even). The method used is identical to that of the previous step and is therefore omitted.
4. Compute  $\theta(v, w)$  for all pairs  $(v, w)$  of parities (even, even). The method is very similar to that of the previous two steps and is therefore omitted.

The time, processor, and work complexities of the above method satisfy the recurrences:

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

$$P(\ell, h) \leq \max\{P(\ell/2, h), \ell h / \log h\},$$

$$Work(\ell, h) \leq Work(\ell/2, h) + c_2 \ell h,$$

where  $c_1$  and  $c_2$  are constants. These recurrences imply that  $T(\ell, h) = O(\log \ell \log h)$ ,  $P(\ell, h) = O(\ell h / \log h)$ , and  $Work(\ell, h) = O(\ell h)$ . This, together with Theorem 1 (Brent's theorem) in which  $T = \log \ell \log h$ ,  $P = \ell h / q$ , and  $W = \ell h$ , implies that the above algorithm can be simulated by  $\ell h / q$  processors in  $O(q + \log \ell \log h)$  time. In our case, we have  $\ell = m$  and  $h = m/2$ , implying that  $PQ$  (and hence  $DIST_{A \cup B}$ ) can be obtained from  $P$  and  $Q$  in  $O(q + \log^2 m)$  time with  $O(m^2 / q)$  processors.

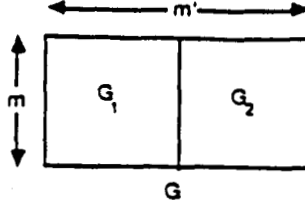


Figure 5. Illustrating Lemma 2.

The above method enables us to obtain  $DIST_G$  from  $DIST_A, DIST_B, DIST_C, DIST_D$  in  $O(q + \log^2 m)$  time and  $O(m^2/q)$  processors. This implies that the overall divide-and-conquer algorithm runs in  $O((q + \log^2 m) \log m)$  time with  $O(m^2/q)$  processors. Choosing  $q = \log^2 m$  establishes Lemma 1.

#### 4. The case $m \leq n$

This section generalizes the algorithm for the case  $m \leq n$ . The main result is the following.

**Theorem 2** *Let  $G$  be an  $m \times n$  grid DAG,  $m \leq n$ . The length of a shortest source-to-sink path in  $G$  can be computed by a CREW-PRAM in  $O(\log n \log^2 m)$  time,  $O(mn)$  space, and with  $O(mn/\log^2 m)$  processors.*

Note that, if  $G$  is  $m \times n$  with  $m \leq n$ , then using the same idea as in Section 3 would result in an unacceptable  $(m+n)(m+n)/\log^2(m+n)$  processor complexity, the  $DIST_G$  matrix we are computing now being  $(m+n) \times (m+n)$ . In order to prove our claimed bounds, we shall abandon the goal of computing such a matrix  $DIST_G$  and settle for computing a  $D_G$  matrix that contains less information than  $DIST_G$ , but enough to obtain the desired quantity: the length of a shortest source-to-sink path in  $G$ .

**Definition 3** *For any  $m \times n$  grid DAG  $G$ ,  $m \leq n$ , let  $D_G$  be the  $m \times m$  matrix containing the lengths of all the shortest paths that begin at the left boundary of  $G$ , and end at the right boundary of  $G$ .*

Note that  $D_G$  is a submatrix of  $DIST_G$ .

The following lemma is another ingredient that we need.

**Lemma 2** *Let  $G$  be an  $m \times m'$  grid DAG that is partitioned by a vertical line into  $G_1$  and  $G_2$ . (See Figure 5.) Then, given  $D_{G_1}$  and  $D_{G_2}$ , the matrix  $D_G$  can be computed by a CREW-PRAM in  $O(\log^2 m)$  time,  $O(m^2)$  space, and with  $O(m^2/\log^2 m)$  processors.*

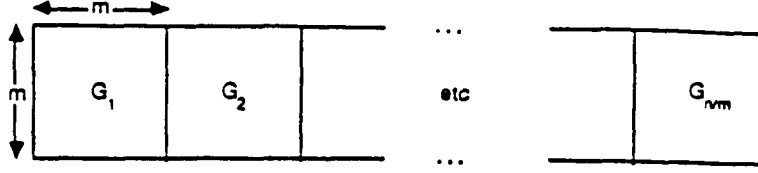


Figure 6. Illustrating the partitioning of  $G$ .

**Proof.** The algorithm proving the above lemma is similar to the procedure we used in Section 3 to obtain  $DIST_{A \cup B}$  from  $DIST_A$  and  $DIST_B$ , and is omitted. •

We are now ready to prove Theorem 2.

**Proof of Theorem 2.** Without loss of generality, assume that  $m$  divides  $n$  (if not then  $G$  can always be “padded” with extra vertices and zero-cost edges so as to make it  $m \times n'$  where  $m$  divides  $n'$  and  $n' - n \leq m$ ). Partition  $G$  by vertical lines into  $n/m$  grid DAGs  $G_1, \dots, G_{n/m}$ , where each  $G_i$  is  $m \times m$  (see Figure 6). In parallel for each  $i \in \{1, \dots, n/m\}$ , use Lemma 1 to obtain the  $DIST_{G_i}$  matrices. This takes  $O(\log^3 m)$  time with a total of  $O((m^2/\log^2 m)(n/m)) = O(mn/\log^2 m)$  processors. From each  $DIST_{G_i}$  matrix, extract its submatrix  $D_{G_i}$ . We are now left with the task of combining the  $D_{G_i}$ ’s into a single  $D_G$ . In parallel, we recursively obtain the  $D$ -matrix of the union of the leftmost  $n/2m$   $G_i$ ’s, and similarly the  $D$ -matrix of the union of the rightmost  $n/2m$   $G_i$ ’s. We then combine these two  $D$  matrices into  $D_G$  by using Lemma 2. This recursive combining procedure takes a total of  $O(\log^2 m \log(n/m))$  time with  $O(mn/\log^2 m)$  processors. The overall time complexity is therefore  $O(\log^3 m + \log^2 m \log(n/m)) = O(\log n \log^2 m)$ . •

In view of the remarks made in Section 1, the following is an immediate consequence of the above theorem.

**Corollary 1** *Let  $x$  and  $y$  be two strings over an alphabet  $I$ . Let  $m = \min(|x|, |y|)$ ,  $n = \max(|x|, |y|)$ . For edit operations of arbitrary nonnegative costs, the edit distance from  $x$  to  $y$  can be computed by a CREW-PRAM in  $O(\log n \log^2 m)$  time,  $O(mn)$  space, and with  $O(mn/\log^2 m)$  processors.*

## 5. Computing the actual path

In this section we sketch a modification of the algorithm given in the previous sections which enables us to compute an actual shortest source-to-sink path in  $G$  within the same time, space, and processor bounds as in the length computation.

