Parallel Searching in Generalized Monge Arrays with Applications

(extended abstract)

Alok Aggarwal* Dina Kravets† James K. Park‡ Sandeep Sen§

Abstract

This paper investigates the parallel time and processor complexities of several searching problems involving *Monge* and *Monge-composite* arrays. We present array-searching algorithms for concurrent-read-concurrent-write (CRCW) PRAMs, concurrent-read-exclusive-write (CREW) PRAMs, hypercubes, cube-connected-cycles, and shuffle-exchange networks. All these algorithms run in 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.

1 Introduction

1.1 Background

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

$$a[i,j] + a[k,l] \le a[i,l] + a[k,j] \,. \qquad (1.1)$$

Similarly, A is called *inverse-Monge* if for $1 \le i < k \le m$ and $1 \le j < l \le n$,

$$a[i,j] + a[k,l] \ge a[i,l] + a[k,j] \,. \qquad (1.2)$$

*IBM Research Division, T. J. Watson Research Center, Yorktown Heights, NY 10598.

†Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. Supported in part by the Air Force under Contract OSR-86-0076, the Defense Advanced Research Projects Agency under Contract N00014-89-J-1988, and the Army under Contract DAAL-03-86-K-0171.

‡Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. 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.

§Department of Computer Science, Duke University, Durham, NC 27706.

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

1. every entry is either a real number or ∞,
2. $b[i,j] = \infty$ implies $b[i,l] = \infty$ for $l > j$ and $b[k,j] = \infty$ for $k > i$, and
3. for $1 \le i < k \le m$ and $1 \le j < l \le n$, (1.1) holds if all four entries $a[i,j]$, $a[i,l]$, $a[k,j]$, and $a[k,l]$ are finite.

The definition of a *staircase-inverse-Monge* array is identical, except that (1.2) holds if all four entries $a[i,j]$, $a[i,l]$, $a[k,j]$, and $a[k,l]$ are finite. Observe that a Monge array is only 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.

Monge arrays have many applications. In the late eighteenth century, G. Monge [Mon81] 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 distance traveled, then the paths followed in transporting these quantities must not properly intersect. In 1961, A. J. Hoffman [Hof61] elaborated upon this idea and showed that a greedy algorithm correctly solves the transportation problem for m sources and n sinks if the corresponding $m \times n$ cost array is a Monge array. More recently, Monge arrays have found applications in a number of other areas. F. Yao [Yao80] used these arrays to obtain an efficient sequential algorithm for computing optimal binary trees. Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87] 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 [AP89b] gave efficient sequential algorithms based on the Monge-array abstraction for several problems in computational geometry and VLSI river routing. Furthermore, many researchers [AP89b, LS89, EGGI90] have used Monge arrays to obtain efficient dynamic programming algorithms for problems related to molecular biology. And, more recently, Aggarwal and Park [AP90] have used Monge arrays to obtain efficient algorithms for the economic-lot size model.

Like Monge arrays, staircase-Monge arrays have also found applications in a number of areas. Aggarwal and Park [AP89b], Larmore and Schieber [LS89], and Eppstein, Giancarlo, and Galil [EGGI90] use staircase-Monge arrays to obtain algorithms for problems related to molecular biology. Aggarwal and Suri [AS87] 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 [AK88] and Klawe and Kleitman [KK88] 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 [AP89a] 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 string editing and other related problems. (See also [AALM88].) Similarly, Atallah, Kosaraju, Larmore, Miller, and Teng [AKL+89] have used Monge-composite arrays to construct Huffman and other such codes on CRCW- and CREW-PRAMs.

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 algorithms for searching in Monge, staircase-Monge, and Monge-composite arrays in the CRCW- and CREW-PRAM models of parallel computation, as well as in several interconnection networks including the hypercube, the cube-connected cycles, and the shuffle-exchange network. However, before we can describe our results, we need a few definitions; these are provided in the next subsection.

1.2 Definitions

The row-maxima problem for a two-dimensional array is that of finding the maximum entry in each row of the array. (If a row has several maxima, 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 will be given in later sections. Aggarwal, Klawe, Moran, Shor, and Wilber [AKM+87] showed that the row-maximum problem for an $m \times n$ Monge array can be solved in $\Theta(m+n)$ time. Also, Aggarwal and Park [AP89a] have shown that the row-maximum problem for such an array can be solved in $O(\lg n + \lg m)$ time on an $(n+m)$-processor CRCW-PRAM, and in $O(\lg nm \lg\lg nm)$ time on an $((n+m)/\lg\lg nm)$-processor CREW-PRAM. Note that all the algorithms dealing with finding row-maxima in Monge and inverse-Monge arrays can also be used to solve the analogously-defined row-minima problem for the same arrays since reversing the order of an array's columns and/or negating its entries allows us to move back and forth among these problems.

