JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING **9**, 63–68 (1990)

Computing the Largest Empty Rectangle on One- and Two-Dimensional Processor Arrays

FRANK DEHNE

Center for Parallel and Distributed Computing, School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6

Given a rectangle R (with its edges parallel to the coordinate axes) containing a set $S = \{s_1, \ldots, s_n\}$ of n points in the Euclidean plane, consider the problem of finding the largest area subrectangle r in R with sides parallel to the coordinate axes that contains no point of S. We present optimal parallel algorithms for solving this problem on one- and two-dimensional arrays of processors. © 1990 Academic Press, Inc.

1. INTRODUCTION

Consider a rectangle R with its edges parallel to the coordinate axes that contains a set $S = \{s_1, \ldots, s_n\}$ of n points in the Euclidean plane. In this paper, we consider the problem of finding the largest area subrectangle r in R, with its sides parallel to the coordinate axes, that contains no point of S (see Fig. 1).

An efficient solution to this problem is of considerable interest, e.g., in VLSI manufacturing. If a rectangular silicon wafer with several points of impurity is represented by a rectangle R and a point set S, then the largest (isooriented) rectangular area on the wafer which is free of impurities is the largest empty rectangle r described above.

In [1, 3], sequential algorithms have been presented to solve this problem in time $O(m + n \log^2 n)$, $m = O(n^2)$, and time $O(n \log^3 n)$, respectively.

In this paper we present $O(n)$ and $O(\sqrt{n})$ time parallel algorithms for solving this problem on one- and two-dimensional processor arrays of sizes n and $\sqrt{n} \times \sqrt{n}$, respectively.

A one-dimensional processor array is a linear arrangement of n processing elements (PEs) where each PE is connected to its at most two direct neighbors by bidirectional communication links (see, e.g., [2]). A two-dimensional processor array is a set of n PEs arranged on a $\sqrt{n} \times \sqrt{n}$ grid, where each PE is connected to its at most four neighbors (see, e.g., [8]). For both models, each PE is assumed to have a constant number of memory cells available.

On one- and two-dimensional processor arrays, comparing two arbitrary data items takes at least time $O(n)$ and time $O(\sqrt{n})$, respectively, in the worst case; therefore the algorithms presented in this paper are asymptotically optimal.

The remainder of the paper is organized as follows: in Section 2, we present the basic structure of our algorithm, which is commonly applied to both machine models. In Sections 3 and 4 we then describe the implementation of this method on one- and two-dimensional processor arrays, respectively.

2. BASIC STRUCTURE OF THE ALGORITHM

A basic property of the largest empty rectangle r is that each edge of r is supported by either an edge of the bounding rectangle R or at least one point of S; otherwise it would be contained in a larger empty rectangle (cf. [3]). We shall call these edges or points *supporting elements* with respect to S (and R). An empty rectangle is called a *candidate,* if each of its sides has at least one supporting element. To simplify exposition we shall assume for the remainder that all points of S have distinct x_1- and distinct x_2-coordinates and, thus, the largest empty rectangle has exactly four supporting elements. The existence of more supporting elements will not change our algorithms significantly.

For both machine models we assume that initially each PE contains the coordinates of one arbitrary point of S, and that every PE stores the coordinates of the lower left and upper right corner of R. Upon termination of our algorithm, one arbitrary PE will report the largest empty rectangle r.

In the remainder of this section, we describe a general parallel method for solving the largest empty rectangle problem. In Sections 3 and 4, we then apply the method to one- and two-dimensional processor arrays and obtain optimal algorithms for these architectures. For ease of description, we split our method into three parts, which will be presented individually.

Part I

In order to compute the largest empty rectangle r of a point set S with bounding rectangle R, S is first sorted by x_1-coordinate, and the point set S is split by a vertical line l_v into two subsets S_{left} and S_{right} equal in size (i.e., $| \, |S_{\text{left}}| - |S_{\text{right}}| \, | \leq 1$); see Fig. 2a. The subproblems for S_{left} and S_{right} (and the two respective subrectangles of R induced by

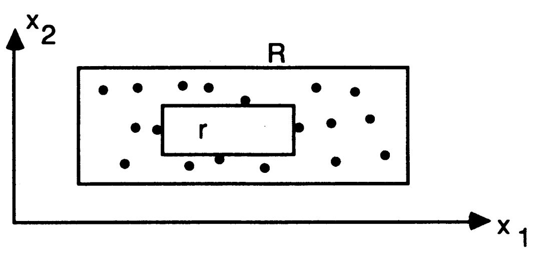

FIG. 1. Definition of the largest empty rectangle.

l_v) are then recursively solved in parallel on the left and right halves, respectively, of the processor array.

