Algorithmica (1997) 19: 291–317

Algorithmica

Parallel Searching in Generalized Monge Arrays

A. Aggarwal,[1] D. Kravets,[2] J. K. Park,[3] and S. Sen[4]

Abstract. This paper investigates the parallel time and processor complexities of several searching problems involving *Monge*, *staircase-Monge*, and *Monge-composite* arrays. We present array-searching algorithms for concurrent-read-exclusive-write (CREW) PRAMs, hypercubes, and several hypercubic networks. All these algorithms run in near-optimal time, and their processor-time products are all within an $O(\lg n)$ factor of the worst-case sequential bounds. Several applications of these algorithms are also given. Two applications improve previous results substantially, and the others provide novel parallel algorithms for problems not previously considered.

Key Words. Monge arrays, CREW-PRAM algorithms, Hypercubes.

1. Introduction

1.1. *Background.* An $m \times n$ array $A = \{a[i, j]\}$ containing real numbers is called *Monge* if, for $1 \leq i < k \leq m$ and $1 \leq j < l \leq n$,

$$a[i, j] + a[k, l] \leq a[i, l] + a[k, j]. \tag{1.1}$$

We refer to (1.1) as the *Monge condition*. Monge arrays have many applications. In the late eighteenth century, Monge [34] observed that if unit quantities (cannonballs, for example) need to be transported from locations X and Y (supply depots) in the plane to locations Z and W (artillery batteries), not necessarily respectively, in such a way as to minimize the total distance traveled, then the paths followed in transporting these quantities must not properly intersect. In 1961, Hoffman [24] elaborated upon this idea and showed that a greedy algorithm correctly solves the transportation problem for m sources and n sinks if and only if the corresponding $m \times n$ cost array is a Monge array.

[1] IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598, USA. aggarwa@watson.ibm.com.

[2] Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102, USA. dina@cis.njit.edu. This author's research was supported by the NSF Research Initiation Award CCR-9308204, and the New Jersey Institute of Technology SBR Grant #421220. Part of this research was done while the author was at MIT and supported by the Air Force under Contract AFOSR-89-0271 and by the Defense Advanced Research Projects Agency under Contracts N00014-87-K-825 and N00014-89-J-1988.

[3] Bremer Associates, Inc., 215 First Street, Cambridge, MA 02142, USA. james.park@bremer-inc.com. This author's work was supported in part by the Defense Advanced Research Projects Agency under Contract N00014-87-K-0825 and the Office of Naval Research under Contract N00014-86-K-0593 (while the author was a graduate student at MIT) and by the Department of Energy under Contract DE-AC04-76DP00789 (while the author was a member of the Algorithms and Discrete Mathematics Department of Sandia National Laboratories).

[4] Department of Computer Science, Indian Institute of Technology, New Delhi, India. ssen@cse.iitd.ernet.in. Part of the work was done when the author was a summer visitor at IBM T. J. Watson Research Center.

Received July 25, 1994; revised March 5, 1996. Communicated by B. Chazelle.

More recently, Monge arrays have found applications in a many other areas. Yao [37] used these arrays to explain Knuth's [28] efficient sequential algorithm for computing optimal binary trees. Aggarwal *et al*. [4] showed that the all-farthest-neighbors problem for the vertices of a convex n-gon can be solved in linear time using Monge arrays. Aggarwal and Park [6] gave efficient sequential algorithms based on the Monge-array abstraction for several problems in computational geometry and VLSI river routing. Furthermore, many researchers [6], [31], [21], [22] have used Monge arrays to obtain efficient dynamic programming algorithms for problems related to molecular biology. More recently, Aggarwal and Park [9] have used Monge arrays to obtain efficient algorithms for the economic-lot size model.

In many applications, the underlying array satisfies conditions that are similar but not the same as in (1.1). An $m \times n$ array A is called *inverse-Monge* if, for $1 \leq i < k \leq m$ and $1 \leq j < l \leq n$,

$$a[i, j] + a[k, l] \geq a[i, l] + a[k, j].^{5} \tag{1.2}$$

An $m \times n$ array $S = \{s[i, j]\}$ is called *staircase-Monge* if

(i) every entry is either a real number or ∞,
(ii) $s[i, j] = \infty$ implies $s[i, \ell] = \infty$ for $\ell > j$ and $s[k, j] = \infty$ for $k > i$, and
(iii) for $1 \leq i < k \leq m$ and $1 \leq j < \ell \leq n$, (1.1) holds if all four entries $s[i, j]$, $s[i, \ell]$, $s[k, j]$, and $s[k, \ell]$ are finite.

The definition of a *staircase-inverse-Monge* array is similar:

(i) every entry is either a real number or ∞,
(ii) $s[i, j] = \infty$ implies $s[i, \ell] = \infty$ for $\ell < j$ and $s[k, j] = \infty$ for $k > i$, and
(iii) for $1 \leq i < k \leq m$ and $1 \leq j < \ell \leq n$, (1.2) holds if all four entries $s[i, j]$, $s[i, \ell]$, $s[k, j]$, and $s[k, \ell]$ are finite.

Observe that a Monge array is a special case of a staircase-Monge array. Finally, a $p \times q \times r$ array $C = \{c[i, j, k]\}$ is called *Monge-composite* if $c[i, j, k] = d[i, j] + e[j, k]$ for all i, j, and k, where $D = \{d[i, j]\}$ is a $p \times q$ Monge array and $E = \{e[j, k]\}$ is a $q \times r$ Monge array.

Like Monge arrays, staircase-Monge arrays have also found applications in many areas. Aggarwal and Park [6], Larmore and Schieber [31], and Eppstein *et al*. [21], [22] use staircase-Monge arrays to obtain algorithms for problems related to molecular biology. Aggarwal and Suri [10] used these arrays to obtain fast sequential algorithms for computing the following largest-area empty rectangle problem: given a rectangle containing n points, find the largest-area rectangle that lies inside the given rectangle, that does not contain any points in its interior, and whose sides are parallel to those of the given rectangle. Furthermore, Aggarwal and Klawe [3] and Klawe and Kleitman [27] have demonstrated other applications of staircase-Monge arrays in computational geometry.

Finally, both Monge and Monge-composite arrays have found applications in parallel computation. In particular, Aggarwal and Park [5] exploit Monge arrays to obtain efficient CRCW- and CREW-PRAM algorithms for certain geometric problems, and they exploit Monge-composite arrays to obtain efficient CRCW- and CREW-PRAM algorithms for

[5] We refer to (1.2) as the *inverse-Monge condition*.

string editing and other related problems. (See also [12].) Similarly, Atallah *et al.* [15] have used Monge-composite arrays to construct Huffman and other such codes on CRCW and CREW PRAMs. Larmore and Przytycka in [30] used Monge arrays to solve the *Concave Least Weight Subsequence* (*CLWS*) *problem* (defined in Section 4.2).

Unlike Monge and Monge-composite arrays, staircase-Monge arrays have not been studied in a parallel setting (in spite of their immense utility). Furthermore, even for Monge and Monge-composite arrays, the study of parallel array-search algorithms has so far been restricted to CRCW and CREW PRAMs. In this paper we fill in these gaps by providing efficient parallel algorithms for searching in Monge, staircase-Monge, and Monge-composite arrays. We develop algorithms for the CREW-PRAM models of parallel computation, as well as for several interconnection networks including the hypercube, the cube-connected cycles, the butterfly, and the shuffle-exchange network. Before we can describe our results, we need a few definitions which we give in the next section.

1.2. *Definitions.* In this section we explain the specific searching problems we solve and give the previously known results for these problems. The row-minima problem for a two-dimensional array is that of finding the minima entry in each row of the array. (If a row has several minima, then we take the leftmost one.) In dealing with Monge arrays we assume that for any given i and j, a processor can compute the (i, j)th entry of this array in $O(1)$ time. For parallel machines without global memory we need to use a more restrictive model. The details of this model are given in later sections. Aggarwal *et al.* [4] showed that the row-minima problem for an $m \times n$ Monge array can be solved in $O(m + n)$ time, which is optimal. Also, Aggarwal and Park [5] have shown that the row-minima problem for such an array can be solved in $O(\lg mn)$ time on an $(m + n)$-processor CRCW PRAM, and in $O(\lg mn \lg\lg mn)$ time on an $((m + n)/\lg\lg mn)$-processor CREW PRAM. Atallah and Kosaraju in [14] improved this to $O(\lg mn)$ using $m + n$ processors on a (weaker) EREW PRAM. Note that all the algorithms dealing with finding row-minima in Monge and inverse-Monge arrays can also be used to solve the analogously defined row-maxima problem for the same arrays. In particular, if $A = \{a[i, j]\}$ is an $m \times n$ Monge (resp. inverse-Monge) array, then $A' = \{a'[i, j] : a'[i, j] = -a[i, n - j + 1]\}$ is a $m \times n$ Monge (resp. inverse-Monge) array. Thus, solving the row-minima problem for A' gives us row-maxima for A.

Unfortunately, the row-minima and row-maxima problems are not interchangeable when dealing with staircase-Monge and staircase-inverse-Monge arrays. Aggarwal and Klawe [3] showed that the row-minima problem for an $m \times n$ staircase-Monge array can be solved in $O((m+n)\lg\lg(m+n))$ sequential time, and Klawe and Kleitman [27] have improved the time bound to $O(m + n\alpha(m))$, where $\alpha(\cdot)$ is the inverse of Ackermann's function. However, if we wanted to solve the row-maxima problem (instead of the row-minima problem) for an $m \times n$ staircase-Monge array, then we could, in fact, employ the sequential algorithm given in [4] and solve the row-maxima problem in $O(m + n)$ time. No parallel algorithms were known for solving the row-minima problem for staircase-Monge arrays.

Given a $p \times q \times r$ Monge-composite array, for $1 \le i \le p$ and $1 \le k \le r$, the (i, k)th tube consists of all those entries of the array whose first coordinate is i and whose third coordinate is k. The tube-minima problem for a $p \times q \times r$ Monge-composite array

is that of finding the minimum entry in each tube of the array. (If a tube has several minima, then we take the one with the minimum second coordinate.) For sequential computation, the result of [4] can be trivially used to solve the tube-minima problem in $O((p+r)q)$ time. Aggarwal and Park [5] and Apostolico *et al.* [12] have independently shown that the tube-minima problem for an $n \times n \times n$ Monge-composite array can be solved in $O(\lg n)$ time using $n^2/\lg n$ processors on a CREW PRAM, and, recently, Atallah [13] has shown that this tube-minima problem can be solved in $O(\lg \lg n)$ time using $n^2/\lg \lg n$ processors on a CRCW PRAM. Both results are optimal with respect to time and processor-time product. In view of the applications, we assume that the two $n \times n$ Monge arrays $D = \{d[i, j]\}$ and $E = \{e[j, k]\}$, that together form the Monge-composite array, are stored in the global memory of the PRAM. Again, for parallel machines without a global memory, we need to use a more restrictive model; the details of this model are given later. No efficient algorithms (other than the one that simulates the CRCW-PRAM algorithm) were known for solving the tube-minima problem for a hypercube or a shuffle-exchange network.