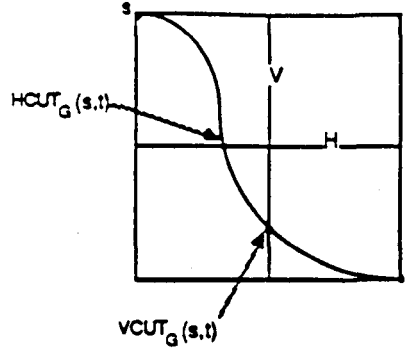


Figure 7. Illustrating the computation of the actual path.

**Theorem 3** Let  $G$  be an  $m \times n$  grid DAG,  $m \leq n$ . A shortest source-to-sink path in  $G$  can be computed by a CREW-PRAM in  $O(\log n \log^2 m)$  time,  $O(mn)$  space, and with  $O(mn / \log^2 m)$  processors.

The rest of this section proves the above theorem.

We begin with the case  $m = n$ , i.e. an  $m \times m$  grid DAG. We cannot afford to let the matrix  $DIST_G$  of Section 3 be a matrix of paths instead of lengths, because that would take  $m^3$  space, killing any hope of a polylog time algorithm that does not use an almost cubic number of processors. Instead, we modify the algorithm of Section 3 so that it also has the “side effect” of computing two  $(2m) \times (2m)$  matrices  $HCUT_G$  and  $VCUT_G$  (mnemonics for “horizontal cut” and “vertical cut”, respectively) having the same index domain as  $DIST_G$ . These two matrices are *global* in the sense that they remain even after the recursive call returns, and their significance is as follows. Let  $H$  be the horizontal boundary between  $A \cup C$  and  $B \cup D$ , and let  $V$  be the vertical boundary between  $A \cup B$  and  $C \cup D$  (see Figure 7). Let  $PATH(x, y)$  be the *lowest*  $x$ -to- $y$  path of cost  $DIST_G(x, y)$ ; i.e. no other  $x$ -to- $y$  path of length  $DIST_G(x, y)$  goes through any vertex that is below a vertex of  $PATH(x, y)$ . It is easy to prove that there is a unique such path  $PATH(x, y)$  (the proof is straightforward and is omitted). Then  $HCUT_G(x, y)$  is the leftmost intersection of  $PATH(x, y)$  with  $H$ , and  $VCUT_G(x, y)$  is the lowest intersection of  $PATH(x, y)$  with  $V$ . If the intersection of  $PATH(x, y)$  with  $H$  (resp.  $V$ ) is empty, then  $HCUT_G(x, y)$  (resp.  $VCUT_G(x, y)$ ) is undefined. Because these additional matrices are global, after the algorithm terminates it leaves behind  $N(m)$  of them where

$$N(m) = 4N(m/2) + 2 = O(m^2).$$



Fortunately, even though there are  $O(m^2)$  such *HCUT* and *VCUT* matrices that remain, the total storage space they take is  $S(m)$  where

$$S(m) = 4S(m/2) + cm^2 = O(m^2 \log m).$$

Before showing how  $S(m)$  is decreased to  $O(m^2)$ , we show how the matrices *HCUT* and *VCUT* are used to retrieve the shortest source-to-sink path in  $G$ . It suffices to output the points on this path as a set (i.e. in arbitrary order), since a postprocessing sorting step puts them in the right order in  $O(\log m)$  time and  $O(m)$  processors [6]. Let  $s$  and  $t$  denote the source and sink of  $G$ , respectively. We first print  $HCUT_G(s, t)$  and  $VCUT_G(s, t)$ , and then we recursively print the three portions of the shortest  $s$ -to- $t$  path determined by its two intersections with  $H$  and  $V$  (this involves three  $(m/2) \times (m/2)$  grid DAGs; see Figure 7). The procedure can be implemented to run in  $O(h + \log m)$  time and  $2m/h$  processors, where  $h \leq m$  is an integer of our choice, by maintaining the property that each recursive call of size  $m' \geq h$  gets assigned  $2m'/h$  processors (the bottom of the recursion is when problem size  $m'$  becomes  $\leq h$ , at which time a single processor finishes the job sequentially, in  $O(m')$  time). (We would, of course, choose  $h = \log m$ .)

We bring the space complexity  $S(m)$  down by storing each row (say, row  $p$ ) of the *HCUT* (or *VCUT*) matrix in an  $O(m)$ -bit vector  $ROW(p)$  that is “packed” in  $O(m/\log m)$  registers of size  $\log m$  bits each. (The assumption that word size is a logarithmic function of problem size is a standard one [3].) Let us immediately point out that a consequence of this encoding scheme is that we now have  $S(m) = O(m^2)$ . To see this, let  $BITS(m)$  be the total number of bits used by the encoding scheme, and note that  $S(m) = O(BITS(m)/\log m)$ , since each register contains  $\log m$  bits. Thus it suffices to show that  $BITS(m) = O(m^2 \log m)$ . But this trivially follows from the fact that  $BITS(m) = 4BITS(m/2) + O(m^2)$ .

We now describe the encoding scheme used for storing row  $p$  of (e.g.) *HCUT* in the  $O(m)$ -bit vector  $ROW(p)$ . We exploit the fact that the contents of row  $p$  happen to be sorted by the left-to-right linear ordering of the points on  $H$ . More precisely, if the points of  $H$  are denoted by  $1, \dots, m$  in left-to-right order, then row  $p$  contains a nondecreasing sequence of  $O(m)$  integers between 1 and  $m$ . Instead of storing the entries of row  $p$ , we therefore store the sequence of *differences* between the consecutive entries of row  $p$ . This sequence of differences is stored in unary in the  $O(m)$ -bit vector  $ROW(p)$ , with as many consecutive 1's as needed to encode a particular difference, and using a 0 as a separator between consecutive non-zero entries. For example, if row  $p$  contains the sequence  $(3, 3, 5, 7, 9, 11)$  then the

sequence of differences is  $(3, 0, 2, 2, 2, 2)$  and  $ROW(p) = (11100110110110110)$ . We can actually obtain  $ROW(p)$  without going through the intermediate step of computing the sequence of differences: simply observe that if the  $i$ -th entry of row  $p$  is  $k$  then the  $(i+k)$ th entry of  $ROW(p)$  is a 0 (in our example, the fourth entry is 7 and hence the eleventh entry of  $ROW(p)$  is a 0). This observation implies that we can obtain  $ROW(p)$  in  $O(q + \log m)$  time with  $O(m/q)$  processors by first initializing all the entries of  $ROW(p)$  to 1, and then changing some of these into 0's according to the observation. Reading the  $k$ -th entry of row  $p$  is now done by computing the sum of all the entries of  $ROW(p)$  that precede its  $k$ -th leftmost zero; i.e. it requires a parallel prefix computation [10] on  $ROW(p)$  and hence  $O(\log m)$  time, so that extracting the  $s$ -to- $t$  path now takes  $O(\log^2 m)$  time rather than the previous  $O(\log m)$ . This fact is of no consequence, however, since the bottleneck in the time complexity comes from the computation of the  $DIST_G$  matrix.