Part II

Given the largest empty rectangles with respect to S_{left} and S_{right}, respectively, the maximum area one of these both needs to be compared with the largest empty rectangle r^+ which has at least one supporting element of S_{left} and one supporting element of S_{right}, each. We apply a second divide and conquer procedure:

S is sorted by x_2-coordinate and then split by an additional horizontal line l_h into four disjoint subsets S_1, S_2, S_3, S_4 as indicated in Fig. 2b such that

$$S_{\text{left}} = S_1 \cup S_2,$$
$$S_{\text{right}} = S_3 \cup S_4, \text{ and}$$
$$|\,|S_2 \cup S_3| - |S_1 \cup S_4|\,| \leq 1.$$

Recursively, the following two subproblems are solved in parallel on one-half of the processor array, each:

(1) Compute the largest empty rectangle having at least one supporting element with respect to S_2 and S_3, each, and none with respect to S_1 or S_4.

(2) Compute the largest empty rectangle having at least one supporting element with respect to S_1 and S_4, each, and none with respect to S_2 or S_3.

Part III

In order to compute the rectangle r^+, the two rectangles obtained in Part II need to be compared with the maximum area empty rectangle r^* having the following property (which will be referred to as *Property* $*$):

Let B_1, B_2, B_3, B_4 be the sets of supporting elements of r^* with respect to S_1, S_2, S_3, S_4, respectively; then

$$|B_1| + |B_2| + |B_3| + |B_4| = 4$$
$$|B_1| + |B_2| > 0$$
$$|B_3| + |B_4| > 0$$
$$|B_2| + |B_3| > 0$$
$$|B_1| + |B_4| > 0.$$

In the remainder of this section, we discuss how to determine the rectangle r^*.

Let $p_0, \ldots, p_4$ denote the intersection points of l_v and l_h with the bounding rectangle R as indicated in Fig. 2b, and let R_i ($i = 1, \ldots, 4$) be the rectangle with opposite vertices p_{i-1} and $p_{i \bmod 4}$.

LEMMA 1. *If r' is an empty rectangle with Property $*$ and e is an edge of r' which is part of R_i ($i = 1, \ldots, 4$), then e is supported by p_{i-1} or $p_{i \bmod 4}$.*

Proof. From Property $*$ it follows that both vertical (horizontal) edges of r' cross l_h (l_v, respectively). Thus, Lemma 1 follows immediately. ■

A consequence of Lemma 1 is that, in order to determine r^*, we do not have to consider the bounding rectangles if we add instead the points p_i ($i = 0, \ldots, 3$) and consider all empty rectangles with exactly four supporting points and Property $*$ with respect to $\Sigma_1, \ldots, \Sigma_4$, where $\Sigma_i := S_i \cup \{p_{i-1}, p_{i \bmod 4}\}$. Note that, at each stage of the recursion, the current number of subproblems is $O(n)$ and therefore we add at most $O(n)$ points simultaneously.

In [4] we presented an optimal $O(\sqrt{n})$ time algorithm for computing, on a two-dimensional processor array, all maximal elements of a set S of n points in the Euclidean plane. (On a one-dimensional processor array, an $O(n)$ solution to this problem is obvious). These results are easily generalized to the computation of the maximal elements with respect to the northeast (NE), northwest (NW), southwest (SW), and southeast (SE) directions as depicted in Fig. 3. M_1, M_2, M_3, M_4 will refer to the NE, SE, SW, NW maximal elements of $\Sigma_1, \Sigma_2, \Sigma_3, \Sigma_4$, respectively.