1.3. *Our Main Results.* The time and processor complexities of algorithms for computing row minima in two-dimensional Monge, row minima in two-dimensional staircase-Monge arrays, and tube minima in three-dimensional Monge-composite arrays are listed in Tables 1.1, 1.2, and 1.3, respectively. We assume a *normal* model of hypercube computation, in which each processor uses only one of its edges in a single time step, only one dimension of edges is used at any given time step, and the dimension used at time step $t + 1$ is within 1 module d of the dimension used at time step t, where d is the dimension of the hypercube (see Section 3.1.3 of [32]). It is known that such algorithms for the hypercube can be implemented on other hypercubic bounded-degree networks like Butterfly and shuffle-exchange without asymptotic slow-down. Observe that our results for staircase-Monge arrays match the corresponding bounds for Monge arrays.

Following are some applications of these new array-searching algorithms.

1. *All Pairs Shortest Path* (*APSP*) *Problem.* Consider the following problem: given a weighted directed graph $G = (V, E)$, $|V| = n$, $|E| = m$, we want to find the shortest path between every pair of vertices in V. In the sequential case, Johnson [26] gave an $O(n^2 \lg n + mn)$-time algorithm for APSP. In the parallel case, APSP can be solved by repeated squaring in $O(\lg^2 n)$ time using $n^3/\lg n$ processors on a CREW PRAM. Atallah *et al.* [15] show how to solve APSP in $O(\lg^2 n)$ time using $n^3/\lg n$ processors on a CREW PRAM (this solution follows from their $O(\lg^2 n)$-time $(n^2/\lg n)$-processor solution to the single source shortest paths problem on such a graph). In Section 4.1 we give the algorithm of Aggarwal *et al.* [2] which runs in $O(\lg^2 n)$ time using n^2 CREW-PRAM

Table 1.1. Row-minima results for an $n \times n$ Monge array.

Model	Time	Processors	Reference
CREW PRAM	$O(\lg n)$	n	[14]
Hypercube	$O(\lg n \lg \lg n)$	n	Theorem 3.2

Table 1.2. Row-minima results for an $n \times n$ staircase-Monge array.

Model	Time	Processors	Reference
CREW PRAM	$O(\lg n)$	n	Theorem 2.3
Hypercube	$O(\lg n \lg\lg n)$	n	Theorem 3.4

processors for the special case of the APSP problem when the graph is acyclic and the edge weights satisfy the quadrangle inequality.[6]

2. *Huffman Coding Problem.* Consider the following problem: given an alphabet $\mathcal{C}$ of n characters and the function f_i indicating the frequency of character $c_i \in \mathcal{C}$ in a file, construct a prefix code which minimizes the number of bits needed to encode the file, i.e., construct a binary tree T such that each leaf corresponds to a character in the alphabet and the weight of the tree, $\mathcal{W}(T)$, is minimized, where

$$\mathcal{W}(T) = \sum_{i=1}^{n} f_i d_i, \tag{1.3}$$

and d_i is the depth in T of the leaf corresponding to character c_i. The weight of the tree $\mathcal{W}(T)$ is exactly the minimum number of bits needed to encode the file (see [18]). The construction of such an optimal code (which is called a Huffman code) is a classical problem in data compression. In the sequential domain, Huffman in [25] showed how to construct Huffman codes greedily in $O(n)$ time (once the character frequencies are in sorted order). In [15], Atallah *et al.* reduced Huffman coding to $O(\lg n)$ tube minimization problems on Monge-composite arrays, thereby obtaining parallel algorithms for Huffman coding that run in $O(\lg^2 n)$ time using $n^2/\lg n$ processors on a CREW PRAM and in $O(\lg n(\lg\lg n)^2)$ time using $n^2/(\lg\lg n)^2$ processors on a CRCW PRAM. Larmore and Przytycka in [30] reduce Huffman coding to the *Concave Least Weight Subsequence* (*CLWS*) *problem* (defined in Section 4.2) and then show how to solve CLWS, and thereby Huffman coding, in $O(\sqrt{n}\lg n)$ time using n processors on a CREW PRAM. Theirs is the first known parallel algorithm for Huffman coding requiring $o(n^2)$ work. In Section 4.2 we present the result of Czumaj [20] for finding the Huffman code in $O(\lg^{r+1} n)$ time and a total of $O(n^2 \lg^{2-r} n)$ work on a CREW PRAM, for any $r \geq 1$. This is the first NC algorithm that achieves $o(n^2)$ work.

Table 1.3. Tube-minima results for an $n \times n \times n$ Monge-composite array.

Model	Time	Processors	Reference
CREW PRAM	$O(\lg n)$	$n^2/\lg n$	[5], [12]
Hypercube	$O(\lg n)$	n^2	Theorem 3.5

[6] Given an ordering of the vertices of a graph, *the quadrangle inequality* states that any four distinct vertices appearing in increasing order in that ordering, i_1, i_2, j_1, and j_2, must satisfy $d(i_1, j_1) + d(i_2, j_2) \geq d(i_1, j_2) + d(i_2, j_1)$. In other words, in the quadrangle formed by $i_1 i_2 j_1 j_2$, the sum of the diagonals is greater than the sum of the sides. Notice that this condition is the same as (1.2) and they both appear in the literature.

3. *The String Editing Problem and Other Related Problems.* Consider the following problem: given two input strings $x = x_1x_2 \cdots x_m$ and $y = y_1y_2 \cdots y_n$, $m = |x|$ and $n = |y|$, find a sequence of *edit operations* transforming x to y, such that the sum of the individual edit operations' costs is minimized. Three different types of edit operations are allowed: we can delete the symbol x_i at cost $D(x_i)$, insert the symbol y_j at cost $I(y_j)$, or substitute the symbol x_i for the symbol y_j at cost $S(x_i, y_j)$. In [36], Wagner and Fischer gave an $O(mn)$-time sequential algorithm for this problem. PRAM algorithms for this problem were provided in [5] and [12]; these algorithms reduce the string editing problem to a shortest-path problem in a special kind of directed graph called a *grid-DAG* and use array-searching to solve this shortest-path problem. Using our tube-minima algorithms for hypercubes and related networks, in Section 4.3 we solve the string editing problem in $O(\lg m \lg n)$ time on an mn-processor hypercube. Our result significantly improves the results of Ranka and Sahni [35], who give two SIMD hypercube algorithms for the $m = n$ special case of the string editing problem: one algorithm runs in $O(\sqrt{(n^3 \lg n)/p} + \lg^2 n)$ time using p processors, $n^2 \leq p \leq n^3$; the other algorithm runs in $O(\sqrt{(n^3 \lg n)/p})$ time using p processors, $n \lg n \leq p \leq n^2$.

4. *The Largest-Area* (*Not Necessarily Empty*) *Rectangle* (*LAR*) *Problem.* Consider the following problem: given a set of n planar points, compute the largest-area rectangle that is formed by taking any two of the n points as the rectangle's opposite corners and whose sides are parallel to the x- and y-axes. For this problem, we obtain (in Section 4.4) a CREW-PRAM algorithm that takes $O(\lg n)$ time and uses n processors. This geometric problem is motivated by the following problem in electronic circuit simulation and has been recently studied by Melville [33]. Imagine an integrated circuit containing n nodes. Because of the nature of integrated circuit fabrication, there will be leakage paths between all pairs of nodes. For which pair of nodes is a leakage path (between those nodes) most detrimental to circuit performance? In [33], Melville argues that this pair of nodes correspond to the pair forming the largest-area rectangle.

5. *The Nearest-Visible-, Nearest-Invisible-, Farthest-Visible-, and Farthest-Invisible-Neighbors Problems for Convex Polygons.* Consider the following problem which we call the nearest-visible-neighbor (nearest-invisible-neighbor) problem: given two nonintersecting convex polygons P and Q, determine for each vertex x of P, the vertex of Q nearest to x that is visible (resp. not visible) to x. If P and Q contain m and n vertices, respectively, then the nearest-visible-neighbor problem can be solved optimally in $O(\lg(m + n))$ time using $((m + n)/\lg(m + n))$ processors on a CREW PRAM. Furthermore, we can use the row-minima algorithm developed for staircase-Monge arrays to show in Section 4.5 that the nearest-invisible-neighbor problem can be solved in $O(\lg(m + n))$ time on a CREW PRAM with $m + n$ processors. The farthest-visible-neighbor (resp. farthest-invisible-neighbor) problem for P and Q can be defined similarly, and it can be solved in the same time and processor bounds as the nearest-visible-neighbor (resp. nearest-invisible-neighbor) problem.

The remainder of this paper is organized as follows. In Section 2 we give the PRAM algorithms for finding row minima in staircase-Monge arrays. In Section 3 we give the hypercube algorithms for finding row minima in Monge and staircase-Monge arrays

and tube minima in Monge-composite arrays. Details of the applications are given in Section 4.

2. CREW-PRAM Algorithms to Compute Row Minima in Staircase-Monge Arrays. In this section we give CREW-PRAM algorithms for computing row minima in staircase-Monge arrays. We use the CREW-PRAM algorithms for computing row minima in Monge arrays summarized in Table 1.1. In [3], Aggarwal and Klawe gave an $O((m+n)\lg\lg(m+n))$-time sequential algorithm for finding the row minima of an $m \times n$ staircase-Monge array. This was subsequently improved to $O(m+n\alpha(m))$ time by Klawe and Kleitman [27]. In the discussion below we parallelize Aggarwal and Klawe's sequential algorithm [3] using the techniques developed in [5].

Let $A = \{a[i, j]\}$ be an $m \times n$ staircase-Monge array, $m \geq n$. The basic idea is first to compute the minimum entry in (approximately) every (m/n)th row of A. Then we use the location of the minima just computed, together with the structure of a staircase-Monge array, to limit those entries that need to be considered for the minima in the remaining rows. For $1 \leq i \leq m$, let $f[i]$ be the smallest index such that $a[i, f[i]] = \infty$. Let R_i denote the (is)th row of the array, where $s = \lfloor m/n \rfloor$, and let R_i^t denote the row obtained by changing the jth column entry of R_i to an ∞ for each j with $f[(i+1)s] \leq j < f[is]$. Furthermore, let A^t denote the $n \times n$ array consisting of the rows R_i^t. Clearly, A^t is a staircase-Monge array. To simplify the proofs, we augment A and A^t with row 0 where $a[0, j] = j$ for $1 \leq j \leq n$. Let $f[0] = n+1$. Note that this addition is consistent with Mongeness of A and A^t. We prove the following lemma.

LEMMA 2.1. *Given the row minima of A^t, we can compute the row minima of A in $O(\lg m + \lg n)$ time using $m/\lg m + n/\lg n$ processors on a CREW PRAM.*

PROOF. Look at the positions of the row minima of A^t carefully. Let μ_i be the minimum in row R_i^t. Figure 2.1 shows array A with row R_i replaced by R_i^t, for $1 \leq i \leq n$. Note that the array as shown is not Monge. Nevertheless, we will be able to use the minima of A^t within the Monge areas of this array to narrow down our search space for row minima. From [3], the minima of A^t induce a partitioning of A such that certain regions can be omitted from further searching for row minima because of the Monge condition. The feasible regions (for row minima) can be categorized into two classes: Monge arrays (the F_i's) and staircase-Monge arrays (the P_i's). Then the minimum in a row of A is either the row minimum in a P_i region, or the row minimum in an F_i region, or is among the elements of $R_i \setminus R_i^t$.