This completes the proof of Theorem 3 for the case  $m = n$ .

It is not hard to see that, so long as  $m = n$ , the above procedure actually works when  $s$  and  $t$  are arbitrary points on the boundary of  $G$ . This observation implies that, for the case  $m \leq n$ , it suffices to find for every  $i \in \{1, \dots, (n/m) - 1\}$  the lowest point (call it  $CROSS(i)$ ) at which a shortest path from  $s$  to  $t$  crosses the boundary between  $G_i$  and  $G_{i+1}$ . Once we have these  $CROSS(i)$ 's, we can use the procedure of the previous paragraph to obtain the actual path joining each  $CROSS(i)$  to  $CROSS(i+1)$  in time  $O(\log^3 m)$ , space  $O(m^2 n/m) = O(mn)$  and with  $O((m^2/\log^2 m)(n/m)) = O(mn/\log^2 m)$  processors. We obtain the  $CROSS(i)$ 's as follows. Refer to Section 4, the proof of Theorem 2: We modify that procedure so that, as the procedure computes the  $D$ -matrix, it now also produces as a side effect a global  $m \times m$  matrix  $CUT_G$ . The significance of this matrix is that  $CUT_G(x, y)$  is the lowest point of intersection of any shortest  $x$ -to- $y$  path with the boundary separating the two recursive calls. The total number of such  $CUT$  matrices is  $O(n/m)$ , and their total storage is  $O(mn)$ . We use these  $CUT$  matrices to output the  $CROSS(i)$ 's as a set (i.e. unordered) by first printing  $CUT_G(s, t)$ , and then recursively printing the  $CROSS(i)$ 's that are to the left of  $CUT_G(s, t)$ , and simultaneously (i.e. in parallel) those to its right. It is easily seen that the  $CROSS(i)$ 's are produced in time  $O(\log(n/m))$ , and that there are enough processors to carry out the procedure. A post-processing sorting step orders the  $CROSS(i)$ 's. This completes the proof of Theorem 3. •

An immediate consequence of Theorem 3 is the following.

**Corollary 2** *Let  $x$  and  $y$  be two strings over an alphabet  $I$ . Let  $m = \min(|x|, |y|)$ ,  $n = \max(|x|, |y|)$ . For edit operations of arbitrary nonnegative costs, an optimal edit script from  $x$  to  $y$  can be computed by a CREW-PRAM in  $O(\log n \log^2 m)$  time,  $O(mn)$  space, and with  $O(mn/\log^2 m)$  processors.*

## 6. A faster CREW-PRAM algorithm

This section gives a CREW algorithm that is faster by a  $\log m$  factor and uses  $O(mn/\log m)$  processors. More precisely, we establish the following.

**Theorem 4** *Let  $G$  be an  $m \times n$  grid DAG,  $m \leq n$ . A shortest source-to-sink path in  $G$  can be computed by a CREW-PRAM in  $O(\log n \log m)$  time,  $O(mn)$  space, and with  $O(mn/\log m)$  processors.*

**Corollary 3** *Let  $x$  and  $y$  be two strings over an alphabet  $I$ . Let  $m = \min(|x|, |y|)$ ,  $n = \max(|x|, |y|)$ . For edit operations of arbitrary nonnegative costs, an optimal edit script from  $x$  to  $y$  can be computed by a CREW-PRAM in  $O(\log n \log m)$  time,  $O(mn)$  space, and with  $O(mn/\log m)$  processors.*

From the developments of sections 2–5, it should be clear that in order to establish the above theorem, it suffices to show that:

1. The matrix  $DIST_{A \cup B}$  can be obtained from  $DIST_A$  and  $DIST_B$  in  $O(\log m)$  time,  $O(m^2)$  space, and with  $O(m^2/\log m)$  processors, and
2. The matrix  $D_G$  can be obtained from  $D_{G_1}$  and  $D_{G_2}$  (see Definition 3 and Figure 5) in  $O(\log m)$  time,  $O(m^2)$  space, and with  $O(m^2/\log m)$  processors.

Since the proofs of 1 and 2 are very similar, we only give that for 1. Thus the rest of this section deals with how to obtain  $DIST_{A \cup B}$  from  $DIST_A$  and  $DIST_B$  in  $O(\log m)$  time,  $O(m^2)$  space, and with  $O(m^2)$  processors.

### 6.1. Obtaining one row of $DIST_{A \cup B}$

This subsection gives an  $O(\log m)$  time,  $O(m \log m)$  space, and  $O(m \log m)$  processor algorithm for obtaining one particular row of  $DIST_{A \cup B}$ , i.e. computing  $\theta(v, w)$  for a fixed  $v \in L_A$  and all  $w \in L_B$ . The fixed vertex  $v$  is implicit in the rest of this subsection, so that whenever we refer to a “path to  $w$ ” it is understood that this path originates at  $v$ .

We refer to the vertices on the boundary common to  $A$  and  $B$  (denoted  $A \cap B$  for short) as *crossing vertices* and number them  $c_1, c_2, \dots, c_{m/2}$ , where the numbering is from left to right along the common boundary. We refer to the vertices in  $L_B$  as *destination vertices* and denote them  $w_1, w_2, \dots, w_m$ , numbered according to  $<_B$ , their order in  $L_B$ .

**Definition 4** A crossing interval is a non-empty set of contiguous crossing vertices  $\{c_i, c_{i+1}, \dots, c_j\}$ .

We say that crossing interval  $I$  is *to the left of* crossing interval  $J$ , and  $J$  is *to the right of*  $I$ , if the rightmost vertex of  $I$  is to the left of the leftmost vertex of  $J$ .

**Definition 5** Let  $F \subseteq A \cap B$  and  $w \in L_B$ , i.e.  $F$  is a set of crossing vertices (not necessarily an interval) and  $w$  is a destination vertex. Let  $\theta_F(w)$  denote the leftmost crossing vertex in  $F$  incident to a  $(v, w)$  path that is shortest among all  $(v, w)$  paths constrained to pass through  $F$ .

Note that  $\theta_F(w)$  may differ from  $\theta(v, w)$ , but that  $\theta_{A \cap B}(w) = \theta(v, w)$ .

The following lemma is the analogue, for constrained paths, to property (2) of Section 2.

**Lemma 3** Let  $F \subseteq A \cap B$  and  $w_1, w_2 \in L_B$ . If  $w_1 <_B w_2$ , then  $\theta_F(w_1)$  is not to the right of  $\theta_F(w_2)$ .

**Proof.** Identical to that of property (2) of Section 2, and hence omitted. •

We now give an informal description of the algorithm.

If  $U$  is any set of destination vertices and  $I$  is any crossing interval, then we will define  $\theta_I(U)$  to be a data structure that contains enough information to determine  $\theta_I(w)$  for all  $w \in U$ . The details of that data structure will be explained later.