LEMMA 2. *If r' is a maximum area empty rectangle supported by four points $\{t_1, \ldots, t_4\} \subseteq \Sigma_1 \cup \Sigma_2 \cup \Sigma_3 \cup \Sigma_4$ with Property $*$, then $\{t_1, \ldots, t_4\} \subseteq M_1 \cup M_2 \cup M_3 \cup M_4$.*

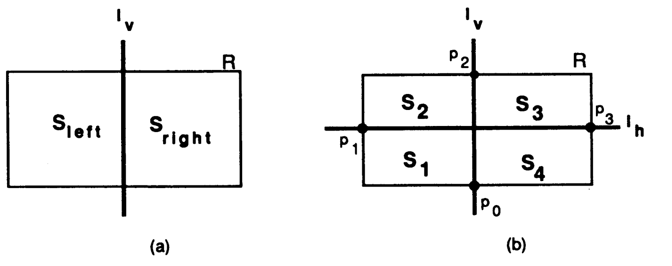

FIG. 2. Recursive splitting of the largest empty rectangle problem.

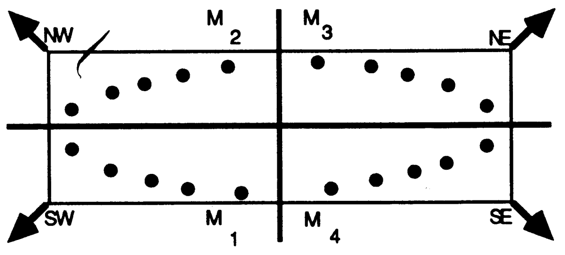

FIG. 3. The maximal elements of $\Sigma_1, \Sigma_2, \Sigma_3$, and Σ_4, respectively.

Proof. From Property $*$ it follows that both vertical and horizontal edges of r' cross l_h and l_v, respectively, Furthermore, r' has to be empty. Thus, Lemma 2 follows from the definition of maximality. ■

Summarizing Lemmas 1 and 2, we obtain:

THEOREM 1. *The rectangle r^* described above can be computed by finding the maximum area rectangle of all empty rectangles which are supported by four points $t_1, \ldots, t_4 \in B_1 \cup B_2 \cup B_3 \cup B_4$ with $B_i \subseteq M_i (i, \ldots, 4)$, $|B_1| + |B_2| + |B_3| + |B_4| = 4$, $|B_1| + |B_2| > 0$, $|B_3| + |B_4| > 0$, $|B_2| + |B_3| > 0$, and $|B_1| + |B_4| > 0$.*

All possible (16) cases that match these requirements are listed in Fig. 4. All combinations of cardinalities of B_1, B_2, B_3, B_4 are derived from Theorem 1. There are essentially three types of empty rectangles which need to be considered:

A *type A* rectangle is supported by two points of M_1 [M_2] and M_3 [M_4, respectively], each. A *type B* rectangle is supported by two points of one quadrant (M_1, M_2, M_3, or M_4) and one point each of two other quadrants, while a *type C* rectangle is supported by one point of each quadrant.

The directions of support indicated in Fig. 4 are derived as follows:

—For type A rectangles (Cases 3 and 4), the two possible combinations of directions of support are obvious.

—For type B rectangles consider, e.g., Case 5. There is only one supporting point in the left half (i.e., $M_1 \cup M_2$), which must be a left support since, if any of the other three points were a left support, this point could not be a supporting element at all. Furthermore, since there is only one supporting point in the upper half (i.e., $M_2 \cup M_3$) this must be an upper support. Hence, the two points of B_4 are lower and right supporting elements, respectively.

For all other cases of type B, a similar argument holds.

—For exactly one supporting point in each quadrant (type C) there are two possible cases. The supporting point in the lower left quadrant M_1 is a left or lower support; otherwise there could be no supporting point in M_2, M_3, or M_4. Assuming that it is a left [lower] support, the other three directions of support are uniquely determined.

Our strategy for determining the rectangle r^* is to compute the largest empty rectangle for each case (if it exists) separately, and then determine the maximum area one of these.

Before we proceed with describing the implementation of the above method on one- and two-dimensional processor arrays, it is necessary to state the following *definitions and observations:*

The x_1- and x_2-coordinates of a point x are denoted by $x[1]$ and $x[2]$, respectively. Two points $a, b \in M_i$ ($i = 1, \ldots, 4$) with $a[1] < b[1]$ are called *close neighbors* of M_i if and only if there is no other $c \in M_i$ with $a[1] < c[1] < b[1]$.