Unfortunately, the row-minima and row-maxima problems are not interchangeable when dealing with staircase-Monge and staircase-inverse-Monge arrays. Aggarwal and Klawe [AK88] showed that the row-minimum 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 [KK88] have improved the time bound to $O(m + n\alpha(m))$, where $\alpha(\cdot)$ is the inverse Ackermann's function. However, if we wanted to solve the row-maximum problem (instead of the row-minimum problem) for an $m \times n$ staircase-Monge array, then we could, in fact, employ the sequential algorithm given in [AKM+87] and solve the row-maximum problem in $\Theta(m+n)$ time. No parallel algorithms were known for solving the row-minimum problem for staircase-Monge arrays.

Given a $p \times q \times r$ Monge-composite array, for $1 \leq i \leq p$ and $1 \leq j \leq q$, the (i,j)-th tube consists of all those entries of the array whose first two coordinates are i and j, respectively. The tube maxima problem for a $p \times q \times r$ Monge-composite array is that of finding the maximum entry in each tube of the array. (If a tube has several maxima, then we take the one with the minimum third coordinate.) For sequential computation, the result of [AKM+87] can be trivially used to solve the tube maxima problem in $O((p+r)q)$ time. Aggarwal and Park [AP89a] and Atallah, Apostolico, Larmore, and McFaddin [AALM88] have independently shown that the tube maxima problem for an $n \times n \times n$ Monge-composite array can be solved in $\Theta(\lg n)$ time using $n^2/\lg n$ processors on a CREW-PRAM, and recently, Atallah [Ata89] has shown that this tube-maxima problem can be solved in $\Theta(\lg\lg n)$ time using $n^2/\lg\lg n$ processors on a CRCW-PRAM. In view of the applications, we assume that the two $n \times n$ Monge arrays $D = \{d_{ij}\}$ and $E = \{e_{jk}\}$ 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 will be given later. No efficient algorithms (other than the one that simulates the CRCW-PRAM algorithm) were known for solving the tube-maxima problem for a hypercube or a shuffle-exchange network.

Finally, we illustrate the utility of Monge arrays by the following example. Suppose we are given a convex polygon and that we divide it into two convex chains P and Q (containing m and n vertices, respectively) by removing two edges, as is shown in Figure 1.1. Let $p_1, \ldots, p_m$ denote the vertices of P in counterclockwise order and let $q_1, \ldots, q_n$ denote the vertices of Q in counterclockwise order. Then for $1 \leq i < k \leq m$ and $1 \leq j < l \leq n$, consider the quadrilateral formed by p_i, p_k, q_j, and q_l. By the *quadrangle inequality* (which states that the sum of the lengths of the diagonals of any quadrilateral is greater than the sum of the lengths of any pair of opposite sides), we have

$$\mathrm{d}(p_i, q_j) + \mathrm{d}(p_k, q_l) \geq \mathrm{d}(p_i, q_l) + \mathrm{d}(p_k, q_j).$$

Thus, if we imagine an $m \times n$ array $A = \{a_{i,j}\}$ where $a_{i,j}$ is the Euclidean distance from vertex $p_i \in P$ to vertex $q_j \in Q$, then by (1.2) this array is inverse-Monge. Moreover, any entry of this array can be computed in constant time, since it is just the Euclidean distance between two points. Thus, using the row-maxima algorithm of [AKM+87], we can find the farthest vertex in Q for every vertex in P in $O(m+n)$ time.

1.3 Main Results of this Paper

The time and processor complexities of algorithms for computing row-maxima in two-dimensional Monge, row-minima in two-dimensional staircase-Monge arrays and tube-maxima in three-dimensional Monge-composite arrays are listed in Tables 1.1, 1.2 and 1.3, respectively. Observe that Tables 1.1 and 1.2 show that our results for staircase-Monge arrays subsume those for Monge arrays.

In the following, we list some applications of these new array-searching algorithms; some details regarding the third application are given in the appendix, whereas details regarding the other applications will appear in the final version of this paper.

1. *The largest-area empty rectangle problem.* Consider the following problem: given a rectangle containing n points, compute the largest-area rectangle that is contained in the given rectangle, whose sides are parallel to the given rectangle, and that does not contain any of the n given points in its interior. For the sequential case, Aggarwal and Suri [AS87] gave an $O(n \lg^2 n)$-time algorithm for solving this problem, and recently, Aravind and Pandurangan [AP89c] have provided two parallel algorithm for the CREW-PRAM; one algorithm takes $O(\lg^3 n)$ time and uses $O(n \lg n)$ processors and the other algorithm takes $O(\lg n)$ time and uses $O(\frac{n^2}{\lg n})$ processors. Using the results on staircase-Monge arrays, we can obtain an $O(\lg^2 n)$-time algorithm on a CRCW-PRAM with $n \log n$ processors and an $O(\lg^2 n \lg\lg n)$-time algorithm on a CREW-PRAM with $n \lg n / \lg\lg n$ processors. Consequently, for both CRCW-PRAMs and CREW-PRAMs our algorithms improve the processor-time product.