We first deal with the feasible staircase-Monge arrays. Because we substituted row R_i^t for row R_i, row minima of A^t tells us nothing about the regions in the Figure 2.1 which are labeled as P_i's. Formally, P_i consists of a subarray of A given by rows $(i-1)s+1$ through $is-1$ and columns $f[is]$ through $f[(i-1)s-1]$, for $1 \leq i \leq n$. The total number of elements in all the P_i's is $(n+1)\lfloor m/n \rfloor = O(m)$. We use a brute-force search of these elements to find the row minima.

The only regions left for us to consider are the feasible Monge arrays.

PROPOSITION 2.2. *There are at most $2n+1$ feasible Monge arrays.*

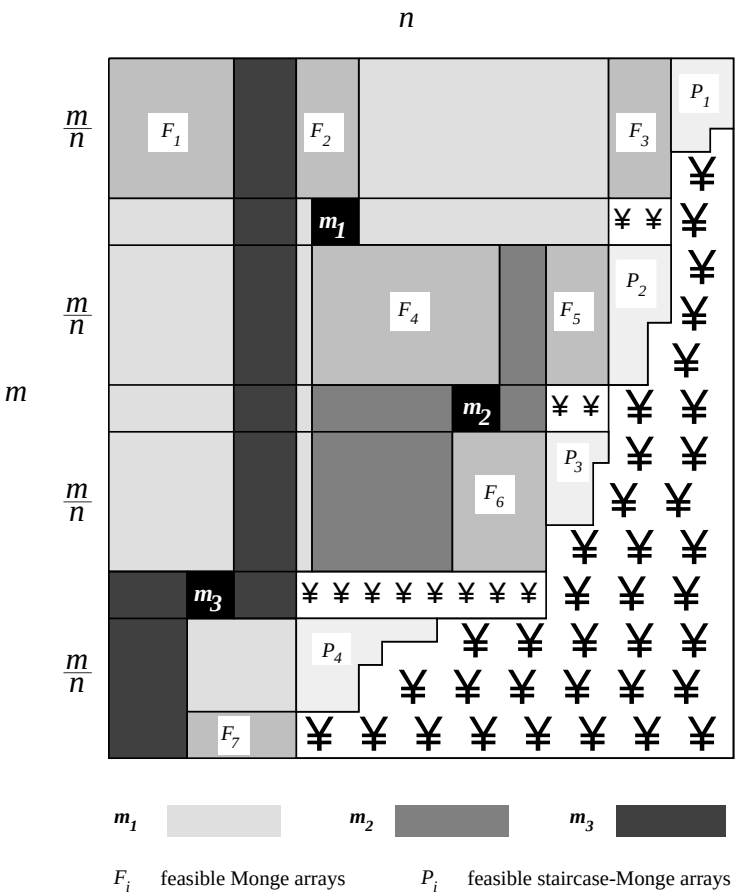

Fig. 2.1. Array A with row R_i replaced by R_i^t. Each minimum μ_i in row R_i eliminates certain regions of A from consideration for row minima. An infeasible region is covered by the pattern of the μ_i that made it infeasible. Many of the regions are eliminated by more than one μ_i in this case, we show arbitrarily one such pattern.

PROOF. If the minimum of R_{i+1}^t lies to the left of the minimum of R_i^t, then there is at most one feasible Monge region (F_6 in Figure 2.1) where the minima of the rows in A between R_i^t and R_{i+1}^t can lie. However, if the minimum of R_i^t lies to the left of the minimum of R_{i+1}^t, then there can be more than one feasible Monge region where the minima can lie (e.g., F_4 and F_5). We claim that the number of extra feasible Monge regions is equal to the number of minima which are "bracketed" by the minimum of R_i^t. We define "bracketed" as follows. Minimum μ is said to bracket another minimum $\hat{\mu}$ if μ is the closest northwest neighbor of $\hat{\mu}$, i.e., μ lies above and to the left of $\hat{\mu}$, and among all the minima which have this property with respect to $\hat{\mu}$, the row of μ is the maximum. Intuitively, a minimum μ of R_i^t leaves the region below and to the right of it, which we call L, as potentially feasible for row minima. If there is a minimum $\hat{\mu}$ in row R_{i+k}^t bracketed by μ, then $\hat{\mu}$ eliminates the region to the right (up to column $f[s(i+k+1)]$) and above $\hat{\mu}$ from being considered for row minima. Thus, in effect,

$\hat{\mu}$ splits L into two regions, one to the right of $\hat{\mu}$ and one to the left of $f[s(i+k+1)]$. In Figure 2.1, μ_2 is bracketed by μ_1, adding the region F_5. Recall that A^t is augmented with row 0 in which the minimum μ_0 is in column 1 (not shown). Thus, both μ_1 and μ_3 are bracketed by μ_0, adding regions F_3 and F_2, respectively. Note that μ_3 adds only the region F_2 (as opposed to the entire region left and above $f[3s]$) since other minima have carved the larger region into smaller feasible/infeasible blocks. Since each minimum can be bracketed at most once, the total number of minima that are bracketed is at most n. Thus, the total number of feasible Monge regions is $2n+1$. □

Note that all the F_i's have nonoverlapping columns (except possibly for the columns in which the minima of A^t occur) and have s rows. Therefore, the total number of elements in all the feasible Monge arrays is $n\lfloor m/n \rfloor + m = O(m)$. Since all the feasible Monge regions contain $O(m)$ elements, we again use a brute-force search to find the row minima, provided that we can find all the F_i's efficiently.

We determine the F_i's as follows. From Proposition 2.2, we know that there is exactly one feasible Monge region in the rows between R_i^t and R_{i+1}^t if μ_{i+1} is to the left of μ_i. We find all such regions. Next, we find all the bracketed minima. To do this we form a list $L = \langle \ell_0, \ell_2, \ldots, \ell_s \rangle$ such that ℓ_i is the column of the minimum of R_i^t. Minimum μ_i brackets minimum μ_j if $i < j$ and $\ell_i < \ell_j$. In [16], Berkman *et al.* define *the All Nearest Smallest Value* (*ANSV*) *problem* as follows: given a list $W = (w_1, w_2, \ldots, w_n)$ of elements from a totally ordered domain, determine for each w_i, $1 \le i \le n$, the nearest element to its left in the list and the nearest element to its right in the list that are less than w_i (if they exist). They show how to solve ANSV in $O(\lg n)$ time using $n/\lg n$ processors on a CREW PRAM. Thus, an application of their ANSV algorithm gives us all the bracketed minima. Suppose minimum μ_{i_1} brackets minimum $\mu_{i_2}, \mu_{i_3}, \ldots, \mu_{i_k}$, $i_1 < i_2 < \cdots < i_k$. Then these minima create k regions in rows $i_1 s + 1$ through $(i_1+1)s - 1$. The first region is columns *column*(μ_{i_1}) through *column*(μ_{i_2}), the last region is columns $f[s(i_2+1)]$ through $f[si_2 - 1]$, and the jth region, $1 < j < k$, is columns $f[s(i_{k-j+2}+1)]$ through *column*$(\mu_{(i_{k-j+1})})$. This gives us all the F_i's.

Finally, because we have changed certain entries of the R_i's to ∞, we need to reconsider the minima we have for these rows. Since there were no more than n entries of A that were changed to ∞ in producing A^t, we can find the minima in these rows by brute-force search. Combining these row minima with the row minima we get from the F_i's and the P_i's, we can easily determine the row minima of A. For the complexity analysis, notice that we used the algorithm of Berkman *et al.* [16] and the brute-force search for a minimum among n and m elements. Both of these procedures can be done in $O(\lg mn)$ time using $m/\lg m + n/\lg n$ processors on a CREW PRAM. □

Given this lemma,we can prove the following result.

THEOREM 2.3. *The row minima of an $n \times n$ staircase-Monge array can be computed in $O(\lg n)$ time using n processors on a CREW PRAM.*

PROOF. We use an approach very similar to Aggarwal and Park's [5]. Given the $n \times n$ staircase-Monge array B, define $f[i]$, R_i's, R_i^t's, and B^t as before, except that $s = \lfloor \sqrt{n} \rfloor$. Let $u = \lceil n/\sqrt{n} \rceil$. B^t is a $u \times n$ staircase-Monge array with at most u "steps." Thus,

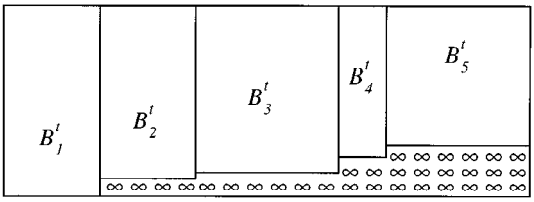

Fig. 2.2. Decomposition of B^t into $B_1^t, \ldots, B_u^t$.

B^t can be decomposed into at most u Monge arrays $B_1^t, \ldots, B_u^t$, such that each B_i^t is a $u_i \times v_i$ array, for $u_i \leq u$ and some $v_i > 0$ (see Figure 2.2). Atallah and Kosaraju [14] show how to find the row minima for an $m \times n$ Monge array in $O(\lg mn)$ time using $m+n$ processors. Thus, using their result, the row minima for all the B_i^t's can be computed in $O(\lg n)$ time using

$$\sum_{i=1}^{u} \left(\frac{u_i}{\lg u_i} + v_i \right) \leq \sum_{i=1}^{u} \left(\frac{\sqrt{n}}{\lg \sqrt{n}} + v_i \right) \leq 2n$$

processors.

The minima of B^t induces a partition of the array B, similar to that of Figure 2.1. We first determine the minima in all the feasible Monge arrays. From the proof of Lemma 2.1, we know that there are at most $2u + 1$ feasible Monge arrays and that these arrays have nonoverlapping columns, except for the columns in which the minima of B^t occur. Using the algorithm of Berkman *et al.* [16] we can find these arrays in $O(\lg n)$ time using n processors. Note that, unlike in the proof of Lemma 2.1, we cannot use brute force to find the row minima in the feasible Monge arrays since the total number of elements in all the feasible Monge arrays is $ns + n = O(n\sqrt{n})$. Instead, we use [14] to find the row minima in all the feasible Monge arrays. This can be done in $O(\lg n)$ time using at most

$$\sum_{i=1}^{2u+1} \left(\frac{s}{\lg s} + w_i \right) \leq 2n.$$

For the feasible staircase-Monge regions, we call the algorithm recursively by subdividing the arrays into $s \times s$ pieces. For the arrays which have less than s columns we use the scheme of Aggarwal and Park [5] and Lemma 2.1 to bound the number of processors to $O(n)$ processors. To find the minimum of every row, we choose the minimum of the minimum elements of the Monge arrays and the staircase-Monge array.