LEMMA 3. *If r' is an empty rectangle for which the conditions of Theorem 1 hold, and $\{t_1, t_2\} = B_i \subseteq M_i$ is a set of two supporting points of r' in the same quadrant M_i ($i = 1, \ldots, 4$), then t_1 and t_2* are close neighbors in M_i.

Proof. Let $i = 1$, and let r' be an empty rectangle as described above (see Fig. 5). Assume there is a $t' \in M_1$ with $t_1[1] < t'[1] < t_2[1]$; then t' lies inside r' since it is NE-maximal with respect to Σ_1, which is a contradiction. ■

Hence, if two supporting elements are in the same quadrant then the respective corners of the empty rectangle r^* are defined by a pair of close neighbors of maximal elements. Therefore, we define the sets

$\mathscr{C}(M_1) := \{(\min\{p[1], q[1]\}, \min\{p[2], q[2]\}) \mid p, q$ close neighbors in $M_1\}$,
$\mathscr{C}(M_2) := \{(\min\{p[1], q[1]\}, \max\{p[2], q[2]\}) \mid p, q$ close neighbors in $M_2\}$,
$\mathscr{C}(M_3) := \{(\max\{p[1], q[1]\}, \max\{p[2], q[2]\}) \mid p, q$ close neighbors in $M_3\}$, and
$\mathscr{C}(M_4) := \{(\max\{p[1], q[1]\}, \min\{p[2], q[2]\}) \mid p, q$ close neighbors in $M_4\}$,

where $\mathscr{C}(M_i)$ is referred to as the sets of *corner elements* of M_i, i.e., the set of possible vertices of empty rectangles induced by pairs of close neighbors of M_i. For each $p \in \mathscr{C}(M_1)$ we define (see Fig. 6):

—$x_1(p) := \min\{q[1] \mid q \in M_4$ and $q[2] > p[2]\}$;
—$x_2(p) := \min\{q[2] \mid q \in M_2$ and $q[1] > p[1]\}$;
—p_u [p_l] is the point in $\mathscr{C}(M_3)$ with maximum x_2-coordinate smaller than $x_2(p)$ [maximum x_1-coordinate smaller than $x_1(p)$, respectively];

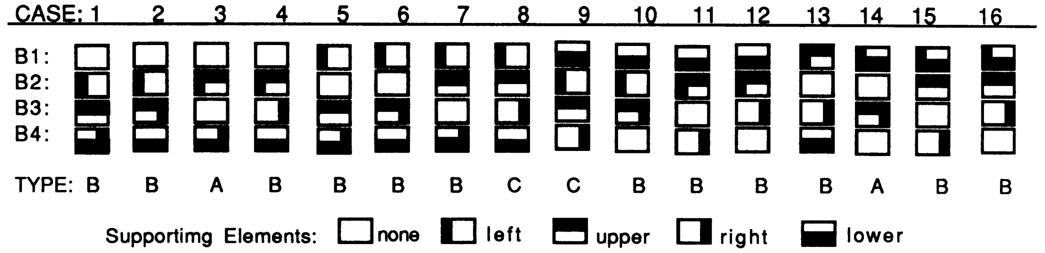
CASE: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
B1:
B2:
B3:
B4:
TYPE: B B A B B B B C C B B B B A B B
Supportimg Elements: none left upper right lower

FIG. 4. The possible cases following from Theorem 1.

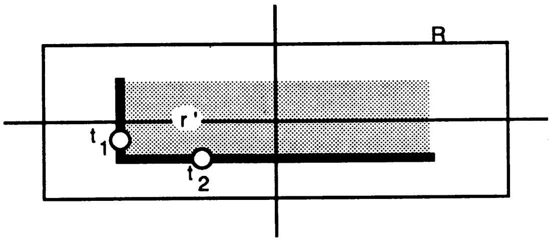

FIG. 5. Two supporting points in the same quadrant.

—$\mathcal{C}(p)$ is the set of all points of $\mathcal{C}(M_3)$ with x_1-coordinate smaller than $x_1(p)$ and x_2-coordinate smaller than $x_2(p)$, i.e., the set of all $q \in \mathcal{C}(M_3)$ such that the rectangle with lower left corner p and upper right corner q does not contain any point of $M_2 \cup M_4$.