2. *The largest-area (not necessarily empty) rectangle 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 use the algorithms developed here to obtain an optimal CRCW-PRAM algorithm that takes $\Theta(\lg n)$ 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 [Mel89]. 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 [Mel89], Melville argues that this pair of nodes correspond to the pair forming the largest-area rectangle.

3. *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 non-intersecting convex polygons P and Q, determine for each vertex x of P, the vertex of Q nearest to x that is visible (not visible, respectively) to x. If P and Q contain m and n vertices, respectively, then the nearest-visible-neighbor problem can be easily solved in $\Theta(\lg(m+n))$ 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 that the nearest-invisible-neighbor problem can be solved in $O(\lg(m+n))$ time on a CRCW-PRAM with $n+m$ processors and in $O(\lg(m+n)\lg\lg(m+n))$ time using $(m+n)/\lg\lg(m+n)$ processors on a CREW-PRAM. The farthest-visible-neighbor (farthest-invisible-neigbor) 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 (nearest-invisible-neighbor, respectively) problem.

4. *The string editing problem and other related problems.* Consider the following problem: given two

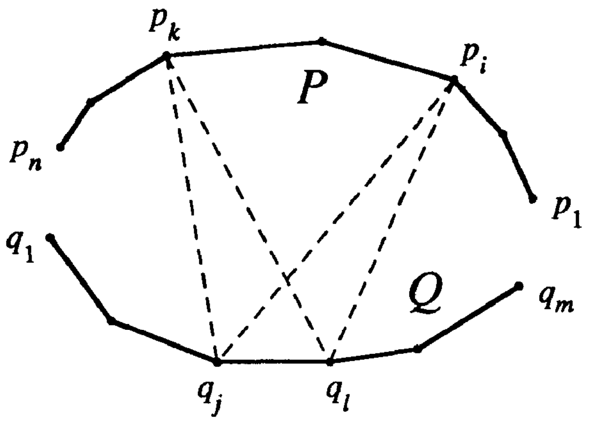

Figure 1.1: For $1 \le i < k \le m$ and $1 \le j < l \le n$, $d(p_i, q_j) + d(p_k, q_l) \ge d(p_i, q_l) + d(p_k, q_j)$.

Model	*Time*	*Processors*	*Reference*
CRCW-PRAM	$O(\lg n)$	n	[AP89a]
CREW-PRAM	$O(\lg n \lg\lg n)$	$n/\lg\lg n$	[AP89a]
hypercube, etc.	$O(\lg n \lg\lg n)$	$n/\lg\lg n$	Theorem 3.2

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

input strings $x = x_1x_2\ldots x_s$ and $y = y_1y_2\cdots y_t$, $s = |x|$ and $t = |y|$, 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 [WF74], Wagner and Fischer gave an $O(st)$-time sequential algorithm for this problem. PRAM algorithms for this problem were provided in [AP89a, AALM88]; these algorithms reduce the string editing problem to a shortest-paths problem in a special kind of directed graph called a *grid-DAG* and use array-searching to solve this shortest-paths problem. (Details of this reduction and other problems related to grid-DAGs are given in [AP89a].) Using our tube-maxima algorithms for hypercubes and related networks, we show that the string editing problem for an m-character start string and an n-character target string can be solved in $O(\lg n \lg m)$ time on an nm-processor hypercube, cube-connected cycles, or shuffle-exchange network. This improves the result of Ranka and Sahni [RS88], who obtained algorithms for a SIMD hypercube that determine a minimum cost edit sequence to transform one string of length n into another string of length n. For n^2p, $1 \le p \le n$, processors, they give an algorithm that runs in time $O(\sqrt{\frac{n \lg n}{p}} + \lg^2 n)$; for p^2, $n \lg n \le p^2 \le n^2$, processors, they give an algorithm that runs in time $O(\frac{n^{1.5}}{p}\sqrt{\lg n})$.

2 PRAM Algorithms for Searching in Staircase-Monge Arrays

In this section, we give CREW- and CRCW-PRAM algorithms for computing row minima in staircase-Monge arrays. We begin with a number of technical lemmas.