It is useful to think of the computation as progressing through the nodes of a tree  $T$  which we now proceed to define.

We define a crossing interval to be *diadic* if it is either  $A \cap B$  (i.e. it consists of all crossing vertices), or if it is the left or right half of a diadic crossing interval. Note that there are exactly  $m - 1$  diadic crossing intervals, which form a complete binary tree  $T$  rooted at  $A \cap B$ , and whose  $m/2$  leaves are the  $m/2$  crossing vertices (the  $i$ -th leaf of  $T$  containing  $c_i$ , the  $i$ -th leftmost crossing vertex). Thus the diadic crossing interval at an interior node of  $T$  is simply the union of the diadic crossing intervals of its two children in  $T$ . We can talk

about the *height* and the *children* of a diadic crossing interval (= its height and children in  $T$ ).

Since the  $m - 1$  diadic crossing intervals are the only crossing intervals we shall be interested in, from now on we simply say “interval” as a shorthand for “diadic crossing interval”. Thus whenever we refer to an interval  $I$  we are implicitly assuming that  $I \in T$ , i.e. that  $I$  is one of the  $m - 1$  diadic crossing intervals. We use  $|I|$  to denote the size of the interval, i.e. the number of crossing vertices in it. Observe that  $\sum_{I \in T} |I| = O(m \log m)$ . Thus we have enough processors to associate  $|I|$  of them with each interval  $I$  (i.e. node  $I$ ) of  $T$ . Similarly, we can afford to use  $O(|I|)$  space per interval  $I$ . The computation proceeds in  $2 \log m - 1$  stages, each of which takes constant time. The ultimate goal is for every interval  $I$  to compute  $\theta_I(L_B)$ . The structure of the algorithm is reminiscent of the *cascading divide-and-conquer* technique [6,4]: each  $I \in T$  will compute  $\theta_I(U)$  for progressively larger subsets  $U$  of  $L_B$ , subsets  $U$  that double in size from one stage to the next of the computation. We now proceed to state precisely what these subsets are.

**Definition 6** A  $k$ -sample of  $L_B$  is obtained by choosing every  $k$ -th element of  $L_B$  (i.e. every element whose rank in  $L_B$  is a multiple of  $k$ ). For example, a 4-sample of  $L_B$  is  $(w_4, w_8, \dots, w_m)$ . For  $k \in \{0, 1, \dots, \log m\}$ , let  $U_k$  denote an  $(m/2^k)$ -sample of  $L_B$ .

For example:

$$U_0 = \{w_m\},$$

$$U_1 = \{w_{m/2}, w_m\},$$

$$U_2 = \{w_{m/4}, w_{m/2}, w_{3m/4}, w_m\},$$

$$U_3 = \{w_{m/8}, w_{m/4}, w_{3m/8}, w_{m/2}, w_{5m/8}, w_{3m/4}, w_{7m/8}, w_m\},$$

...

$$U_{\log m} = \{w_1, w_2, \dots, w_m\} = L_B.$$

Note that  $|U_k| = 2^k = 2|U_{k-1}|$ .

At the  $t$ -th stage of the algorithm, an interval  $I$  of height  $h$  in  $T$  will use its  $|I|$  processors to compute, in constant time,  $\theta_I(U_{t-h})$  if  $h \leq t \leq h + \log m$ . It does so with the help of information from  $\theta_I(U_{t-1-h})$ ,  $\theta_{LeftChild(I)}(U_{t-h})$ , and  $\theta_{RightChild(I)}(U_{t-h})$ , all of which are available from the previous stage  $t - 1$ . If  $h > t$  or  $t > h + \log m$  then interval  $I$  does nothing during stage  $t$ . Thus before stage  $h$  the interval  $I$  lies “dormant”, then at stage  $t = h$  it first “wakes up” and computes  $\theta_I(U_0)$ , then at the next stage  $t = h + 1$  it computes  $\theta_I(U_1)$ , etc. At step  $t = h + \log m$  it computes  $\theta_I(U_{\log m})$ , after which it is done. The details of what

information  $I$  stores and how it uses its  $|I|$  processors to perform stage  $t$  in constant time are given below. First, we observe the following.

**Lemma 4** *The algorithm terminates after  $2 \log m - 1$  stages.*

**Proof.** After stage  $h + \log m$  every interval  $I$  of height  $h$  is done, i.e. it has computed  $\theta_I(L_B)$ . The root interval has height  $\log m - 1$  and thus is done after stage  $2 \log m - 1$ . •

Thus to establish the main claim of this subsection, it suffices to prove the following lemma.

**Lemma 5** *With  $|I|$  processors and  $O(|I|)$  space assigned to each interval  $I \in T$ , every stage of the algorithm can be completed in constant time.*

The rest of this subsection proves the above lemma.

We begin by describing the way in which an interval  $I$  at height  $h$  in  $T$  stores  $\theta_I(U_{t-h})$  using only  $|I|$  space. Rather than directly storing the values  $\theta_I(w)$  for all  $w \in U_{t-h}$  (which would require  $|U_{t-h}|$  space), we store instead the *inverse* mapping, which turns out to have a compact  $O(|I|)$  space encoding because of the monotonicity property guaranteed by Lemma 3. In other words, for each  $c \in I$ , let

$$\pi_I(c, t) = \{w \in U_{t-h} \mid \theta_I(w) = c\}.$$

Then Lemma 3 implies that the elements of  $\pi_I(c, t)$  are contiguous in the list  $U_{t-h}$ . More specifically, the sets  $\pi_I(c, t)$ ,  $c \in I$ , form a partition of the set  $U_{t-h}$  into  $|I|$  subsets each of which is either empty or contains contiguous elements in  $U_{t-h}$ . Therefore  $I$  does not need to store the elements of  $\pi_I(c, t)$  explicitly, but rather by just remembering where they begin and end in  $U_{t-h}$ , i.e.  $O(1)$  space for each  $c \in I$ . Of course  $U_{t-h}$  is itself not stored explicitly by  $I$ , since the height  $h$  and stage number  $t$  implicitly determine it. Thus  $O(|I|)$  space is enough for storing  $\pi_I(c, t)$  for all  $c \in I$ .

Interval  $I$  stores the sets  $\pi_I(c, t)$ ,  $c \in I$ , in an array  $RANGE_I$ , with entries  $RANGE_I(c) = (w_i, w_j)$  such that  $w_i$  (resp.  $w_j$ ) is the first (resp. last) element of  $U_{t-h}$  that belongs to  $\pi_I(c, t)$ . If  $\pi_I(c, t)$  is empty then  $RANGE_I(c)$  equals  $\emptyset$ . At stage  $t$  of the algorithm,  $I$  must update the  $RANGE_I$  array so that it changes from being a description of the  $\pi_I(c, t-1)$ 's to being a description of the  $\pi_I(c, t)$ 's. The rest of this subsection need only show how such an update is done in constant time by the  $|I|$  processors assigned to  $I$ . Of course, since we are ultimately interested in  $\theta_{A \cap B}(w)$  for every  $w \in L_B$ , at the end of the algorithm we

must run a postprocessing procedure which recovers this information from the  $RANGE_{A \cap B}$  array available at the root of  $T$ , i.e. it explicitly obtains  $\theta_{A \cap B}(w)$  for all  $w \in U_{\log m}$ . But this postprocessing is trivial to perform in  $O(\log m)$  time with  $O(m)$  processors, and we shall not concern ourselves with it any more.