For each $p \in M_2$ [$p \in M_3$, $p \in M_4$, respectively], $x_1(p)$, $x_2(p)$, p_u, p_l, and $\mathcal{C}(p)$ are defined corresponding to the above.

3. AN OPTIMAL SOLUTION FOR THE ONE-DIMENSIONAL PROCESSOR ARRAY

We now turn to the implementation of the above algorithm on a one-dimensional processor array. Part I of the algorithm consists of an $O(n)$ time sorting procedure (see, e.g., [2]) and parallel recursive calls for two largest empty rectangle problems of size $n/2$ on processor arrays of size $n/2$, each. Thus, if we let $T(n)$ refer to the time complexity of the entire computation, we obtain

$$T(1) = O(1)$$
$$T(n) = T(n/2) + g(n) + c_1 n,$$

with $g(n)$ denoting the time complexity of Part II. With the same arguments, we obtain the same recurrence equation for $g(n)$, i.e.,

$$g(1) = O(1)$$
$$g(n) = g(n/2) + h(n) + c_2 n,$$

with $h(n)$ denoting the time complexity of Part III.

In order to implement Part III, we compute the sets M_1, M_2, M_3, and M_4 of maximal elements, and then for each case listed in Fig. 4 we compute the largest empty rectangle separately. The rectangle r^* as described in Theorem 1 is the largest of these at most 16 rectangles (for some cases listed in Fig. 4, there may not exist a corresponding rectangle). The following is a description of how for each type of these 16 cases the largest empty rectangle can be computed in linear time (see Fig. 7 for an illustration).

Cases of Type A. Consider, e.g., Case 3. From Lemma 3 we know that all pairs of supporting elements of M_3 [M_1] are close neighbors. After M_1, M_2, M_3, and M_4 are sorted by x_1-coordinate, each $p \in \mathcal{C}(M_3)$ is first shifted through M_2 and M_4 to compute $x_1(p)$ and $x_2(p)$ and then through $\mathcal{C}(M_1)$ to determine the corresponding largest empty rectangle containing no point of M_2 or M_4. Pipelining the shift processes yields linear running time.

Cases of Type B. Consider, e.g., Case 2. M_1, M_2, M_3, and M_4 are sorted by x_1-coordinate, and then each $p \in \mathcal{C}(M_3)$ is shifted through M_2 and M_4 to compute the respective supporting left point $l_2 \in M_2$ and supporting lower point $l_4 \in M_4$ (l_2 [l_4] is the rightmost [uppermost] point of M_2 [M_4] with smaller x_2-coordinate [x_1-coordinate]). Pipelining the shift processes yields linear running time, too.

Cases of Type C. Computing the largest empty rectangle for a case of type C is essentially the same, since for each

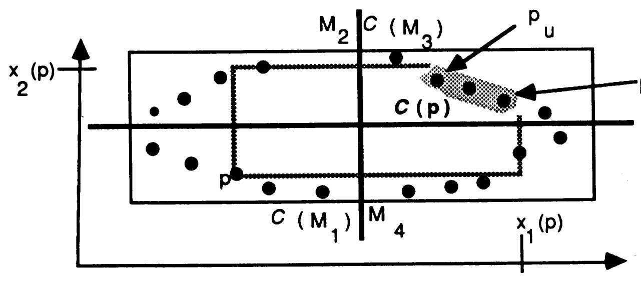

FIG. 6. Illustration of $x_1(p)$, $x_2(p)$, p_u, p_l, and $C(p)$.