The complexity of all the nonrecursive procedures in this proof is dominated by the use of [14] to compute the B_i^t's. Thus, the complexities are as claimed:

$$\textit{Time} = T(n) = T(\sqrt{n}) + O(\lg n) = O(\lg n),$$

$$\textit{Processors} = P(n) = \max\{n, \sqrt{n}P(\sqrt{n})\} = n,$$

where $T(1) = 1$ and $P(1) = 1$. □

COROLLARY 2.4. *The row minima of an $m \times n$ staircase-Monge array can be computed in $O(\lg mn)$ time using $m/\lg m + n$ processors on a CREW PRAM.*

PROOF. The proof follows on the lines of [14]. Let B be an $m \times n$ staircase-Monge array. The case corresponding to $m \leq n$ is easy. Partition the array into $\lceil n/m \rceil$ arrays of size $m \times m$. Compute the row minima in $O(\lg m)$ time using n processors. Then compute the minimum in each row from the $\lceil n/m \rceil$ elements in $O(\lg n)$ time using $n/\lg n$ processors. For the case $m \geq n$, we compute the minima of an $n \times n$ array B^t in $O(\lg n)$ time using n processors (Theorem 2.3) and then use a scheme similar to Lemma 2.1 to compute the row minima of B in $O(\lg mn)$ time using $m/\lg m + n/\lg n$ processors on a CREW PRAM. □

3. Algorithms for Hypercubes and Related Networks. In this section we give three hypercube algorithms for searching in Monge arrays. The first algorithm computes the row minima of two-dimensional Monge arrays, the second computes the row minima of two-dimensional staircase-Monge arrays, and the third computes the tube minima of three-dimensional Monge arrays. These algorithms can be adapted for several hypercubic networks.

3.1. *Preliminaries.* Our hypercube algorithms are based on the corresponding CREW-PRAM algorithms of Aggarwal and Park [5], Apostolico *et al.* [12], and Atallah and Kosaraju [14]. However, there are three important issues that need to be addressed in converting from CREW-PRAM algorithms to hypercube algorithms:

(i) We can no longer use Brent's theorem [17] which converts a P-processor algorithm that runs in time T and performs a total of W operations on a CREW PRAM into a (W/T)-processor algorithm that runs in time $O(T)$ on a CREW PRAM. (This theorem is used in [5] to get the results given in Table 1.1.)
(ii) We must deal more carefully with the issue of processor allocation, especially in recursing on problems of uneven sizes.
(iii) We need to consider the data movement through the hypercube.

This last issue requires a bit more explanation. Since the hypercube lacks a global memory, our assumption that any entry of the Monge, staircase-Monge, or Monge-composite array in question can be computed in constant time by any processor is no longer valid, at least in the context of our applications. We instead use the following model. In the case of two-dimensional Monge and staircase-Monge arrays $A = \{a[i, j]\}$, we assume there are two vectors $g[1], \ldots, g[m]$ and $h[1], \ldots, h[n]$ such that a processor needs to know both $g[i]$ and $h[j]$ to compute $a[i, j]$ in constant time. Similarly, in the case of Monge-composite arrays $C = \{c[i, j, k]\}$, where $c[i, j, k] = d[i, j] + e[j, k]$, and $D = \{d[i, j]\}$ and $E = \{e[j, k]\}$ are Monge arrays, we assume that a processor needs to know both $d[i, j]$ and $e[j, k]$ to compute $c[i, j, k]$. The manner in which the $g[i]$, $h[j]$, $d[i, j]$, and $e[j, k]$ are distributed through the hypercube is then an important consideration. We assume that initially the entries of g and h (or of D and E) are

uniformly distributed in the obvious way among the local memories of the hypercube's processors.

We use the *normal* model of hypercube computation (defined in the Introduction). Moreover, all the processors use the edges corresponding to each of the $O(\log N)$ dimensions in a cyclic order in consecutive time steps. This is in contrast to the the multiport model of the hypercube in which all the edges of the hypercube may be used during a single step of the algorithm, i.e., each processor of an N-processor hypercube can send and receive $\lg N$ messages in a single time step. The advantage of the weaker model is in the greater adaptability of its algorithms in other bounded-degree models like Butterfly networks and Shuffle-exchange networks (without asymptotic slowdown with the same number of processors). However, for some of our algorithms the multiport model can achieve the same timebound by using an $O(\log N)$ factor less processors. In this section hypercube refers to the *normal* model unless mentioned otherwise.

Each processor of an N-processor hypercube has a unique index $1, \ldots, N$. In our proofs, we use algorithms for the following problems:

(i) parallel prefix,
(ii) merging two sorted lists,
(iii) monotone routing, and
(iv) routing a fixed permutation.

We specify when we use segmented parallel prefix, a standard variation of the parallel prefix. When unclear from the context, we give the associative operation performed by the parallel prefix. A *monotone routing* problem is that of routing packets such that the relative order of the packets is unchanged. Formally, if we want to route packets $u_1, u_2, \ldots, u_j$ $(j < N)$, the packet u_i originates at the processor indexed $orig(i)$, $orig(1) < orig(2) < \cdots < orig(j)$, and is destined for the processor indexed $dest(i)$, then the routing is monotone if and only if $dest(1) < dest(2) < \cdots < dest(j)$. If the input consists of N elements, then all four of the above problems can be solved in a pipelined fashion on an N-processor butterfly in $O(\lg N)$ time. The reader åis referred to Leighton's book [32] for detailed descriptions of the hypercubic networks and these algorithms.

Finally, we need an algorithm for a special case of a one-to-many routing problem. Suppose we have an $s \times t$ array on an $(N = 2^{\lceil \lg_2 st \rceil})$-processor hypercube such that processor indexed k, $k \leq st$, is responsible for the entries in row $\lceil k/t \rceil$ and column $k \bmod t$. Processors 1 through t contain values u_1 through u_t. A *row-copy problem* is that of copying the values contained in the first-row processors down the columns so that all the processors responsible for column j get the value u_j. Specifically, processor k, $k \leq t$, needs to distribute u_k to processors $k + t,\ k + 2t, \ldots, k + (s-1)t$. Notice that this operation is not monotone routing: processor $k+1$ needs to distribute value u_{k+1} to processors $k+1+t, k+1+2t, \ldots, k+1+(s-1)t$ and, clearly, $k + 2t \not\leq k + 1 + t$ in the general case. To solve the row-copy problem efficiently we exploit the fact that a hypercube of size (number of processors) 2^v contains 2^w node-disjoint hypercubes of size 2^{v-w} each. Let $z = 2^{\lfloor \lg_2 t \rfloor}$. First, we pack the u_i values into the first z processors. In other words, we route (monotone) value u_i to processor $\lceil i/2 \rceil$ if $i \leq 2(t - z)$ and to processor $i - (t - z)$ if $i > 2(t - z)$. Each processor $k \leq z$ now has at most two values.

Next, we "break up" the N-processor hypercube into z subhypercubes of size N/z so that each processor $k \leq z$ is in a different subcube. This can be accomplished if all the nodes with the same last $\lg_2 z$ digits are assigned to the same (N/z)-processor subcube. Now, each processor $k \leq z$ copies its u values to all the processors in its subcube using a parallel prefix operation. To finish up the row-copy, we need to unpack the u values using monotone routing. If processor $k \leq N$ has two u values, it sends its lower-subscripted u value to processor $\lfloor k/z \rfloor t + 2(k \bmod z) - 1$ and its higher-subscripted u value to processor $\lfloor k/z \rfloor t + 2(k \bmod z)$. Otherwise, if processor k has only one u value, it sends its u value to processor $\lfloor k/z \rfloor t + (t - z) + k \bmod z$. Since the only operations used by the row-copy algorithm are monotone routing and parallel prefix, this algorithm takes $O(\lg N)$ time on an (N)-processor hypercube.

3.2. *A Technical Lemma.* We begin with a technical lemma that gives the flavor of our approach to the three issues mentioned above.

LEMMA 3.1. *Given an $m \times n$ Monge array $A = \{a[i, j]\}$, $m \geq n$, suppose we know the minimum in every $(\lfloor m/n \rfloor)$th row of A. Then we can compute the remaining row minima of A in $O(\lg m)$ time using an (m)-processor hypercube.*

PROOF. For the sake of simplicity, we only prove this lemma for m and n being powers of 2. In this proof, i is always in the range $1 \leq i \leq n$. Let j_i denote the index of the column containing the minimum entry of row $i(m/n)$. Also, let $j_0 = 1$. Assume that processors $1, \ldots, (n+1)$ contain $j_0, \ldots, j_n$. Note that $j_{i-1} \leq j_i$ because of the Monge condition. Consider a subarray A_i of A containing rows $(i-1)(m/n)+1$ through $i(m/n)-1$ and columns j_{i-1} through j_i. Let $|A_i|$ denote the number of elements in A_i. Since A is Monge, the minima in rows $(i-1)(m/n)+1$ through $i(m/n)-1$ must lie in A_i. Thus, the total number of elements under consideration for the remaining row minima of A is

$$\sum_{i=1}^{n} |A_i| = \sum_{i=1}^{n} \left(\frac{m}{n} - 1\right)(j_i - j_{i-1} + 1) = \left(\frac{m}{n} - 1\right)(j_n - j_0 + n + 1) \leq \left(\frac{m}{n}\right) 2n \leq 2m.$$

Since there are m processors and at most $2m$ candidates for row minima, the row minima can be determined by a segmented parallel prefix operation, provided that the data is distributed so that the processors dealing with the entries in the same row of A are "neighbors" in the parallel prefix, i.e., have consecutive indices. The procedure to satisfy these conditions is broken up into three steps:

(i) Subdivide the m processors into n groups of sizes $|A_1|, |A_2|, \ldots, |A_n|$.
(ii) Assign the processors in the group associated with A_i to the different entries of A_i.
(iii) Distribute the appropriate values from the distance vectors g and h to each processor so that it can compute its assigned entries in A_i.

To simplify this proof, we first show how to satisfy these conditions on $2m$ processors. The first step is accomplished as follows. Processor i sends value j_{i-1} to processor $i-1$. This is simply monotone routing. Processor i computes

$$|A_i| = \left(\frac{m}{n} - 1\right)(j_i - j_{i-1} + 1).$$

Processors 1 through n perform a parallel prefix on the $|A_i|$'s, so that processor i computes the value $u_i = \sum_{k=1}^{i} |A_k|$. Merge list $1, 2, \ldots, 2m$ of processor indices with list $u_1, u_2, \ldots, u_n$, where each u_i value carries with it a record containing $\langle i, j_i, j_{i-1}\rangle$. Note that since there are only $2m$ processors, any dual occurrences of a value (one occurrence from list $1, 2, \ldots, 2m$ and the other from list $u_1, u_2, \ldots, u_n$) are stored in the same processor. In the resulting sorted list, there are exactly $|A_i|$ processors between processors containing u_{i-1} and u_i. Any processor containing a u_i value determines a segment boundary (or barrier). Using a segmented parallel prefix distribute the the record $\langle i, j_i, j_{i-1}\rangle$ associated with u_i to all the processors between segment boundaries u_{i-1} and u_i. As a result, the $2m$ processors are subdivided as desired and each processor in the group associated with A_i knows the values u_i, i, j_i, and j_{i-1}.