In the rest of this subsection, intervals  $L$  and  $R$  are the left and (respectively) right children of  $I$  in  $T$ . Observe that, for any destination  $w$ ,  $\theta_I(w)$  is one of  $\theta_L(w)$  or  $\theta_R(w)$ . Furthermore, if  $\theta_I(w) = \theta_L(w)$  then  $\theta_I(w') \in L$  for every  $w'$  smaller than  $w$  (in the  $<_B$  ordering). Similarly, if  $\theta_I(w) = \theta_R(w)$  then  $\theta_I(w') \in R$  for any  $w'$  larger than  $w$ . (These observations follow from Lemma 3.)

The  $RANGE_I$  array alone is not enough to enable  $I$  to perform the updating required at stage  $t$ . In addition, at each stage  $t$ ,  $I$  must compute in a register called  $CRITICAL_I$  an entry  $Critical_I(t)$  defined as follows.

**Definition 7** *At each stage  $t$ , let the critical destination for  $I$ , denoted  $Critical_I(t)$ , be the largest  $w \in U_{t-h}$  such that  $\theta_I(w) = \theta_L(w)$ . If there is no such  $w$  (i.e. if  $\theta_I(w) = \theta_R(w)$  for all  $w \in U_{t-h}$ ), then  $Critical_I(t) = \emptyset$ .*

Note that Lemma 3 ensures that  $Critical_I(t)$  is well defined. We shall later show how storing and maintaining this critical destination enables  $I$  to update the  $RANGE_I$  array in constant time. Of course it also places on  $I$  the burden of updating its  $CRITICAL_I$  register so that after stage  $t$  it contains  $Critical_I(t)$  rather than  $Critical_I(t-1)$ . We shall later show that updating the  $CRITICAL_I$  register can be done in constant time as well.

We now complete this subsection by explaining how  $I$  performs stage  $t$ , i.e. how it obtains  $Critical_I(t)$  and the  $\pi_I(c, t)$ 's using the  $\pi_L(c, t-1)$ 's, the  $\pi_R(c, t-1)$ 's, and its previous critical index  $Critical_I(t-1)$ . The fact that the  $|I|$  processors can do this in constant time is based on the following three observations:

$Critical_I(t)$  is either the same as  $Critical_I(t-1)$ , or the successor of  $Critical_I(t-1)$  in  $U_{t-h}$ . (5)

If  $c \in L$  then  $\pi_I(c, t) = \pi_L(c, t-1) - \{\text{the elements of } \pi_L(c, t-1) \text{ that are larger than } Critical_I(t) \text{ in the } <_B \text{ ordering}\}$ . (6)

If  $c \in R$  then  $\pi_I(c, t) = \pi_R(c, t-1) - \{\text{the elements of } \pi_R(c, t-1) \text{ that are less than or equal to } Critical_I(t) \text{ in the } <_B \text{ ordering}\}$ . (7)

Correctness of (5)–(7) follows from the definitions. Their algorithmic implications are discussed next.

### Updating the $CRITICAL_I$ register

Relationship (5) implies that in order to update  $CRITICAL_I$  (i.e. compute  $Critical_I(t)$ ) all  $I$  has to do is determine which of  $Critical_I(t-1)$  or its successor in  $U_{t-h}$  is the correct value of  $Critical_I(t)$ . This is done as follows. If  $Critical_I(t-1)$  has no successor in  $U_{t-h}$  then  $Critical_I(t-1) = w_m$  and hence  $Critical_I(t) = Critical_I(t-1)$ . Otherwise the updating is done in the following two steps. For shorthand, let  $r$  denote  $Critical_I(t-1)$ , and let  $s$  denote the successor of  $r$  in  $U_{t-h}$ .

- The first step is to compute  $\theta_L(s)$  and  $\theta_R(s)$  in constant time. This involves a search in  $L$  (resp.  $R$ ) for the crossover  $c$  in  $L$  (resp.  $R$ ) whose  $\pi_L(c, t-1)$  (resp.  $\pi_R(c, t-1)$ ) contains  $s$ . These two searches in  $L$  and  $R$  are done in constant time with the  $|I|$  processors available. We explain how the search in  $L$  is done (that in  $R$  is similar and omitted).  $I$  assigns a processor to each  $c \in L$ , and that processor tests whether  $s$  is in  $\pi_L(c, t-1)$ ; the answer is “yes” for exactly one of those  $|L|$  processors and thus can be collected in constant time. Thus  $I$  can determine  $\theta_L(s)$  and  $\theta_R(s)$  in constant time.
- The next step consists of comparing which of the following two paths to  $s$  is better: the one through  $\theta_L(s)$ , or the one through  $\theta_R(s)$ . If the path through  $\theta_R(s)$  is the better of the two then  $Critical_I(t)$  is the same as  $Critical_I(t-1)$  and the  $CRITICAL_I$  register stays the same (containing  $r$ ). Otherwise  $Critical_I(t)$  is  $s$ , and we set  $CRITICAL_I$  equal to  $s$ . This comparison of the two paths and resulting update are done in constant time (by one processor, in fact).

We next show how the just computed  $Critical_I(t)$  value is used to compute the  $\pi_I(c, t)$ 's in constant time.

### Updating the $RANGE_I$ array

Relationship (6) implies the following for each  $c \in L$ :

1. If  $\pi_L(c, t-1)$  is to the left of  $Critical_I(t)$  then  $\pi_I(c, t) = \pi_L(c, t-1)$ .
2. If  $\pi_L(c, t-1)$  is to the right of  $Critical_I(t)$  then  $\pi_I(c, t) = \emptyset$ .
3. If  $\pi_L(c, t-1)$  contains  $Critical_I(t)$  then it consists of the portion of  $\pi_L(c, t-1)$  up to (and including)  $Critical_I(t)$ .



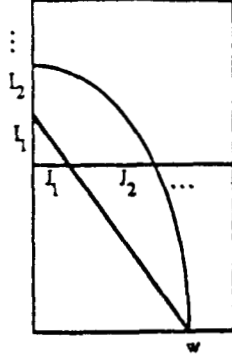


Figure 8. Illustrating the second stage of the computation.