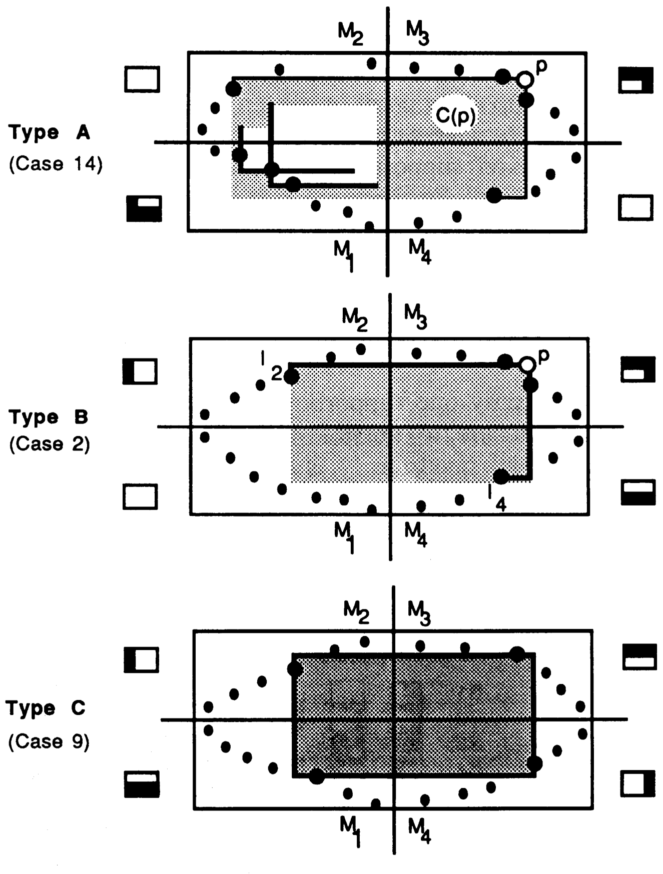

FIG. 7. Illustration of the determination of the largest empty rectangle for each case.

left [lower] supporting point in M_1 the three other supporting points are determined similarly.

Summarizing, we can also implement Part III with linear time complexity, i.e., $h(n) = O(n)$. Hence $g(n) = O(n)$ and, finally, $T(n) = O(n)$. Thus, we obtain

THEOREM 2. *The largest empty rectangle problem can be solved on a one-dimensional array of n processors in time $O(n)$ which is asymptotically optimal.*

4. AN OPTIMAL SOLUTION FOR THE TWO-DIMENSIONAL PROCESSOR ARRAY

On a two-dimensional processor array, sorting n points requires $O(\sqrt{n})$ steps (see, e.g., [8]). Thus, using the same notation as that in Section 2, we obtain

$$T(1) = O(1)$$
$$T(n) = T(n/2) + g(n) + c_1\sqrt{n}$$

and

$$g(1) = O(1)$$
$$g(n) = g(n/2) + h(n) + c_2\sqrt{n},$$

with $h(n)$ denoting the time complexity of Part III.

In order to implement Part III, we compute the sets M_1, M_2, M_3, and M_4 of maximal elements as described in [4] and then sort them by x_1-coordinate in time $O(\sqrt{n})$ (see, e.g., [10]). Then, for each case listed in Fig. 4, we again compute the largest empty rectangle separately. However, it is not possible to implement each of these steps by a sequence of shift operations as described in Section 3; this would not result in an $O(\sqrt{n})$ time algorithm. For cases of type A, one more divide and conquer step is necessary.

Let us again consider Case 14 as an example of a type A case. For each point $p \in \mathcal{C}(M_3)$ let p' denote the point of $C(p)$ such that the rectangle with upper right corner p and lower left corner p' is larger than all other empty rectangles with upper right corner p and lower left corner $p'' \in C(p)$. (To simplify exposition we assume that for each $p \in \mathcal{C}(M_3)$ there are no two $p', p'' \in C(p)$ such that the empty rectangle with upper right corner p and lower left corner p' and p'', respectively, have the same area. The existence of such points does not change the remaining significantly.)

LEMMA 4. *Let $p, q \in \mathcal{C}(M_3)$ and $p', q' \in \mathcal{C}(p) \cap C(q)$; then $p[1] < q[1] \Rightarrow p'[1] < q'[1]$.*

Proof [6, 7]. Assume that p' has smaller x_1-coordinate than q' and consider the areas a, b, c, d, e, f, g of the rectangles depicted in Fig. 8. We obtain $c + d + e > d + e + f + g$ and $b + d + f > a + b + c + d \Rightarrow c > f + g$ and $f > a + c \Rightarrow c - g > f > c + a$. This is a contradiction, since a, b, c, d, e, f are all positive. ■

Lemma 4 leads to an efficient $O(\sqrt{n})$ time algorithm for solving cases of type A. Before we proceed with describing the final solution, we first introduce a procedure *MAXBELOW* which will be used in the remainder.