Lemma 2.1 *The row minima of an $m \times n$ Monge array can be computed in $O(\lg m + \lg n)$ time using $(m/\lg m) + n$ processors in the CRCW-PRAM model.*

Proof The row minima of an $n \times n$ Monge array can be computed in $O(\lg n)$ time using n processors on a CRCW-PRAM [AP89a, AALM88]. We consider two cases.
Case 1: If $m \ge n$, then consider the $n \times n$ array A' that is formed by taking every $\lceil m/n \rceil$-th row of A; clearly, this array is Monge, and using [AP89a], its row minima can be computed in $O(\lg n)$ time using n processors. Furthermore, it is easily seen that at most $\lceil m/n \rceil n + m = O(m)$ entries of A need be considered

Model	*Time*	*Processors*	*Reference*
CRCW-PRAM	$O(\lg n)$	n	Theorem 2.3
CREW-PRAM	$O(\lg n \lg\lg n)$	$n/\lg\lg n$	Theorem 2.3
hypercube, etc.	$O(\lg n \lg\lg n)$	$n/\lg\lg n$	Theorem 3.3

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

Model	*Time*	*Processors*	*Reference*
CRCW-PRAM	$\Theta(\lg\lg n)$	$n^2/\lg\lg n$	[Ata89]
CREW-PRAM	$\Theta(\lg n)$	$n^2/\lg n$	[AP89a, AALM88]
hypercube, etc.	$\Theta(\lg n)$	n^2	Theorem 3.4

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

for the remaining row minima. Hence, these row minima can be computed in $O(\lg m)$ time using $m/\lg m$ processors.

Case 2: If $m < n$, we partition the array into $\lceil n/m \rceil$ square arrays of size $m \times m$ (except possibly the last one). For each such array, we compute the row minima in $O(\lg m)$ time using m processors for each array. Note that the total number of processors required is at most $\lceil n/m \rceil m = n + 1$. The minima for each row of the original array can be computed by computing the minimum of $\lceil n/m \rceil$ elements which are the row minima of the partitioned array. This can be done in $O(\lg n)$ time using $n/\lg n$ processors. ■

In [AK88], 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 [KK88]. In the discussion below we extend the results of [AP89a] to staircase-Monge arrays.

Let $A = \{a[i,j]\}$ be an $m \times n$ staircase-Monge array, $m \geq n$, and 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 j-th 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 array consisting of the rows R_i^t. Clearly, A^t is a staircase-Monge array. We claim the following lemma.

Lemma 2.2 *Given the row minima of A^t, we can compute the row minima of A in $O(\lg m + \lg n)$ time using $(m/\lg m) + n$ processors on a CRCW-PRAM.*

Proof From [AK88], the minima of A^t induce partitioning of A such that certain regions can be left out 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 and staircase-Monge arrays (see Figure 2.2). Within each class, the arrays have non-overlapping columns (except possibly for the columns in which the minima of A^t occur) and have s rows. There are at most $2(n+1)$ feasible Monge arrays and at most $n+1$ feasible staircase-Monge arrays. It can be shown that the total number of elements in each category of the arrays is $O(m)$. Thus, a brute-force search of these elements suffices to find the row minima. Clearly, this can be done in $O(\lg m)$ time using $(m/\lg m)$ processors, since the maximum row-length of any array is n (which is less than m). 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 in $O(\lg n)$ time using n processors. In our above discussion, we ignored the issue of processor allocation. We shall now show that it can be done within the same bounds.

Let us look at the positions of the row minima of A^t more carefully (see Figure 2.2). We would like to characterize the feasible Monge regions in a manner that will enable us to do the processor allocation quickly. Notice that if the minima of R_{i+1}^t lies to the left of minima of R_i^t, then there is at most one feasible Monge region (F_6 in Figure 2.2) where the minima of the rows in A between R_i^t and R_{i+1}^t can lie and this can be quickly determined. However, if the minima of R_i^t lies to the left of R_{i+1}^t then there can be more than one feasible Monge region where the minima can lie (F_4 and F_5). The number of extra feasible Monge regions that need to be considered in this case is equal to the number of minima which are "bracketed" by the minima of R_i^t. We define "bracketed" as follows. Minima m_1 is said to bracket another minima m_2 if m_1 is the closest north-west neighbor of m_2, i.e., m_1 lies above and to the left of m_2 and among

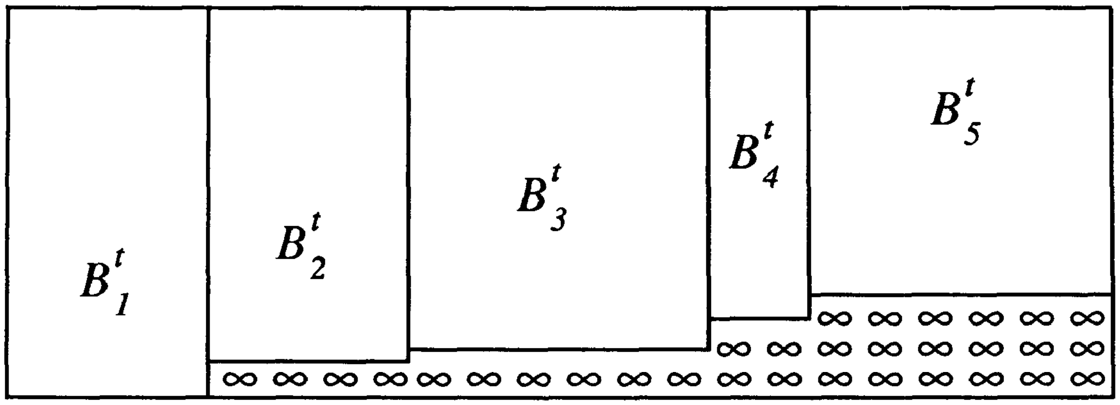

Figure 2.1: Decomposition of B^t into $B^t_1, \ldots, B^t_u$.