The above facts 1–3 immediately imply that  $O(1)$  time is enough for  $|L|$  of the  $|I|$  processors assigned to  $I$  to compute  $\pi_I(c, t)$  for all  $c \in L$ , by adjusting the  $RANGE_I(c)$  value according to rules 1–3 above (recall that the  $\pi_L(c, t-1)$ 's are available in  $L$  from the previous stage  $t-1$ , and  $Critical_I(t)$  has already been computed and is in the  $CRITICAL_I$  register).

A similar argument shows that relationship (7) implies that  $|R|$  processors are enough for computing  $\pi_I(c, t)$  for all  $c \in R$ . Thus  $I$  can update its  $RANGE_I$  array in constant time with  $|I|$  processors. This completes the proof of Lemma 5.

## 6.2. Obtaining all rows of $DIST_{A \cup B}$

This subsection shows that  $O(m^2/\log m)$  processors and  $O(m^2)$  space suffice for computing in  $O(\log m)$  time all the  $\theta(v, w)$ 's (hence for computing the  $DIST_{A \cup B}$  matrix). Let  $L_A$  and  $L_B$  be as in previous sections. Our task is to compute  $\theta(v, w)$  for all  $v \in L_A$  and all  $w \in L_B$ . We use  $S(L, k)$  to denote the  $k$ -sample of a list  $L$ .

In the first stage of the computation, we assign  $m \log m$  processors to each  $v \in S(L_A, \log^2 m)$ . Then, in parallel for all  $v \in S(L_A, \log^2 m)$ , we use the method of the previous subsection to obtain  $\theta(v, w)$  for all  $w \in L_B$ . This first stage of the computation takes  $O(\log m)$  time,  $O(m^2)$  space, and  $O(m^2/\log m)$  processors, and obtains  $\theta(v, w)$  for all  $v \in S(L_A, \log^2 m)$  and  $w \in L_B$ .

In the second stage of the computation, we assign  $2m$  processors to each  $w \in S(L_B, \log m)$ , with the task of computing  $\theta(v, w)$  for all  $v \in L_A$ . These  $2m$  processors perform this computation for their particular  $w$  in  $O(\log m)$  time, as follows. The set of  $m/\log^2 m$  values  $\{\theta(v, w) \mid v \in S(L_A, \log^2 m)\}$  partitions the common boundary of  $A$  and  $B$  into  $m/\log^2 m$  pieces  $J_1, J_2, \dots$  (see Figure 8). Let  $I_1, I_2, \dots$  be the  $m/\log^2 m$  pieces (of size  $\log^2 m$  each)

into which  $S(L_A, \log^2 m)$  partitions  $L_A$  (see Figure 8). Partition the group of  $2m$  processors assigned to  $w$  into  $m/\log^2 m$  subgroups, where the  $i$ -th subgroup contains  $\log^2 m + |J_i|$  processors whose task is to compute, for all  $v \in I_i$ , which element of  $J_i$  equals  $\theta(v, w)$ . This subgroup of  $\log^2 m + |J_i|$  processors does this as follows.

1. It gives each of the  $\log m$  elements of  $S(I_i, \log m)$  (say, to element  $v$ )  $1 + |J_i|/\log m$  processors that  $v$  uses to find out, in  $O(\log m)$  time, which element of  $J_i$  equals  $\theta(v, w)$ . The set of  $\log m$  values  $\{\theta(v, w) \mid v \in S(I_i, \log m)\}$  partitions  $J_i$  into  $\log m$  pieces  $J_{i,1}, J_{i,2}, \dots$ . Let  $I_{i,1}, I_{i,2}, \dots$  be the  $\log m$  pieces (of size  $\log m$  each) into which  $S(I_i, \log m)$  partitions  $I_i$ .
2. It partitions its  $\log^2 m + |J_i|$  processors into  $\log m$  subsubgroups, where the  $k$ -th subsubgroup contains  $\log m + |J_{i,k}|$  processors whose task is to compute, for all  $v \in I_{i,k}$ , which element of  $J_{i,k}$  equals  $\theta(v, w)$ . This subsubgroup of  $\log m + |J_{i,k}|$  processors does this in  $O(\log m)$  time by giving to each of the  $\log m$  elements of  $I_{i,k}$  (say, to element  $v$ )  $1 + |J_{i,k}|/\log m$  processors that  $v$  uses to find out, in  $O(\log m)$  time, which element of  $J_{i,k}$  equals  $\theta(v, w)$ .

In the third stage of the computation, we assign  $2m/\sqrt{\log m}$  processors to each  $v \in S(L_A, \sqrt{\log m})$ , with the task of computing  $\theta(v, w)$  for all  $w \in L_B$ . These  $2m/\sqrt{\log m}$  processors perform this computation for their particular  $v$  in  $O(\log m)$  time, as follows. The set of  $m/\log m$  values  $\{\theta(v, w) \mid w \in S(L_B, \log m)\}$  partitions the common boundary of  $A$  and  $B$  into  $m/\log m$  pieces  $J_1, J_2, \dots$ . Let  $I_1, I_2, \dots$  be the  $m/\log m$  pieces (of size  $\log m$  each) into which  $S(L_B, \log m)$  partitions  $L_B$ . Partition the group of  $2m/\sqrt{\log m}$  processors assigned to  $v$  into  $m/\log m$  subgroups, where the  $i$ -th subgroup contains  $\sqrt{\log m} + |J_i|/\sqrt{\log m}$  processors whose task is to compute, for all  $w \in I_i$ , which element of  $J_i$  equals  $\theta(v, w)$ . This subgroup of  $\sqrt{\log m} + |J_i|/\sqrt{\log m}$  processors does this as follows.

1. It gives each of the  $\sqrt{\log m}$  elements of  $S(I_i, \sqrt{\log m})$  (say, to element  $w$ )  $1 + |J_i|/\log m$  processors that  $w$  uses to find out, in  $O(\log m)$  time, which element of  $J_i$  equals  $\theta(v, w)$ . The set of  $\sqrt{\log m}$  values  $\{\theta(v, w) \mid w \in S(I_i, \sqrt{\log m})\}$  partitions  $J_i$  into  $\sqrt{\log m}$  pieces  $J_{i,1}, J_{i,2}, \dots$ . Let  $I_{i,1}, I_{i,2}, \dots$  be the  $\sqrt{\log m}$  pieces (of size  $\sqrt{\log m}$  each) into which  $S(I_i, \sqrt{\log m})$  partitions  $I_i$ .
2. It partitions its  $\sqrt{\log m} + |J_i|/\sqrt{\log m}$  processors into  $\sqrt{\log m}$  subsubgroups. The  $k$ -th subsubgroup contains  $1 + |J_{i,k}|/\sqrt{\log m}$  processors whose task is to compute, for all

$w \in I_{i,k}$ , which element of  $J_{i,k}$  equals  $\theta(v, w)$ . This subsubgroup of  $1 + |J_{i,k}|/\sqrt{\log m}$  processors does this in  $O(\log m)$  time as follows:

- (a) If  $|J_{i,k}| \geq \log m$ , by giving to each of the  $\sqrt{\log m}$  elements of  $I_{i,k}$  (say, to element  $w$ )  $|J_{i,k}|/\log m$  processors that  $w$  uses to find out, in  $O(\log m)$  time, which element of  $J_{i,k}$  equals  $\theta(v, w)$ .
- (b) If  $|J_{i,k}| < \log m$ , by partitioning  $I_{i,k}$  into  $1 + |J_{i,k}|/\sqrt{\log m}$  equal pieces  $I_{i,k,1}, I_{i,k,2}, \dots$  (each of size roughly  $\log m/|J_{i,k}|$ ) and giving each  $I_{i,k,l}$  one processor. This processor sequentially finds  $\theta(v, w)$  for all  $w \in I_{i,k,l}$  in  $O(\log m)$  time, since  $|I_{i,k,l}||J_{i,k}| = O(\log m)$ .