Assume that on a $\sqrt{n} \times \sqrt{n}$ processor array each processor has three registers X, Y, and Z (for positive numbers) and that all processors are divided into two sets P' and P''. Procedure MAXBELOW(P', P'', X, Y, Z) computes (and stores into register X) for each $p' \in P'$ the maximum contents of the registers Y of all processors $p'' \in P''$ whose register Z is smaller than the register Z of processor p' (or zero if no such processor p'' exists).

The MAXBELOW procedure can be performed in time $O(\sqrt{n})$ as follows:

—Sort with respect to the contents of the Z register, and in snake-like ordering as described in [10].

—Each row of PEs performs a cyclic shift of $\sqrt{n}$ steps and computes for each processor the maximum Y register of all processors in the row, as well as the maximum Y register of all processors in the row with smaller Z register.

—Each column of PEs performs a cyclic shift of $\sqrt{n}$ steps and computes for each processor the maximum Y register of all rows with smaller Z register.

We are now able to compute the largest empty rectangle for each case of types A, B, and C in time $O(\sqrt{n})$ as follows:

Cases of Type A. Again consider Case 14 (see Fig. 7) and assume that $|\mathcal{C}(M_3)| \geq |\mathcal{C}(M_1)|$; otherwise exchange M_1 and M_3 for the remainder.

Choose a point $p \in \mathcal{C}(M_3)$ with median x_1-coordinate, i.e., $|A| - |B| \leq 1$, where $A := \{q \in \mathcal{C}(M_3) \mid q[1] < p[1]\}$ and $B := \{q \in \mathcal{C}(M_3) \mid q[1] > p[1]\}$. The point p, the sets A and B, as well as the corresponding points $p_u, p_l \in \mathcal{C}(M_1)$ and the range $C(p)$ as defined in Section 2, are depicted in Fig. 9.

Consider the point $p' \in \mathcal{C}(M_1)$ inducing a maximum area empty rectangle with lower left corner $p' \in \mathcal{C}(M_1)$ and upper right corner $p \in \mathcal{C}(M_3)$.

With C, D, E, and F denoting the subsets of $\mathcal{C}(M_1)$ depicted in Fig. 9, we observe that for each $q \in A$, $C(q) \subseteq T \cup U \cup V$, and for each $q \in B$, $C(q) \subseteq S \cup T \cup U$. However, from Lemma 4 it follows that for each $q \in A$ $[q \in B]$ it is impossible that $q' \in U$ $[q' \in T$, respectively$]$. Thus, for each $q \in A$ $[q \in B]$ we have $q \in T \cup V$ $[q \in S \cup U]$. Hence, the largest empty rectangle with respect to Case 14 can be computed as follows:

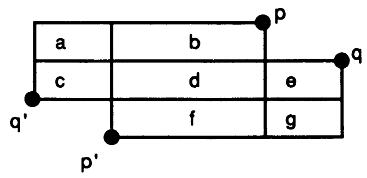

FIG. 8. Illustration of the proof of Lemma 4.

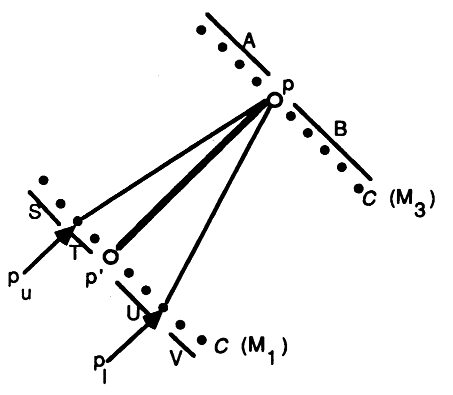

FIG. 9. Illustration of the determination of the largest empty rectangle for Case 14.

—For all $p \in \mathcal{C}(M_3)$ the points p_u, $p_l \in \mathcal{C}(M_1)$ are determined in time $O(\sqrt{n})$ using the MAXBELOW procedure described above.

—A point $p \in \mathcal{C}(M_3)$ with median x_1-coordinate and its corresponding point $p' \in \mathcal{C}(M_1)$ is computed in time $O(\sqrt{n})$ by sorting $\mathcal{C}(M_3)$ and broadcasting p through $\mathcal{C}(M_1)$.

—The problem is recursively solved for A, T, V and for B, S, U in parallel.