For the second step, each processor first computes its rank within its segment using segmented parallel prefix operation. More formally,

$$rank(k) = k - \max_{k'<k}\{\textit{processor } k' \textit{ contains a } v \textit{ value}\}.$$

A processor k containing the value u_i and the record $\langle i, j_i, j_{i-1}\rangle$ computes the row and the column of its entry in A_i:

$$\begin{array}{ll} \text{row} & (i-1)\dfrac{m}{n} + \left\lceil \dfrac{rank(k)+1}{j_i - j_{i-1}+1} \right\rceil \\ \text{column} & j_{i-1} + rank(k) \bmod (j_i - j_{i-1}+1) \end{array}$$

For the third step, assume that the distance vectors $g[1], \ldots, g[m]$ are stored in processors $1 \cdots m$, and vectors $h[1], \ldots, h[n]$ in processors $1 \cdots n$. Route values $h[j_{i-1}], \ldots, h[j_i]$ to the appropriate processors responsible for the first row of A_i. Similarly, route values $g[(i-1)(m/n)+1], \ldots, g[\min\{i(m/n)-1, m\}]$ to the appropriate processors responsible for the first column of A_i. Notice that in both cases the routing is monotone. Now, each processor in the first row of A_i distributes its h value down its column of A_i (i.e., to other processors responsible for that column of A_i) and each processor in the first column of A_i distributes its g value down its row of A_i. All the processors responsible for a row of A_i have consecutive indices; thus, the g values can be distributed using a segmented parallel prefix operation. We use the row-copy algorithm described in Section 3.1 to distribute the h values. Having spread all the data so that the aforementioned conditions are satisfied, run a segmented parallel prefix with each row of A_i forming a segment. This finds all the row minima of A.

All the procedures we have done in this proof took $O(\lg m)$ time on a $(2m)$-processor hypercube and hence can be done in the same asymptotic time-bound in an m-processor hypercube. □

3.3. *An Algorithm to Compute Row Minima in Two-Dimensional Monge Arrays*

THEOREM 3.2. *The row minima of an $n \times n$ Monge array $A = \{a[i, j]\}$ can be computed in $O(\lg n \lg\lg n)$ time on an (n)-processor hypercube.*

PROOF. We use divide-and-conquer techniques similar to those used by Aggarwal and Park [5]. For the sake of simplicity, we only prove this theorem for the case of $n = 2^{2^c}$,

where c is some positive integer. Assume that processor p contains entries $g[p]$ and $h[p]$.

In this proof, i is always in the range $1 \le i \le \sqrt{n}$, and, in this paragraph, ℓ is in the range $1 \le \ell \le \sqrt{n}$. Consider the $\sqrt{n} \times n$ array R formed by taking every $\sqrt{n}$th row of A. Partition R into $\sqrt{n}$ subarrays where the ℓth subarray R_ℓ contains columns $(\ell-1)\sqrt{n}+1$ through $\ell\sqrt{n}$ of R. We assign processors $(i-1)\sqrt{n}+1$ through $i\sqrt{n}$ to R_ℓ and recursively compute the row minima of R_ℓ. Notice that, for the recursion, the processors already have the appropriate h values. So, only the g values need to be distributed. First, processor $i\sqrt{n}$ routes (monotone) its g value to processor i. Using fixed permutation routing, processor i then distributes its g value to processors $i+\ell\sqrt{n}$, $1 \le \ell \le \sqrt{n}$. After the recursion, processor $(\ell-1)\sqrt{n}+i$ contains the minimum of row i in R_ℓ. Assign processors $(i-1)\sqrt{n}+1$ through $i\sqrt{n}$ to row i of R and route (a fixed permutation) the values of the minima so that processors $(i-1)\sqrt{n}+1$ through $i\sqrt{n}$ get the minimum entries in row i of all the R_ℓ's. The row minima of R is simply the minimum over the row minima of all the R_ℓ's. Route the row minima of R to processors $(i-1)\sqrt{n}+1$ through $i\sqrt{n}$.

Let j_i denote the index of the column containing the minimum entry in row $i\sqrt{n}$ of A (equivalently, row i of R), and let $j_0 = 1$. Since A is Monge, the minimum entries in rows $(i-1)\sqrt{n}+1$ through $i\sqrt{n}-1$ of A must lie in columns j_{i-1} through j_i. Let

$$v_i = \left\lceil \frac{j_i - j_{i-1}}{\sqrt{n}} \right\rceil.$$

For $1 \le \ell \le v_i$, let $S_{i,\ell}$ be the subarray of A that contains rows $(i-1)\sqrt{n}+1$ through $i\sqrt{n}$ and columns $j_{i-1}+(\ell-1)\sqrt{n}+1$ through $\min\{j_i, j_{i-1}+\ell\sqrt{n}\}$. The minimum entries in rows $(i-1)\sqrt{n}+1$ through $i\sqrt{n}$ are either in one of $S_{i,1}, \ldots, S_{i,v_i}$ or in column j_{i-1}. Arrays $S_{i,1}, \ldots, S_{i,v_i-1}$ are all $\sqrt{n} \times \sqrt{n}$. Let $w_i = (j_i - j_{i-1}) \bmod \sqrt{n}$ and let T_i be a $w_i \times w_i$ subarray of S_{i,v_i} formed by taking every $\lfloor \sqrt{n}/w_i \rfloor$th row of S_{i,v_i}. Note that T_i may be empty.

For $1 \le \ell \le v_i - 1$, assign $\sqrt{n}$ processors to $S_{i,\ell}$ and w_i processors to T_i. The total number of processors assigned is

$$\sum_{i=1}^{\sqrt{n}} (v_i - 1)\sqrt{n} + w_i = \sum_{i=1}^{\sqrt{n}} j_i - j_{i-1} \le n.$$

The processor assignment uses a similar technique to that of Lemma 3.1. Processor i copies j_{i-1} from processor $i-1$ and then computes v_i and w_i. Using parallel prefix, processor i computes

$$u_i = \sum_{\ell=1}^{i} i - 1(v_\ell - 1)\sqrt{n} + w_\ell.$$

Now processor i routes (monotone) record $\langle i, v_i, w_i, j_i \rangle$ to processor u_i. Using segmented parallel prefix with processors u_i forming the segment boundaries, compute the rank of each processor within its segment. Also, copy $\langle i, v_i, w_i, u_i, j_i \rangle$ to all the processors between u_i and u_{i+1}. Within the segment bounded by u_i and u_{i+1}, processors with ranks $(\ell-1)\sqrt{n}+1, \ldots, \ell\sqrt{n}$ are assigned to $S_{i,\ell}$, for $1 \le \ell \le v_i - 1$, and processors with

ranks $(v_i - 1)\sqrt{n} + 1, \ldots, (v_i - 1)\sqrt{n} + w_i$ are assigned to T_i. Recursively find the row minima in these subarrays. Before the recursion, distribute the data so that the pth processor of $S_{i,p}$ gets $g[i + p]$ and $h[j_i - 1 + (p - 1)\sqrt{n} + p]$ and the pth processor of T_i gets $g[i + p\lfloor\sqrt{n}/w_i\rfloor]$ and $h[j_i - 1 + (v_i - 1)\sqrt{n} + p]$. This can be accomplished in the manner of Lemma 3.1.

Next, assign $\sqrt{n}$ processors to each T_i and using Lemma 3.1 and the row minima of T_i compute the row minima of S_{i,v_i}. Finally, the minimum entry in row ℓ, $(i-1)\sqrt{n}+1 \leq \ell \leq i\sqrt{n}$, is the minimum of $a[p, j_{i-1}]$ and the v_i values obtained for row ℓ in solving the row-minima problems for $S_{i,1}, \ldots, S_{i,v_i}$. This computation is done using segmented parallel prefix.

The time complexity of this algorithm has two components: the $O(1)$ nonrecursive hypercube operations taking $O(\lg n)$ time on n processors, and the two recursive calls. For the recursions, the processor complexity is dominated by the first recursive call. Thus, the complexities are

$$\begin{aligned} T(n) &\leq 2T(\sqrt{n}) + O(\lg n) = k \lg n + 2^k T(n^{1/2^k}), \\ P(n) &\leq \max\{n, \sqrt{n}P(\sqrt{n})\} = \max\{n, n^{1-1/2^k} P(n^{1/2^k})\} \end{aligned}$$

after k levels. Using $k = \log\log n$ gives us the claimed bounds. □

3.4. *An Algorithm to Compute Row Minima in Two-Dimensional Staircase-Monge Arrays*

LEMMA 3.3. *Given an $m \times n$ staircase-Monge array $A = \{a[i, j]\}$, $m \geq n$, suppose we know the minimum in every $(\lfloor m/n \rfloor)$th row of A. Then we can compute the remaining row minima of A in $O(\lg n)$ time using an (n)-processor hypercube.*

PROOF. This proof is very similar to the proof of Lemma 3.1. Instead of the A_i's, A get partitioned into the F_i's and the P_i's discussed in detail in the proof of Lemma 2.1. All the steps given in the proof of Lemma 2.1 for finding the F_i's and the P_i's are easily adaptable for the hypercube, except for the algorithm of [16] for the ANSV problem. On the hypercube, the ANSV problem can be solved in $O(\lg n)$ time using n processors [29]. □

THEOREM 3.4. *The row minima of an $n \times n$ staircase-Monge array $A = \{a[i, j]\}$ can be computed in $O(\lg n \lg\lg n)$ time on an (n)-processor hypercube.*

PROOF. This proof follows closely the proof of Theorem 3.2. Whenever we used some algorithm for Monge arrays, we now use the corresponding algorithm for staircase-Monge arrays. Upon finding the row minima in every $\sqrt{n}$th row of A, instead of $S_{i,\ell}$'s and T_i's we get a partition into the F_i's and the P_i's discussed in detail in the proof of Lemma 2.1. Finding the F_i's and the P_i's and distributing the processors appropriately is discussed in the proof of Lemma 3.3. □

3.5. *An Algorithm to Compute Tube Minima in Three-Dimensional Monge-Composite Arrays*

THEOREM 3.5. *The tube-minima of an $n \times n \times n$ Monge-composite array $C = \{c[i, j, k]\}$ can be computed in $O(\lg n)$ time on an (n^2)-processor hypercube.*

PROOF. We use divide-and-conquer techniques similar to those used by Aggarwal and Park [5]. Assume that entry $d[i, j]$ of array D is stored in processor $(i-1)n+j$ and entry $e[j, k]$ of array E in processor $(k-1)n+j$. When convenient, we refer to processor $p = (i-1)n+j$ by the index of its D value, i.e., by $[i, j]$. In this proof, i and k are always in the range $1 \le i, k \le \sqrt{n}$, and j is always in the range $1 \le j \le n$.

Let R denote the $\sqrt{n} \times n \times \sqrt{n}$ subarray of C given by entries $\{c[i\sqrt{n}, j, k\sqrt{n}]\}$. Assign processors $[1, 1], \ldots, [1, n]$ to the n entries of tube $(\sqrt{n}, \sqrt{n})$, processors $[2, 1], \ldots, [2, n]$ for tube $(2\sqrt{n}, \sqrt{n})$, and so forth. This assignment can be accomplished using segmented parallel prefix. Note that we can devote one processor per each element of R because $|R| = n^2$. Next, distribute the data so that a processor can compute the entry of R to which it is assigned. Processor $[i\sqrt{n}, j]$ sends its D value to processors $[(i-1)\sqrt{n}+1, j], \ldots, [i\sqrt{n}-1, j]$. This can be done by routing (monotone) from processor $[i\sqrt{n}, j]$ to processor $[(i-1)\sqrt{n}+1, j]$, and copying (parallel prefix) from processor $[(i-1)\sqrt{n}+1, j]$ to processors $[(i-1)\sqrt{n}+2, j], \ldots, [i\sqrt{n}-1, j]$. Similarly, processor $[j, k\sqrt{n}]$ sends its E value to processors $[j, (k-1)\sqrt{n}+1], \ldots, [j, k\sqrt{n}-1]$. With this processor and data assignment we can compute the tube minima of R by brute force using parallel prefix.