The fourth stage of the computation "fills in the blanks" by actually computing  $\theta(v, w)$  for all  $v \in L_A$  and  $w \in L_B$ . It does so with only  $m^2/\log m$  processors by exploiting what was computed in the previous stages. Partition  $L_A$  into  $m/\sqrt{\log m}$  contiguous blocks  $X_1, X_2, \dots$  of size  $\sqrt{\log m}$  each. Similarly, partition  $L_B$  into  $m/\sqrt{\log m}$  contiguous blocks  $Y_1, Y_2, \dots$  of size  $\sqrt{\log m}$  each. Let  $Z_{ij}$  be the interval on the boundary common to  $A$  and  $B$  that is defined by the set of  $\theta(v, w)$  such that  $v \in X_i$  and  $w \in Y_j$ . Of course we already know the beginning and end of each such interval  $Z_{ij}$  (from the second and third stages of the computation). Furthermore, we have the following:

**Lemma 6**  $\sum_{i=1}^{m/\sqrt{\log m}} \sum_{j=1}^{m/\sqrt{\log m}} |Z_{ij}| = O(m^2/\sqrt{\log m})$ .

**Proof.** First, observe that  $Z_{ij}$  and  $Z_{i+1,j+1}$  are adjacent intervals that are disjoint except for one possible common endpoint (the rightmost point in  $Z_{ij}$  and the leftmost point in  $Z_{i+1,j+1}$  may coincide). This observation implies that for any given integer  $\delta$  ( $0 \leq |\delta| \leq m/\sqrt{\log m}$ ), we have: (It is understood that  $|Z_{ij}| = 0$  if  $j < 1$  or  $j > m/\sqrt{\log m}$ .)

$$\sum_{i=1}^{m/\sqrt{\log m}} |Z_{i,i+\delta}| = O(m).$$

The lemma follows from the above simply by re-writing the summation in the lemma's statement:

$$\sum_{\delta=-m/\sqrt{\log m}}^{m/\sqrt{\log m}} \sum_{i=1}^{m/\sqrt{\log m}} |Z_{i,i+\delta}| \cdot \bullet$$

The above lemma implies that with a total of  $m^2/\log m$  processors, we can afford to assign a group of  $1 + |Z_{ij}|/\sqrt{\log m}$  processors to each pair  $X_i, Y_j$ . The task of this group is to

compute  $\theta(v, w)$  for all  $v \in X_i$  and  $w \in Y_j$  (of course each such  $\theta(v, w)$  is in  $Z_{ij}$ ). It suffices to show how such a group performs this computation in  $O(\log m)$  time. If  $|Z_{ij}| \leq \sqrt{\log m}$  then a single processor can solve the problem in  $O((\sqrt{\log m})^2) = O(\log m)$  time, by the quadratic work method of Section 3. If  $|Z_{ij}| > \sqrt{\log m}$  then we partition  $Z_{ij}$  into  $|Z_{ij}|/\sqrt{\log m}$  pieces  $J_1, J_2, \dots$  of size  $\sqrt{\log m}$  each. We assign to each  $J_k$  one processor which solves sequentially the sub-problem defined by  $X_i, J_k, Y_j$ , i.e. it computes for each  $v \in X_i$  and  $w \in Y_j$  the leftmost point of  $J_k$  through which passes a path that is shortest among the  $v$ -to- $w$  paths that are constrained to go through  $J_k$ . This sequential computation takes  $O(\log m)$  time (again, using the method of Section 3). It is done in parallel for all the  $J_k$ 's. Now we must, for each pair  $v, w$  with  $v \in X_i$  and  $w \in Y_j$ , select the best crossing point for it among the  $|Z_{ij}|/\sqrt{\log m}$  possibilities returned by each of the above-mentioned sequential computations. This involves a total (i.e. for all such  $v, w$  pairs) of  $O(|X_i||Y_j||Z_{ij}|/\sqrt{\log m}) = O(|Z_{ij}|\sqrt{\log m})$  comparisons, which can be done in  $O(\log m)$  time by the  $|Z_{ij}|/\sqrt{\log m}$  processors available (Brent's Theorem).

## 7. CRCW-PRAM algorithm

This subsection briefly sketches how the partitioning schemes of Subsection 6.2 translate into a CRCW-PRAM algorithm of time complexity  $O(\log n(\log \log m)^2)$  and processor complexity  $O(mn/\log \log m)$ . Again, it suffices to show how  $DIST_{A \cup B}$  can be obtained from  $DIST_A$  and  $DIST_B$  in  $O((\log \log m)^2)$  time and with  $m^2/\log \log m$  processors.

The procedure is recursive, and we describe it for the more general case when  $DIST_A$  is  $\ell \times h$  and  $DIST_B$  is  $h \times \ell$  (that is,  $|L_A| = |L_B| = \ell$  and the common boundary has size  $h$ ). It suffices to show that for any integer  $q \leq h$  of our choice,  $\ell h/q$  processors can, in  $O((q + \log \log h) \log \log \ell)$  time, compute  $\theta(v, w)$  for all  $v \in L_A$  and  $w \in L_B$ . If we can show this then we are done because we can choose  $q = \log \log h$ , and we have  $\ell = m$  and  $h = m/2$ .

The first stage of the computation partitions  $L_A$  into  $\sqrt{\ell}$  contiguous blocks  $X_1, X_2, \dots$  of size  $\sqrt{\ell}$  each. Similarly,  $L_B$  is partitioned into  $\sqrt{\ell}$  contiguous blocks  $Y_1, Y_2, \dots$  of size  $\sqrt{\ell}$  each. For each pair  $v, w$  such that  $v$  is an endpoint of an  $X_i$  and  $w$  is an endpoint of a  $Y_j$ , we assign  $h/q$  processors (we have enough processors because there are  $O(\ell)$  such pairs). These processors compute, in  $O(q + \log \log h)$  time, the point  $\theta(v, w)$ . Thus, if we let  $Z_{ij}$  denote the interval on the boundary common to  $A$  and  $B$  that is defined by the set  $\theta(v, w)$  such

that  $v \in X_i$  and  $w \in Y_j$ , then after this stage of the computation we know the beginning and end of each such interval  $Z_{ij}$ .