—The largest empty rectangles computed for these subproblems are compared with the rectangle induced by p and p'; the largest one is chosen as the final result.

Let $t(n)$ denote the time complexity for computing the largest empty rectangle with respect to Case 14 as described above. Then

$$t(1) = O(1)$$

$$t(n) = t(3n/4) + c\sqrt{n};$$

hence,

$$t(n) = O(\sqrt{n}).$$

Cases of Type B. Consider, e.g., Case 2 (see Fig. 7). For each $p \in \mathcal{C}(M_3)$ the respective supporting left point $l_2 \in M_2$ and supporting lower point $l_4 \in M_4$ is the rightmost [uppermost] point of M_2 [M_4] with smaller x_2-coordinate [x_1-coordinate]. Using the MAXBELOW procedure, in $O(\sqrt{n})$ time both these points can be computed in parallel for all $p \in \mathcal{C}(M_3)$ and the rectangles induced by these points can be tested to determine whether they contain points of M_1. Hence, the maximum area of these rectangles can be determined in time $O(\sqrt{n})$.

Cases of Type C. Computing the largest empty rectangle for a case of type C is essentially the same.

Received March 12, 1987; revised February 2, 1989

Summarizing, we obtain that Part III can be implemented on a two-dimensional processor array with a time complexity of

$$h(n) = O(\sqrt{n}).$$

Hence, $g(n) = O(\sqrt{n})$ and, finally, $T(n) = O(\sqrt{n})$.

THEOREM 3. *The largest empty rectangle problem can be solved on a two-dimensional array of $\sqrt{n} \times \sqrt{n}$ processors in time $O(\sqrt{n})$ which is asymptotically optimal.*

5. CONCLUSION

In this paper we have presented $O(n)$ and $O(\sqrt{n})$ time, respectively, parallel algorithms for solving the largest empty rectangle problem for a set of n points on one- and two-dimensional processor arrays of size n. Since, on these architectures, comparing two arbitrary data items takes (in the worst case) at least time $O(n)$ and time $O(\sqrt{n})$, respectively, the algorithms described are asymptotically optimal.

REFERENCES

1. Atallah, M. J., and Fredrickson, G. N. A note on finding a maximum empty rectangle. *Discrete Appl. Math.* **10** (1986).
2. Chazelle, B. Computational geometry on a systolic chip. *IEEE Trans. Comput.* **33,** 9 (1984), 774–785.
3. Chazelle, B., Drysdale, R. L., and Lee, D. T. Computing the largest empty rectangle. *Proc. Symposium on Theoretical Aspects of Computer Science,* 1984, pp. 43–54.
4. Dehne, F. $O(n^{1/2})$ algorithms for the maximal elements and ECDF searching problem on a mesh-connected parallel computer. *Inform. Process. Lett.* **22** (1986), 303–306.
5. Dehne, F. Parallel computational geometry and clustering methods. Ph.D. thesis, University of Würzburg (West Germany), 1986; also available as Tech. Rep. No. 104, School of Computer Science, Carleton University, Ottawa, Ontario, Canada K1S 5B6.
6. McKenna, M., O'Rourke, J., and Suri, S. Finding the largest rectangle in an orthogonal polygon. Report JHU/EECS-85/09, Johns Hopkins University, Baltimore, MD, 1985.
7. McKenna, M., and Suri, S. Private communication.
8. Miller, R., and Stout, Q. F. Computational geometry on a mesh-connected computer. *Proc. International Conference on Parallel Processing,* 1984, pp. 66–73.
9. Naamad, A. W., Hsu, W. L., and Lee, D. T. On the maximum empty rectangle problem. *Discrete Appl. Math.* **8** (1984).
10. Thompson, C. D., and Kung, H. T. Sorting on a mesh-connected parallel computer. *Comm. ACM* **20,** 4 (1977), 263–271.

FRANK DEHNE was born in Hannover, West Germany, in 1960. He received the M.C.S. degree (Dipl. Inform.) from the Technical University of Aachen (West Germany) in 1983 and the Ph.D. degree (Dr. rer. nat.) from the University of Würzburg (West Germany) in 1986. Since July 1986 he has been an assistant professor at the School of Computer Science of Carleton University, Ottawa, Canada. His research interests include computational geometry, data structures, and parallel algorithms and VLSI.