all the minima which have this property with respect to m_2, the row of m_1 is the maximum. We implement this as follows. We first form a list $L = (l_1, l_2, \ldots, l_s)$ such that the i-th element of this list corresponds to the minima of R^t_i. We then store the column number (y-coordinate) of this minimum in $L(i)$. For every element of this array, we have to determine the nearest neighbor to its left which has a y-coordinate less than it. In [BBG+89], Berkman, Breslauer, Galil, Schieber, and Vishkin define the All Nearest Smallest Value (ANSV) problem as follows: given a list $A = (a_1, a_2, \ldots, a_n)$ of elements from a totally ordered domain, determine for each a_i, $1 \leq i \leq n$, the nearest element to its left and the nearest element to its right that are less than a_i (if they exist). Moreover, they also give an algorithm that executes in $O(\lg n)$ time using $O(n/\lg n)$ processors in the CREW-PRAM model. Thus, an application of their ANSV algorithm followed by sorting enables us to allocate processors. If we use an $O(\lg n)$-time, n-processor sorting algorithm, then the entire procedure can be done optimally in $O(\lg n)$ time. ■

Given this, we can state the following result.

Theorem 2.3 *The row minima of an $n \times n$ staircase-Monge array can be computed in $O(\lg n \lg\lg n)$ time using $n/\lg\lg n$ processors in the CREW-PRAM model. and in $O(\lg n)$ time using n processors in the CRCW-PRAM model*

Proof We give only the CRCW-PRAM algorithm; the CREW-PRAM algorithm is analogous. We use an approach very similar to [AP89a].

1. Given the $n \times n$ staircase-Monge array B, define f_i, R_i's and R^t_i as before, except that $s = \lfloor\sqrt{n}\rfloor$. Obtain B^t from B. Let $u = \lceil n/\sqrt{n}\,\rceil$. Clearly, B^t is a $u \times n$ staircase-Monge array. Furthermore, B^t can be decomposed into at most u Monge arrays $B^t_1, \ldots, B^t_u$, such that each B^t_i is a $u_i \times v_i$ array, for $u_i \leq u$ and some $v_i > 0$ (See Figure 2.1). Using the algorithm of [AP89a], and Lemma 2.1, the row minima for these arrays can be computed in $O(\lg n)$ time using
$$\sum_{i=1}^{u}(u_i/\lg u_i + v_i) = \sum_{i=1}^{u}(\sqrt{n}/\lg\sqrt{n} + v_i) = O(n)$$
processors.

2. These minima would induce a partition of the array B, similar to that of Figure 2.2. We shall first determine the minima in all the feasible Monge arrays using Lemma 2.1. This can be done in $O(\lg n)$ time using
$$\sum_{i=1}^{2u}(s + v_i) = O(n)$$
processors.

3. 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 [AP89a] and Lemma 2.2 to bound the number of processors to $O(n)$.)

4. To find the minimum of every row, we choose the minimum of the minimum elements of the Monge arrays and the staircase-Monge array.

We can write down the recurrence relation for the time complexity as
$$T(n) = T(\sqrt{n}) + O(\lg n)\ ,$$
which yields $T(n) = O(\lg n)$. The processor complexity is $O(n)$ from our previous discussion. ■

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

Proof The proof follows on the lines of Lemma 2.1. The case corresponding to $m \le n$ is easy. Partition the array into $\lceil n/m \rceil$ arrays of size $m \times m$. Compute the row minima (in $\lg m$ time using n processors) and then compute the minimum in each row from the $\lceil n/m \rceil$ elements. This can be done in the required time using $n/\lg n$ processors. For the case $m \ge n$, we use a scheme similar to Lemma 2.2. In this case, however, we actually compute the minima of a $n \times n$ array in $O(\lg n)$ time using n processors. The bounds follow from Lemma 2.2. ■

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-maxima of two-dimensional Monge arrays, the second algorithm computes the row-minima of two-dimensional staircase-Monge arrays, and the third computes the tube maxima of three-dimensional Monge arrays. We then argue that these algorithms can also be used for shuffle-exchange graphs and other hypercube-like networks.