The second stage of the computation "fills in the blanks" by doing, in parallel,  $\ell$  recursive calls, one for each  $X_i, Y_j$  pair. The call for pair  $X_i, Y_j$  returns  $\theta(v, w)$  for all  $v \in X_i$  and  $w \in Y_j$  (of course each such  $\theta(v, w)$  is in  $Z_{ij}$ ).

The time and processor complexities of the above method satisfy the recurrences:

$$T(\ell, h) \leq T(\sqrt{\ell}, h) + c_1(q + \log \log h),$$

$$P(\ell, h) \leq \max\{c_2 \ell h / q, \sum_{i,j} P(\sqrt{\ell}, |Z_{ij}|)\},$$

where  $c_1$  and  $c_2$  are constants. The time recurrence implies that  $T(\ell, h) = O((q + \log \log h) \log \log \ell)$ .

That the processor recurrence implies  $P(\ell, h) = O(\ell h / q)$  becomes apparent once one observes that  $\sum_{i,j} |Z_{ij}| = O(h\sqrt{\ell})$ . The proof of this last fact is similar to that of Lemma 6:  $\sum_{i,j} |Z_{ij}|$  is re-written as  $\sum_{i,\delta} |Z_{i,i+\delta}| \leq \sum_{\delta} h = O(h\sqrt{\ell})$ . This completes the proof of the claimed CRCW-PRAM bound.

Of course the same algorithm as above yields different complexity bounds when one uses in it other CRCW-PRAM methods for computing the min of  $h$  objects. For example, one can compute the min of  $h$  objects in  $O(k)$  time using  $h^{1+2^{-k}}$  processors on a CRCW-PRAM, where  $k$  is any integer of one's choice. If such a method is used in conjunction with the above algorithm, then the algorithm runs in  $O(k \log n \log \log m)$  time with  $O(nm^{1+2^{-k}})$  processors.

## 8. Conclusion

We gave a number of PRAM algorithms for the string editing problem. The algorithms were fast and efficient, but the best *time*  $\times$  *processors* bound was still a factor of  $\log n$  away from the  $O(|x||y|)$  time complexity of the best serial algorithm for the problem.

*Acknowledgement.* The authors are grateful to the referees for their careful reading and useful comments.

## References

- [1] A. Apostolico and C. Guerra. The Longest Common Subsequence Problem Revisited, TR-CS543 Purdue Univ. (1985), also *Algorithmica* 2, pp.315-336 (1987).
- [2] A.V. Aho, D.S. Hirschberg and J.D. Ullman. Bounds on the Complexity of the Longest Common Subsequence Problem, *Journal of the ACM* 23, 1, pp.1-12 (1976).

- [3] A.V. Aho, J.E. Hopcroft and J.D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).
- [4] M.J. Atallah, R. Cole and M.T. Goodrich. Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms, *Proc. 28th Annual IEEE Symp. on Foundations of Computer Sci.*, Marina Del Rey, California, 1987, pp.151-160.
- [5] R.P. Brent. The Parallel Evaluation of General Arithmetic Expressions, *Journal of the ACM* 21, 2, pp.201-206 (1974).
- [6] R. Cole. Parallel Merge Sort, *Proc. 27th Annual IEEE Symp. on Foundations of Computer Science*, pp.511-516 (1986).
- [7] F.E. Fich, R.L. Radge and A. Widgerson. Relation Between Concurrent-Write Models of Parallel Computation, *Proc. Third Annual ACM Symp. on Distributed Computing*, pp.179-189 (1984).
- [8] Z. Galil and R. Giancarlo. Data Structures and Algorithms for Approximate String Matching, Tech. Report, CS Dept., Columbia University, NY (1987).
- [9] A.G. Ivanov. Recognition of an Approximate Occurrence of Words on a Turing Machine in Real Time, *Math. USSR Izvestiya* 24, 479-522 (1985).
- [10] R.E. Ladner and M.J. Fischer. Parallel Prefix Computation, *Journal of the ACM* 27, 4, pp.831-838 (1980).
- [11] G. Landau and U. Vishkin. Introducing Efficient Parallelism into Approximate String Matching and a new Serial Algorithm, *Proceedings of the 18-th ACM STOC*, pp.220-230 (1986).
- [12] V.I Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals, *Soviet Phys. Dokl.* 10, pp.707-710 (1966).
- [13] H. M. Martinez (ed.) Mathematical and Computational Problems in the Analysis of Molecular Sequences, *Bulletin of Mathematical Biology* (Special Issue Honoring M. O. Dayhoff), 46, 4 (1984).
- [14] W. J. Masek and M. S. Paterson. A Faster Algorithm Computing String Edit Distances, *Journal of Computer and System Science* 20, pp.18-31 (1980).
- [15] T. R. Mathies. A Fast Parallel Algorithm to Determine Edit Distance, Tech. Rept. CMU-CS-88-130 (April 1988).
- [16] S. B. Needleman and C.D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino-acid Sequence of Two Proteins, *Journal of Molecular Biology* 48, pp.443-453 (1973).
- [17] S. Ranka and S. Sahni. String Editing on an SIMD Hypercube Multicomputer, Univ. of Minnesota Computer Science Dept. Tech. Rept. 88-29, March 1988. (Submitted to *Journal of Parallel and Distributed Computing*.)
- [18] D. Sankoff. Matching Sequences Under Deletion-insertion Constraints, *Proceedings of the National Academy of Sciences of the U.S.A.* 69, pp.4-6 (1972).
- [19] D. Sankoff and J. B. Kruskal (eds.). Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison, *Addison-Wesley*, Reading, PA (1983).
- [20] P.H. Sellers. An Algorithm for the Distance between two Finite Sequences, *J. of Combinatorial Theory* 16, pp.253-258 (1974).
- [21] P.H. Sellers. The Theory and Computation of Evolutionary Distance: Pattern Recognition, *Journal of Algorithms* 1, pp.359-373 (1980).
- [22] Y. Shiloach and U. Vishkin. Finding the Maximum, Merging and Sorting in a Parallel Model of Computation, *Journal of Algorithms* 2, pp.88-102 (1981).

- [23] E. Ukkonen. Finding Approximate Patterns in Strings, *Journal of Algorithms* 6, pp.132-137 (1985).
- [24] L. Valiant. Parallelism in Comparison Problems, *SIAM J. on Computing* 4, 3, pp.348-355 (1975).
- [25] R. A. Wagner and M. J. Fischer. The String to String Correction Problem, *Journal of the ACM* 21,1, pp.168-173 (1974).
- [26] C.K. Wong and A.K. Chandra. Bounds for the String Editing Problem, *Journal of the ACM* 23, 1, pp.13-16 (1976).