Since C is Monge-composite, tube minima of R limits the search space for the minima in the remaining tubes of C. Let $j_{i,k}$ denote the second coordinate of the minimum entry in tube $(i\sqrt{n}, k\sqrt{n})$. Also, let $j_{0,k} = 1$, $j_{i,0} = 1$. The Monge condition gives us the following inequalities:

$$\begin{aligned} j_{i-1,k-1} &\le j_{i,k-1} \le j_{i,k}, \\ j_{i-1,k-1} &\le j_{i-1,k} \le j_{i,k}. \end{aligned}$$

Let C_{ik} denote the subarray of C containing $\{c[x, y, z]\}$ for

$$\begin{aligned} (i-1)\sqrt{n}+1 &\le x \le i\sqrt{n}, \\ j_{i-1,k-1} &\le y \le j_{i,k}, \\ (k-1)\sqrt{n}+1 &\le z \le k\sqrt{n}. \end{aligned}$$

Because C is Monge-composite, all the remaining tube minima of C are contained within the C_{ik}'s. To find the tube minima of the C_{ik}'s we break the C_{ik}'s into smaller pieces, recurse on those pieces, and finally combine the tube minima of the pieces to get the tube minima of C. We now give the details of these steps.

Let

$$v_{ik} = \left\lceil \frac{j_{i,k} - j_{i-1,k-1} + 1}{\sqrt{n}} \right\rceil \quad \text{and} \quad w_{ik} = (j_{i,k} - j_{i-1,k-1}) \bmod \sqrt{n}.$$

We break up C_{ik} into v_{ik} subarrays S_{ik}^1 through $S_{ik}^{v_{ik}}$. For $1 \le \ell \le v_{ik}$, let subarray S_{ik}^ℓ contains elements $\{c[x, y, z]\}$ for

$$\begin{aligned} (i-1)\sqrt{n}+1 \;&\le\; x \le i\sqrt{n}, \\ j_{i-1,k-1}+(\ell-1)\sqrt{n}+1 \;&\le\; y \le \min\{j_{i,k},\, j_{i-1,k-1}+\ell\sqrt{n}\}, \\ (k-1)\sqrt{n}+1 \;&\le\; z \le k\sqrt{n}. \end{aligned}$$

Let T_{ik} be the $w_{ik} \times w_{ik} \times w_{ik}$ subarray of $S_{ik}^{v_{ik}}$ formed by taking every $\lfloor \sqrt{n}/w_{ik} \rfloor$th tube of $S_{ik}^{v_{ik}}$. In other words,

$$T_{ik} = \left\{ c\left[(i-1)\sqrt{n} + x\left\lfloor \frac{\sqrt{n}}{w_{ik}} \right\rfloor, y, (k-1)\sqrt{n} + z\left\lfloor \frac{\sqrt{n}}{w_{ik}} \right\rfloor \right] \right\}$$

for

$$\begin{aligned} 1 \;&\le\; x \le w_{ik}, \\ j_{i-1,k-1}+(v_{ik}-1)\sqrt{n}+1 \;&\le\; y \le j_{i,k}, \\ 1 \;&\le\; z \le w_{ik}. \end{aligned}$$

Note that T_{ik} may be empty.

We assign n processors to each $\sqrt{n} \times \sqrt{n} \times \sqrt{n}$ array S_{ik}^ℓ and w_{ik}^2 processors to the $w_{ik} \times w_{ik} \times w_{ik}$ array T_{ik}. The total number of processors assigned to C_{ik} is $(v_{ik} - 1)n + w_{ik}^2 < (j_{i,k} - j_{i-1,k-1})\sqrt{n} + n$. Since C is Monge-composite, the total number of processors assigned to compute tube minima in all C_{ik}'s is (see [5] for details)

$$\begin{aligned} \sum_{i=1}^{\sqrt{n}} \sum_{k=1}^{\sqrt{n}} [(j_{i,k} - j_{i-1,k-1})\sqrt{n} + n] \;&\le\; n^2 + \sqrt{n} \sum_{i=1}^{\sqrt{n}} \sum_{k=1}^{\sqrt{n}} j_{i,k} - j_{i-1,k-1} \\ &\le\; n^2 + \sqrt{n}(2n\sqrt{n}) \le 3n^2. \end{aligned}$$

Assigning the correct number of processors to each C_{ik} and distributing the data that is needed by these processors is accomplished by a procedure analogous to that used in Theorem 3.2 to assign the processors to the $S_{i,\ell}$'s and the T_i's. Now we recurse on the S_{ik}^ℓ's and the T_{ik}.

Once we know the tube minima of T_{ik}, using Lemma 3.1 we can get the remaining tube minima of $S_{ik}^{v_{ik}}$. We accomplish this by running the algorithm given in Lemma 3.1 in parallel on the $\sqrt{n}$ ij-planes of $S_{ik}^{v_{ik}}$, each of which is a $\sqrt{n} \times w_{ik}$ Monge array. To find the minima in an individual ij-plane of $S_{ik}^{v_{ik}}$ takes $O(\lg n)$ time on an $(\sqrt{n})$-processor hypercube. The total effort for all the $S_{ik}^{v_{ik}}$'s is $O(\lg n)$ time on a $(2n^2)$-processor hypercube.

Finally, to get the tube minima of C_{ik} we take the minimum of the tube minima of all the S_{ik}^ℓ's. In other words, by decomposing C_{ik} into S_{ik}^ℓ's, we broke each tube of C_{ik} into v_{ik} pieces. We then recursively found the min for each piece of each tube in C_{ik}. Thus, to get a minima for some tube in C_{ik}, we take the min of all the minima of the pieces into which that tube was broken. We accomplish this by a parallel prefix operation. As

already mentioned, the tube minima of the C_{ik}'s gives us the tube minima of C and, therefore, we are done with the algorithm.

The complexity of this algorithm has three components: the $O(1)$ nonrecursive hypercube operations (including the brute-force computation of tube minima in R) taking $O(\lg n)$ time on n^2 processors; the recursive call to compute the tube minima in S_{ik}^{ℓ}'s (for $1 \leq \ell \leq v_{ik} - 1$) and T_{ik}'s; and the call to Lemma 3.1 to compute the $S_{ik}^{v_{ik}}$'s. Thus,

$$\begin{aligned} T(n) &\leq T(\sqrt{n}) + O(\lg n) = O(\lg n), \\ P(n) &\leq \max\{n^2, nP(\sqrt{n})\} = n^2. \end{aligned}$$

□

Note that for the tube-minima problem, we do not achieve the same processor bound obtained by Aggarwal and Park [5] for CREW PRAMs. Aggarwal and Park give an $O(\lg n)$-time, (n^2)-processor CREW-PRAM algorithm and then reduce the processor bound to $n^2/\lg n$ without affecting the asymptotics of the time bound. Unfortunately, the trick they use in reducing the number of processors is not readily applied to our hypercube algorithm, because of problems with the movement of data.

3.6. *Remarks on the Network Algorithms.* A normal hypercube algorithm achieves the same processor/time bounds on any of the bounded-degree variants of the hypercube [32], which we call the *normal hypercubic networks* (e.g., the butterfly, the cube-connected cycle, the shuffle-exchange, and the de Bruijn graph). Then we have the following results.

THEOREM 3.6. *The row minima of an $n \times n$ Monge or staircase-Monge array can be computed in $O(\lg n \lg\lg n)$ time using any (n)-processor normal hypercubic network.*

THEOREM 3.7. *The tube minima of an $n \times n \times n$ Monge-composite array can be computed in $O(\lg n)$ time using any n^2-processor normal hypercubic network.*

The availability of several techniques for emulation of PRAM algorithms on hypercube and related hypercubic networks imply alternate network algorithms for the previous problems directly from the PRAM algorithms. The most general purpose deterministic emulation of PRAM requires $O(\log^2 N)$ per step [11]. However, owing to the special nature of data-movement in our PRAM algorithms, we can obtain a faster emulation by using sorting. This takes $O(\log N \log\log N)$ steps on the N-processor hypercube [19].

Comparing the time bounds obtained from emulation with our direct implementation shows that we are better off by at least a factor of $O(\log N)$ in the time bound for all the problems in the present section. For example, the time bound for row minima obtained from emulation of our PRAM algorithm would yield an $O(\log^2 N \log\log N)$ algorithm instead of the present $O(\log N \log\log N)$ algorithm. It may be also worth mentioning that the $O(\log N \log\log N)$ hypercube sorting algorithm is considered too complex for implemetation. Even by using a faster randomized emulation scheme that takes $O(\log N)$ expected time per step, our direct algorithms are more efficient.

4. Applications

4.1. *The All Pairs Shortest Path Problem.* In this section we present the result of Aggarwal *et al.* [2] that apply algorithms for searching in Monge arrays to the special case of the APSP problem when the graph is acyclic and the edge weights satisfy the Monge condition.

Define the *All Pairs Shortest Path* (*APSP*) *problem* as follows: given a weighted directed graph $G = (V, E)$, $|V| = n$, we want to find the shortest path between every pair of vertices in V. The following theorem is due to Aggarwal *et al.* [2].

THEOREM 4.1 [2]. *Given a directed acyclic graph whose edge weights satisfy the Monge condition* (*or the inverse-Monge condition*), *the APSP problem can be solved in* $O(\lg^2 n)$ *time using* n^2 *processors on a CREW PRAM.*

PROOF. Let $G = (V, E)$ be a graph on n vertices whose edge weights satisfy the Monge condition. We assume that the vertices of G, $v_1, v_2, \ldots, v_n$, are given in topological order so that if $(v_i, v_j) \in E$, then $i < j$. Let $D = \{d[i, j]\}$ be the $n \times n$ cost array for G, i.e.,

$$d[i, j] = \begin{cases} \text{cost of edge } (v_i, v_j) & \text{if} \quad (v_i, v_j) \in E \quad \text{and} \quad i \neq j, \\ 0 & \text{if} \quad i = j, \\ \infty & \text{otherwise.} \end{cases}$$

Notice that the entries of D above the diagonal obey the Monge condition. Were it not for the 0 entries along the diagonal, D would be a Monge array. Thus, we call D a *diagonal-Monge* array. To solve the APSP problem, it suffices to compute D^n over the closed semiring $\{min, +\}$ (in the notation of [18], this is $(\Re \cup \{\infty\}, min, +, \infty, 0)$). Henceforth in this proof, all operations are performed over $\{min, +\}$.

Let $n \times n$ array $A = \{a[i, j]\}$ be defined as follows:

$$a[i, j] = \begin{cases} d[i, j] & \text{if} \quad i \neq j, \\ \infty & \text{otherwise.} \end{cases}$$

Furthermore, let I be the identity array for $\{min, +\}$, i.e., the array with 0's on its diagonal and ∞'s everywhere else. Note that A is a Monge array and I is the identity array for $\{min, +\}$ We can write D as $D = \min\{A, I\} = A + I$. Then $D^2 = (A + I)^2 = A^2 + AI + I$. In general, $D^i = A^i + A^{i-1} + \cdots + A^2 + A + I$. Thus, we can write $D^{2i} = A^i D^i + D^i$. This decomposition of D reduces the computation of D^n to $\lg n$ array multiplications and additions. We can easily do the array addition in $O(1)$ time using n^2 processors. The following lemma makes this decomposition useful by showing that the array multiplications can be computed efficiently.