Each of our hypercube algorithms is based on the corresponding CREW-PRAM algorithm. However, there are three important issues that need to be addressed in converting from CREW-PRAM algorithms to hypercube algorithms:

1. we can no longer use Brent's theorem [Bre74],
2. we must deal more carefully with the issue of processor allocation, and
3. we need to worry about 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 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 $v[1], \ldots, v[m]$ and $w[1], \ldots, w[n]$ (where initially the i-th hypercube processor's local memory holds $v[i]$ and $w[i]$), such that a processor needs to know both $v[i]$ and $w[j]$ before it can compute $a[i,j]$ in constant time. Similarly, in the case of Monge-composite arrays $C = \{c[i,j,k]\}$, where $c[i,j,k] = d[i,j] + e[j,k]$, $D = \{d[i,j]\}$ and $E = \{e[j,k]\}$ are Monge arrays, and initially the entries of D and E are uniformly distributed among the local memories of the hypercube's processors, we assume that a processor needs to know both $d[i,j]$ and $e[j,k]$ before it can compute $c[i,j,k]$. The manner in which the $v[i]$, $w[j]$, $d[i,j]$, and $e[j,k]$ are distributed through the hypercube is then an important consideration.

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 $A = \{a[i,j]\}$, $m \ge n$, suppose we know the maximum in every $(\lfloor m/n \rfloor)$-th row of A. Then we can compute the remaining row maxima of A in $O(\lg m \lg\lg m)$ time using a $(2m/\lg\lg m)$-processor hypercube.*

Proof Assume that m and n are powers of 2. We first show how to compute the remaining row maxima of A in $O(\lg m)$ time using $2m$ processor hypercube. Let $j(i)$ denote the index of the column containing the maximum entry of row $i\lfloor m/n \rfloor$, for $1 \le k \le n$. Also, let $j(0) = 0$ and $j(n+1) = n$. Furthermore, for $1 \le i \le n+1$, let A_i denote the subarray of A containing rows $(i-1)\lfloor m/n \rfloor + 1$ through $\min\{i\lfloor m/n \rfloor - 1, m\}$ and columns $j(i-1)$ through $j(i)$. Let $|A_i|$ denote the number of elements in A_i. Since A is Monge, the maxima in rows $(i-1)\lfloor m/n \rfloor + 1$ through $\min\{i\lfloor m/n \rfloor - 1, m\}$ must lie in A_i. Thus, the total number of elements under consideration for the row maxima is

$$\sum_{i=1}^{n+1} |A_i| = \sum_{i=1}^{n+1} (\lfloor m/n \rfloor - 1)(j(i) - j(i-1) + 1) \le 2m.$$

Since we have $2m$ processors and $2m$ candidates for row maxima, we can determine the row maxima by doing a parallel prefix operation provided that we can distribute the data evenly among the processors. More specifically, we need to distribute the data so that

1. processors responsible for entries in A_i have the values $j(i)$ and $j(i-1)$,
2. there is one array entry per processor, and
3. the processors dealing with the entries in the same row of the array are "neighbors" in the parallel prefix.

Assume that that each processor has a unique index $1 \ldots 2m$ and that processors $1 \ldots n+2$ contain $j(0) \ldots j(n)$. We first merge lists $1 \ldots 2m$ and $j(0) \ldots j(n+1)$. This can be done in $O(\lg m)$ time [LLS89]. Then, we distribute the values i and $j(i)$ to all the elements of the sorted list between $j(i-1)$ and $j(i)$. This can be done using one parallel prefix operation which takes $O(\lg m)$ time. Similarly, we distribute the values i and $j(i)$ to all the elements of the sorted list between $j(i)$ and $j(i+1)$.

Now there are exactly $|A_i|$ processors containing the value $j(i)$, i and $j(i+1)$. Let the group of processors responsible for entries in A_i be $G_i = \langle a_i, \ldots, a_i + |A_i|\rangle$. Furthermore, we subdivide G_i into groups of $j(i) - j(i-1)$, the width of A_i, so that processors $\langle a_i, \ldots, a_i + j(i) - j(i-1)\rangle$ are responsible for the entries in the first row of A_i, processors $\langle a_i + j(i) - j(i-1) + 1, \ldots, a_i + 2(j(i) - j(i-1)) + 1\rangle$ are responsible for the second row of A_i, and so forth. Notice that because each processor has the values i, $j(i)$ and $j(i-1)$, it can determine in $O(1)$ time the entry of A for which it is responsible. We call processors responsible for row (column) s the s-row (s-column) processors. With the processors thus allocated, we distribute the appropriate values of the distance vectors to the first-row and first-column processors of A_i. Assume that the distance vectors $v[1], \ldots, v[m]$ and $w[1], \ldots, w[n]$ are stored in processors $1 \ldots m$. First, we send values $w[j(i-1)], \ldots, w[j(i)]$ to the corresponding first-row processors of A_i for $1 \leq i \leq n+1$. Notice that our allocation of processors allows us to accomplish the previous step via isotone routing which can be done in $O(\lg m)$ time [LLS89]. Similarly, we send values $v[(i-1)\lfloor m/n\rfloor+1], \ldots, v[\min\{i\lfloor m/n\rfloor-1, m\}]$ to the corresponding first-column processors of A_i for all i. Next, we have the first-row processors of A_i distribute their w values down the columns and first-column processors of A_i distribute their v values down the rows. This can be accomplished with two parallel prefix operations.

Having distributed all the data appropriately, we run a segmented parallel prefix operation with each row of A_i forming a segment.

We must now reduce the number of processors used from $2m$ to $2m/\lg\lg m$. For the CREW-PRAM, this was accomplished using Brent's theorem [Bre74]. For the hypercube, we must do this directly. The basic idea is to use the fact that p processors can compute the maximum of m numbers, $m \geq p$, in $O(m/p + \lg m)$ time. ■

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

Proof We omit the bulk of this proof, but note one further issue that must considered in transforming a CREW-PRAM algorithm into a hypercube algorithm. Specifically, we need to ensure that the size of every subproblem we solve recursively is a power of two, so that the subproblem can be assigned to and solved by a complete sub-hypercube. ■

Theorem 3.3 *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 $(n/\lg\lg n)$-processor hypercube.*

Proof Omitted. ■

Theorem 3.4 *The row maxima 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 Omitted. ■

Note that for the tube maxima problem, we do not achieve the same processor bound obtained by Aggarwal and Park [AP89a] 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 the problems with the movement of data; this will be described in the final version of this paper.

References

[AALM88] A. Apostolico, M. J. Atallah, L. L. Larmore, and H. S. McFaddin. Efficient parallel algorithms for string editing and related problems. In *Proceedings of the 26th Allerton Conference on Communication, Control, and Computing*, pages 253–263, October 1988.

[AK88] A. Aggarwal and M. M. Klawe. Applications of generalized matrix searching to geometric algorithms. *Discrete Applied Mathematics*, 1988. To appear. Presented at the Workshop on Computational Combinatorics, Simon Fraser University, August 1987.

[AKL+89] M. J. Atallah, S. R. Kosaraju, L. L. Larmore, G. Miller, and S. Teng. Constructing trees in parallel. In *Proceedings of the 1st Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 421–431, June 1989.

[AKM+87] 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. An earlier version of this paper appears in *Proceedings of the 2nd Annual ACM Symposium on Computational Geometry*, June 1986.

[AP89a] A. Aggarwal and J. Park. Parallel searching in multidimensional monotone arrays.

Journal of Algorithms, 1989. Submitted. Portions of this paper appear in *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 497–512, October 1988.

[AP89b] A. Aggarwal and J. Park. Sequential searching in multidimensional monotone arrays. *Journal of Algorithms*, 1989. Submitted. Portions of this paper appear in *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 497–512, October 1988.

[AP89c] S. Aravind and C. Pandurangan. Efficient parallel algorithms for some rectangle problems. Unpublished manuscript. Department of Computer Science, Indian Institute of Technology, Madras, India, 1989.

[AP90] A. Aggarwal and J. Park. Improved algorithms for economic lot-size problems. Unpublished manuscript, 1990.

[AS87] A. Aggarwal and S. Suri. Fast algorithms for computing the largest empty rectangle. In *Proceedings of the 3rd Annual ACM Symposium on Computational Geometry*, pages 278–290, June 1987.

[Ata89] M. J. Atallah. A faster parallel algorithm for a matrix searching problem. Technical report, Purdue University, 1989.

[BBG$^+$89] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computation*, pages 309–319, 1989.

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 2(2):201–206, 1974.

[EGGI90] D. Eppstein, Z. Galil, R. Giancarlo, and G. F. Italiano. Sparse dynamic programming. *Journal of the ACM*, 1990. To appear. An earlier version of this paper appears in *Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 513–522, January 1990 (is this right?).

[Hof61] A. J. Hoffman. On simple transportation problems. In *Convexity: Proceedings of Symposia in Pure Mathematics, Vol. 7*, pages 317–327. American Mathematical Society, 1961.

[KK88] M. M. Klawe and D. J. Kleitman. An almost linear time algorithm for generalized matrix searching. Technical Report RJ 6275, IBM Research Division, Almaden Research Center, August 1988.

[LLS89] F. T. Leighton, C. E. Leiserson, and E. Schwabe. Theory of parallel and vlsi computation: Lecture notes for 18.435/6.848. Research Seminar Series MIT/LCS/RSS 6, Massachusetts Institute of Technology, March 1989.

[LS89] L. L. Larmore and B. Schieber. On-line dynamic programming with applications to the prediction of RNA secondary structure. In *Proceedings of the 1st Annual ACM Symposium on Discrete Algorithms*, pages 503–512, January 1989.