LEMMA 4.2. *We can compute a product of an* $n \times n$ *Monge array with any* $n \times n$ *array in* $O(\lg n)$ *time using* n^2 *processors on a CREW PRAM.*

PROOF. Let A be an $n \times n$ Monge array and let X be any $n \times n$ array. The jth column of the product AX is simply the row minima of an $n \times n$ Monge array $A_j = \{a_j[i, k]\}$, where

$a_j[i, k] = a[i, k] + x[k, j]$. Since the row minima of a Monge array can be computed in $O(\lg n)$ time using n processors on an EREW PRAM (see [14]), we can compute the product AX in $O(\lg n)$ time using n^2 processors on an CREW PRAM. We cannot do this on an EREW PRAM since n processors need to read the same value of A in one time step. □

Lemma 4.2 can be used to find the product of the form $A^i D^i$ in $O(\lg n)$ time using n^2 processors. Performing $\lg n$ iterations of array multiplications and additions gives us D^n. □

4.2. *Huffman Coding.* In this section we present the first NC algorithm for the Huffman coding problem that does $o(n^2)$ work. This algorithm is due to Czumaj [20]. Larmore and Przytycka in [30] reduced the Huffman coding problem to the *Concave Least Weight Subsequence* (*CLWS*) *problem*. Define CLWS as follows: given a weight function $w(i, j)$, $1 \leq i < j \leq n$, which satisfies the Monge condition (concavity in [30] corresponds to the Monge condition), find a sequence $1 = s_1 < s_2 < \cdots < s_k = n$ such that

$$\sum_{i=1}^{n} w(s_i, s_{i+1})$$

is minimized. We can formulate the CLWS problem as a graph problem. Let $G = (V, E)$ be a weighted directed acyclic graph such that, for $v_i, v_j \in V$, $(v_i, v_j) \in E$ iff $i < j$. The Monge weight function $w(i, j)$ is then the weight of edge (v_i, v_j). The shortest path from vertex v_1 to vertex v_n corresponds to a solution to the CLWS problem. The following theorem is due to Czumaj [20] and closely follows the ideas of Galil and Park [23].

THEOREM 4.3 [20]. *Given a directed acyclic graph whose edge weights satisfy the Monge condition* (*or the inverse-Monge condition*), *we can find the shortest paths from a source vertex to all the other vertices in V in $O(\lg n \lg^r n)$ time and a total of $O((n^2 \lg^2 n)/\lg^r n)$ work on a CREW PRAM, $r \geq 1$.*

PROOF. Define G and the $n \times n$ array D as in the proof of Theorem 4.1. Assume that vertex 1 is the source vertex. In this proof, k is always in the range $1 \leq k \leq x$ for some x to be specified later and ℓ is always in the range $1 \leq \ell \leq \lfloor n/x \rfloor$. Let $D_\ell = \{d_\ell[i, j]\}$ be the $x \times x$ subarray of D where $d_\ell[i, j] = d[i, j]$ for $\ell x + 1 \leq i, j \leq (\ell + 1)x$. We compute all the D_ℓ^n's using Theorem 4.1 in $O(\lg^2 x)$ time using $x^2 \lfloor n/x \rfloor = nx$ processors.

Define $s[i]$ to be the length of the shortest path from 1 to i. To prove the theorem, we must determine $S = \{s[i]\}$. Note that S is equal to row 1 of D^n. Let $S_\ell = \{s[\ell x + 1], s[\ell x + 2], \ldots, s[(\ell + 1)x]\}$. We find S iteratively: during iteration ℓ we compute S_ℓ. Notice that we have already computed S_1 since $s[k] = d_1[1, k]$. We need the following formulation of S to do the iteration:

$$s[\ell x + k] = \min_{\substack{1 \leq i \leq \ell x \\ \ell x < j \leq \ell x + k}} \{s[i] + d[i, j] + d_\ell[j, \ell x + k]\}.$$

In other words, the shortest path from 1 to $\ell x + k$ can be divided into a part containing a path of vertices numbered no higher than ℓx, a path of vertices numbered higher than ℓx, and an edge joining these two subpaths. Computation of $s[\ell x + k]$ is broken into two parts:

$$\begin{aligned} E[j] &= \min_{1 \le i \le \ell x} \{s[i] + d[i, j]\}, \\ s[\ell x + k] &= \min_{\ell x < j \le \ell x + k} \{E[j] + d_\ell[j, \ell x + k]\}. \end{aligned}$$

We use [14] and the decomposition of D given in the proof of Theorem 4.1 to compute E in $O(\lg(\ell x))$ time using ℓx processors. Using brute-force search, we can compute S_ℓ from E and D_ℓ in $O(\lg x)$ time using $x^2/\lg x$ processors. Thus, the total complexity is

$$\begin{aligned} \mathit{Time} &= O(\lg^2 x) + \left\lfloor \frac{n}{x} \right\rfloor O(\lg(\ell x)) + \left\lfloor \frac{n}{x} \right\rfloor O(\lg x) = O\left(\lg^2 x + \frac{n}{x} \lg n + \frac{n}{x} \lg x\right), \\ \mathit{Work} &= O(nx \lg^2 x) + O\left(\left\lfloor \frac{n}{x} \right\rfloor \ell x \lg(\ell x)\right) + O\left(\left\lfloor \frac{n}{x} \right\rfloor \frac{x^2}{\lg x} \lg x\right) \\ &= O\left(nx \lg^2 x + \frac{n^2}{x} \lg n\right). \end{aligned}$$

If we take $x = n/\lg^r n$, we get

$$\begin{aligned} \mathit{Time} &= O(\lg^2 n + \lg^r n \lg n), \\ \mathit{Work} &= O\left(\frac{n^2 \lg^2 n}{\lg^r n}\right). \end{aligned}$$

□

Using Theorem 4.3 and the reduction of Larmore and Przytycka [30], Huffman codes can be computed in $O(\lg^r n \lg n)$ time and a total of $O((n^2 \lg^2 n)/\lg^r n)$ work on a CREW PRAM.

4.3. *String Editing.* In this section we give an $O(\lg m \lg n)$-time mn-processor hypercube algorithm for the string editing problem. Recall that the string editing problem is to find a sequence of *edit operations* transforming a given string $x = x_1 x_2 \cdots x_m$ to a given string $y = y_1 y_2 \cdots y_n$, $m = |x|$ and $n = |y|$, such that the sum of the individual edit operations' costs is minimized. We consider three different types of edit operations that are allowed: we can delete the symbol x_i at cost $D(x_i)$, insert the symbol y_j at cost $I(y_j)$, or substitute the symbol x_i for the symbol y_j at cost $S(x_i, y_j)$. PRAM algorithms for this problem (see [5] and [12]) reduce it to a shortest-path problem in a special kind of directed graph called a *grid-DAG* and use array-searching to solve this shortest-path problem. We give a brief overview of this reduction. Details of this reduction and other problems related to grid-DAGs are given in [5]. An $m \times n$ grid DAG $G = (V, A)$ is

defined as follows: $V = \{v_{i,j} : 0 \le i \le m \text{ and } 0 \le j \le n\}$ and

$$\begin{aligned} A = &\{(v_{i,j}, v_{i,j+1}) : 0 \le i \le m, 0 \le j \le n\} \\ &\cup \{(v_{i,j}, v_{i+1,j}) : 0 \le i \le m, 0 \le j \le n\} \\ &\cup \{(v_{i,j}, v_{i+1,j+1}) : 0 \le i \le m, 0 \le j \le n\}. \end{aligned}$$

The first set of edges is referred to as the *horizontal* edges, the second set as the *vertical* edges, and the last set as the *diagonal* edges. The reduction from the string editing problem on strings of size m and n is as follows. We create an $m \times n$ grid-DAG G with the following weight functions: the weight of a horizontal edge $(v_{i,j}, v_{i,j+1})$ is $I(y_{j+1})$, the weight of a vertical edge $(v_{i,j}, v_{i+1,j})$ is $D(x_{i+1})$, and the weight of a diagonal edge $(v_{i,j}, v_{i+1,j+1})$ is $S(x_{i+1}, y_{j+1})$. There is a one-to-one correspondence between paths from vertex $v_{0,0}$ to vertex $v_{m,n}$ and sequences of edit operations transforming x into y. Moreover, the shortest $v_{0,0} \rightsquigarrow v_{m,n}$ path corresponds to the minimum-cost sequence of edit operations. The parallel approach to finding the shortest path in this graph is divide-and-conquer. Since in the recursive steps, the subproblems actually require a many-to-many shortest-paths solution, we generalize the problem as follows. Let $s_0, \ldots, s_{m+n+1}$ denote the sources given by the vertices $v_{m,0}, v_{m-1,0}, \ldots, v_{1,0}, v_{0,0}, v_{0,1}, \ldots, v_{0,n-1}, v_{0,n}$ and let $t_0, \ldots, t_{m+n+1}$ denote the sinks given by the vertices $v_{m,0}, v_{m,1}, \ldots, v_{m,n-1}, v_{m,n}, v_{m-1,n}, \ldots, v_{1,n}, v_{0,n}$. The problem is to find all source-to-sink shortest paths. This new problem is equivalent to computing all the entries of the distance array $DIST_G$, where $DIST_G[i, j] = \{\textit{length of shortest path from } s_i \textit{ to } t_j\}$. The divide-and-conquer approach is to cut G horizontally and vertically in the middle of each dimension, producing four grid-DAGs A, B, C, and D corresponding to the four quadrants created by the cuts. Thus, A contains vertices $\{v_{i,j} : 0 \le i \le m/2 \text{ and } 0 \le j \le n/2\}$, B contains vertices $\{v_{i,j} : 0 \le i \le m/2 \text{ and } n/2 \le j \le n\}$, C contains vertices $\{v_{i,j} : m/2 \le i \le m \text{ and } 0 \le j \le n/2\}$, and D contains vertices $\{v_{i,j} : m/2 \le i \le m \text{ and } n/2 \le j \le n\}$. After recursively computing all the source-to-sink shortest paths in A, B, C, and D, we compute $DIST_{A \cup B}$ from $DIST_A$ and $DIST_B$, $DIST_{C \cup D}$ from $DIST_C$ and $DIST_D$, and, finally, $DIST_G$ from $DIST_{A \cup B}$ and $DIST_{C \cup D}$. It has been shown that in any grid-DAG G, $DIST_G$ satisfies the Monge condition. Then the computation of $DIST_{A \cup B}$ involves finding the tube minima of a Monge-composite array $DIST_A + DIST_B$. Using our tube-minima algorithms for hypercubes, we can compute $DIST_G$ from $DIST_A$, $DIST_B$, $DIST_C$, and $DIST_D$ in solving the string editing problem in $O(\lg m \lg n)$ time on an mn-processor hypercube.

4.4. *The Largest-Area Rectangle Problem.* In this section we show how to solve the Largest-Area Rectangle (LAR) problem. We reduce the LAR problem to finding row maxima in a Monge array. Recall the LAR problem definition: given a set P of n planar points, compute the largest-area rectangle that is formed by taking any two of the n points as the rectangle's opposite corners and whose sides are parallel to the x- and y-axes.