[Mel89] R. C. Melville. An implementation technique for geometry algorithms. Unpublished manuscript. A. T. & T. Bell Laboratories, Murray Hill, NJ, 1989.

[Mon81] G. Monge. Déblai et remblai. Mémoires de l'Académie des Sciences, 1781.

[RS88] S. Ranka and S. Sahni. String editing on a SIMD hypercube multicomputer. Computer Science Technical Report 88-29, University of Minnesota, March 1988. Submitted to *Journal on Parallel and Distributed Computing.*

[WF74] R. A. Wagner and M. J. Fischer. The string to string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.

[Yao80] F. F. Yao. Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 429–435, 1980.

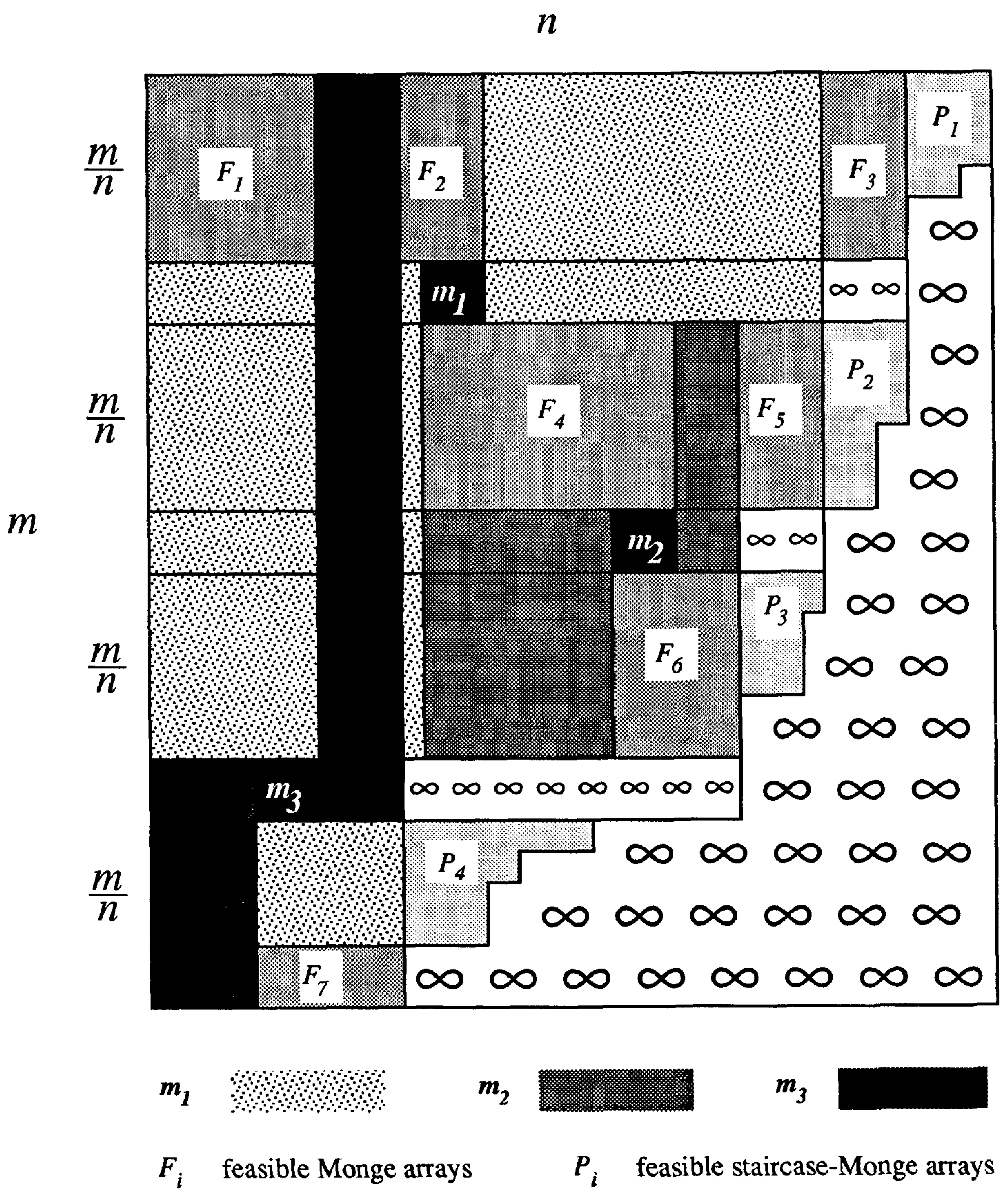

Figure 2.2: The regions covered by one of the m_i patterns indicate the infeasible zones for minima. Many of the regions are made forbidden by more than one m_i. In other words, many entries of the array could be covered by more than one pattern; in this case, we show arbitrarily one such pattern. Minimum m_2 is bracketed by m_1.