Call a rectangle *positive-sloped* (resp. *negative-sloped*) if the two points that form the rectangle are at the top-right and the bottom-left corners (resp. at the top-left and bottom-right). The solution to the LAR problem must be either a positive-sloped rectangle or a negative-sloped rectangle. We show how to find the largest-area positive-sloped rectangle. The other possibility is handled analogously. Let $x[p]$ denote the x-coordinate

of point p and let $x[p]$ denote the y-coordinate of p. Call a point $p \in P$ *maximal* if, for all $q \in P$, either $x[p] \geq x[q]$ or $y[p] \geq y[q]$; and call p *minimal* if, for all $q \in P$, either $x[p] \leq x[q]$ or $y[p] \leq y[q]$.

OBSERVATION 4.4. *The largest-area positive-sloped rectangle is formed by one maximal point and one minimal point.*

Aggarwal *et al.* [1] showed how to find the set of maximal points in a set of n points in $O(\lg n)$ time using n processors on a CREW PRAM. Minimal points can be found analogously. Let $Q = \{q_1, q_2, \ldots, q_s\}$ be the set of minimal points such that $x[q_1] \leq x[q_2] \leq \cdots \leq x[q_s]$ and let $R = \{r_1, r_2, \ldots, r_t\}$ be the set of maximal points such that $x[r_1] \leq x[r_2] \leq \cdots \leq x[r_s]$. We form an $s \times t$ array $B = \{b[i, j]\}$ as follows:

$$b[i, j] = \begin{cases} (x[r_j] - x[q_i])(y[r_j] - y[q_i]) & \text{if } x[r_j] \geq x[q_i] \text{ and } y[r_j] \geq y[q_i], \\ -\infty & \text{otherwise.} \end{cases}$$

OBSERVATION 4.5. *B is inverse-Monge.*

We use our CREW algorithm for row minima in a Monge array to find row maxima in B and thus obtain an optimal CREW-PRAM algorithm for the LAR problem that takes $O(\lg n)$ time using n processors.

4.5. *Proximity Problems for Convex Polygons.* In this section we apply our algorithms for searching in staircase-Monge arrays to the following proximity problem: given two nonintersecting convex polygons P and Q with m and n vertices, respectively, determine, for each vertex x of P,

1. the vertex of Q nearest to x that is not visible to x,
2. the vertex of Q farthest from x that is not visible to x,
3. the vertex of Q nearest to x that is visible to x, and
4. the vertex of Q farthest from x that is visible to x.

We reduce the first two problems to row-maxima (or row-minima) problems for a constant number of staircase-Monge arrays. The other two problems can be solved directly in $O(\lg(m+n))$ time using $(m+n)/\lg(m+n)$ processors on a CREW PRAM. In the discussion that follows, we give a simple $O(\lg(m+n))$-time reduction using $m+n$ CREW-PRAM processors. The reduction is based heavily on the paper by Aggarwal and Klawe [3], and the reader is referred to this work for details of the proofs. We present the relevant algorithmic details for the parallelization. We note that while their reduction was linear time, the processor-time product of our method is $O(n \lg n)$. However, since the array-searching algorithms have similar resource bounds, the reduction is not a bottleneck for the overall running time.

We begin with three previous sequential results involving convex polygons:

LEMMA 4.6. *The intersection of an infinite line with a convex p-gon can be computed in $O(\lg p)$ sequential time.*

LEMMA 4.7. *The two supporting lines from an external point to a convex polygon with p vertices can be computed in $O(\lg p)$ sequential time.*

LEMMA 4.8. *The distance between two disjoint convex polygons with m and n vertices respectively can be computed in $O(\lg mn)$ sequential time. Moreover, we can also find two points on the respective boundaries of these polygons achieving this minimum.*

This last result can be used to find a separating line between two disjoint convex polygons. Given the separating line, we can orient the coordinate axes so that P lies strictly to the left of Q. Let the leftmost and rightmost vertices of each polygon be denoted by l_P, l_Q, r_P, and r_Q. We consider only the problem of finding the farthest-invisible vertex of Q for each vertex x of P lying on P's upper chain; the farthest-invisible vertex of Q for each vertex on P's lower chain and the nearest-invisible vertex of Q for each vertex on P's lower and upper chains can be found analogously.

The following algorithm is used for the reduction. Given a convex upper chain P with vertices labeled $p_1, \ldots, p_m$ in clockwise order and a convex polygon Q with vertices $q_1, \ldots, q_n$ in clockwise order, we compute the two staircase-Monge arrays A_1 and A_2:

1. For each vertex p_i in P, compute the two supporting vertices t_i and b_i of Q such that $\overline{p_i t_i}$ and $\overline{p_i b_i}$ are tangents to Q. The portion of Q between b_i and t_i that is closer to p_i is referred to as Q's *nearside* with respect to p_i and the remaining portion as Q's *farside* with respect to p_i.
2. Define L_i to be the line containing vertices p_i and p_{i+1}. Define c_i to be the intersection of L_i and the farside of Q with respect to p_i, if it exists. Otherwise, set c_i to be the vertex t_i. Note that c_i is not necessarily a vertex of Q.
3. Let A_1 and A_2 be two $(m-1) \times n$ arrays defined as following. The (i, j)th entry of A_1 is the distance from p_{n-i} to q_j if t_{n-i}, q_j, and c_{n-i} are in clockwise order and ∞ otherwise. The (i, j)th entry in A_2 is the distance between p_i and q_j if c_i, q_j, and b_i are in clockwise order and ∞ otherwise.

From Lemmas 4.6 and 4.7, it easily follows that the first two steps given above can be implemented in $O(\lg(m+n))$ time using a linear number of processors. Computing the row maxima of A_1 and A_2 yields the farthest farside vertex of p_i. To compute the farthest nearside vertex of p_i, we define a point n_i analogous to c_i. It can be shown that the contenders for the farthest invisible nearside vertex of p_i is b_i or the neighbor of n_i (which is invisible). The problem of finding nearest invisible vertices can be reduced in an analogous fashion.

References

[1] A. Aggarwal, B. Chazelle, L. Guibas, C. Ó'Dúnlaing, and C. K. Yap. Parallel computational geometry. *Algorithmica*, 3(3):293–327, 1988.

[2] A. Aggarwal, D. Coppersmith, and B. Schieber. Personal communication, 1992.

[3] A. Aggarwal and M. M. Klawe. Applications of generalized matrix searching to geometric algorithms. *Discrete Applied Mathematics*, 27(1–2):3–23, 1990.

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

[5] A. Aggarwal and J. K. Park. Parallel searching in multidimensional monotone arrays. To appear. An earlier version of this appears as [7].

[6] A. Aggarwal and J. K. Park. Sequential searching in multidimensional monotone arrays. To appear. An earlier version of this paper appears as [8].

[7] A. Aggarwal and J. K. Park. Parallel searching in multidimensional monotone arrays. Research Report RC 14826, IBM T. J. Watson Research Center, August 1989. Submitted to *Journal of Algorithms*. Portions of this paper appear in *Proceedings of the* 29*th Annual IEEE Symposium on Foundations of Computer Science*, pages 497–512, 1988.

[8] A. Aggarwal and J. K. Park. Sequential searching in multidimensional monotone arrays. Research Report RC 15128, IBM T. J. Watson Research Center, November 1989. Submitted to *Journal of Algorithms*. Portions of this paper appear in *Proceedings of the* 29*th Annual IEEE Symposium on Foundations of Computer Science*, pages 497–512, 1988.

[9] A. Aggarwal and J. K. Park. Improved algorithms for economic lot-size problems. *Operations Research*, 41(3):549–571, 1993.

[10] A. Aggarwal and S. Suri. Fast algorithms for computing the largest empty rectangle. In *Proceedings of the* 3*rd Annual ACM Symposium on Computational Geometry*, pages 278–290, 1987. Submitted to *SIAM Journal on Computing*.

[11] H. Alt, T. Hagerup, K. Mehlhorn, and F. Preparata. Deterministic simulation of idealized parallel computers on more realistic ones. *SIAM Journal on Computing*, 16(5):808–835, 1987.

[12] A. Apostolico, M. J. Atallah, L. L. Larmore, and H. S. McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM Journal on Computing*, 19(5):968–988, 1990.

[13] M. J. Atallah. A faster parallel algorithm for a matrix searching problem. *Algorithmica*, 9(2):156–167, 1993.

[14] M. J. Atallah and S. R. Kosaraju. An efficient parallel algorithm for the row minima of a totally monotone matrix. *Journal of Algorithms*, 13(3):394–403, 1993.

[15] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. Miller, and S. Teng. Constructing trees in parallel. In *Proceedings of the* 1*st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 421–431, 1989.

[16] O. Berkman, B. Schieber, and U. Vishkin. Optimal doubly logarithmic parallel algorithms based on finding nearest smaller values. *Journal of Algorithms*, 14(3):344–370, 1993.

[17] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, 1974.

[18] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[19] R. Cypher and G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. In *Proceedings of the* 22*nd Annual Symposium on Theory of Computing*, pages 193–203, 1990.

[20] A. Czumaj. Personal communication (via Alok Aggarwal), 1992.

[21] D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano. Sparse dynamic programming i: Linear cost functions. *Journal of the ACM*, 39(3):519–545, 1992.

[22] D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano. Sparse dynamic programming ii: Convex and concave cost functions. *Journal of the ACM*, 39(3):546–567, 1992.

[23] Z. Galil and K. Park. Parallel dynamic programming. Unpublished manuscript, 1992.

[24] A. J. Hoffman. On simple linear programming problems. In V. Klee, editor, *Convexity*: *Proceedings of the Seventh Symposium in Pure Mathematics of the AMS*, Proceedings of Symposia in Pure Mathematics, volume 7, pages 317–327. American Mathematical Society, Providence, RI, 1963.

[25] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[26] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM*, 24(1):1–13, 1977.

[27] M. M. Klawe and D. J. Kleitman. An almost linear time algorithm for generalized matrix searching. *SIAM Journal on Discrete Mathematics*, 3(1):81–97, 1990.

[28] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[29] D. Kravets and C. G. Plaxton. All nearest smaller values on the hypercube. *IEEE Transactions on Parallel and Distributed Systems*, 1995. To appear.

[30] L. L. Larmore and T. M. Przytycka. Parallel construction of trees with optimal weighted path length. In *Proceedings of the* 3*rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 71–80, 1991.

[31] L. L. Larmore and B. Schieber. On-line dynamic programming with applications to the prediction of RNA secondary structure. In *Proceedings of the* 1*st Annual ACM–SIAM Symposium on Discrete Algorithms*, pages 503–512, 1990.

[32] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures*: *Arrays*, *Trees*, *Hypercubes*, volume 1. Morgan Kaufmann, San Mateo, CA, 1992.

[33] R. C. Melville. An implementation technique for geometry algorithms. Unpublished manuscript. A.T.&T. Bell Laboratories, Murray Hill, NJ, 1989.

[34] G. Monge. Déblai et remblai. In *Histoire de l'Académie Royale des Sciences, Année MDCCLXXXI, avec les Mémoires de Mathématique et de Physique, pour la même Année, Tirés des Registres de cette Académie*, 1784.

[35] S. Ranka and S. Sahni. *Hypercube Algorithms*: *With Applications to Image Processing and Pattern Matching*. Bilkent University Lecture Series. Springer-Verlag, New York, 1990.

[36] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[37] F. F. Yao. Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the* 12*th Annual ACM Symposium on Theory of Computing*, pages 429–435, 1980.