Sādhanā, Vol. 17, Part 1, March 1992, pp. 221–236. © Printed in India.

Fast parallel algorithms for the maximum empty rectangle problem

AMITAVA DATTA, R SRIKANT, G D S RAMKUMAR and KAMALA KRITHIVASAN
Department of Computer Science and Engineering, Indian Institute of Technology, Madras 600 036, India

Abstract. We present efficient parallel algorithms for the maximum empty rectangle problem in this paper. On CREW PRAM, we solve the area version of this problem in $O(log^2 n)$ time using $O(nlogn)$ processors. The perimeter version of this problem is solved in $O(logn)$ time using $O(nlog^2 n)$ processors. On EREW PRAM, we solve both the problems in $O(logn)$ time using $O(n^2/logn)$ processors. We also present an $O(logn)$ time algorithm on a mesh-of-trees architecture.

Keywords. Efficient parallel algorithms; maximum empty rectangle problem; mesh-of-trees architecture.

1. Introduction

The *Maximum Empty Rectangle* (MER) problem is the following. Given an isothetic rectangle BR and a point set P contained in BR, we have to find out the maximum area/perimeter isothetic rectangle in BR which does not contain any point from the set P. This problem has been extensively studied recently (Namaad *et al* 1984; Mckenna *et al* 1985; Chazelle *et al* 1986; Atallah & Fredrickson 1986; Aggarwal *et al* 1987; Aggarwal & Suri 1987, pp. 278–290, 1989; Atallah & Kosaraju 1989; Orlowski 1990; Datta 1991) in the sequential domain. An $O(n^2)$ time and $O(n)$ space algorithm was given by Namaad *et al* (1984), where n is the cardinality of the point set P. Later Chazelle *et al* (1986) obtained an $O(nlog^3 n)$ time and $O(nlogn)$ space algorithm. Mckenna *et al* (1985) provide an algorithm which solves the perimeter version of this problem in $O(nlog^4 n)$ time and $O(n)$ space. The sequential lower bound for this problem is $\Omega(nlogn)$ (Aggarwal & Suri 1989). The best existing sequential algorithm is by Aggarwal & Suri (1987, pp. 278–290, 1989). Their algorithms run in $O(nlog^2 n)$ and optimal $O(nlogn)$ time for the area and perimeter cases respectively. Both the algorithms use optimal $O(n)$ space. There are several algorithms which solve this problem by enumerating all candidate empty rectangles (Atallah & Kosaraju 1989; Orlowski 1990; Datta 1991). The expected and worst case running times of these algorithms are $O(nlogn)$ and $O(n^2)$, respectively, using $O(n)$ space. This problem has also been solved in three dimensions (Datta & Krithivasan 1991). Recently, Aggarwal *et al* (1989) have provided an efficient parallel algorithm for computing the maximum empty rectangle. On CRCW PRAM model, their algorithm for the area problem runs

in $O(log^2 n)$ time using $O(nlogn)$ processors. For the CREW PRAM, the algorithm takes $O(log^2 nloglogn)$ time using $O(nlogn/loglogn)$ processors. Aggarwal *et al* (1989) also present an algorithm to solve the perimeter version of the MER problem in $O(log^2 n)$ time using $O(n)$ processors on a CREW PRAM.

In this paper, we present parallel algorithms for solving both the perimeter and area versions of the maximum empty rectangle problem. We present a parallel algorithm to solve the maximum area empty rectangle problem in $O(log^2 n)$ time using $O(nlogn)$ processors on a CREW PRAM. This algorithm improves on the time complexity of the algorithm of Aggarwal *et al* (1989), keeping the processor-time product unchanged. We also solve the maximum perimeter empty rectangle problem in $O(logn)$ time using $O(nlog^2 n)$ processors on a CREW PRAM. This algorithm is optimal in terms of time, but it is worse than the algorithm of Aggarwal *et al* (1989) in terms of the processor-time product. We then propose a new characterization of the MER problem and use it to solve this problem on EREW PRAM and mesh-of-trees architectures. Our algorithm on EREW PRAM runs in $O(logn)$ time using $O(n^2/logn)$ processors. Though the processor-time product is rather high, the time achieved is optimal on this model. The algorithm for the mesh-of-trees architecture runs in optimal $O(logn)$ time. Part of this work appeared previously in Datta & Krithivasan (1990, pp. 344–345). We assume some familiarity of the reader with Aggarwal & Suri (1987, pp. 278–290, 1989) and Chazelle *et al* (1986).

The remaining part of this paper is organized as follows. In § 2, we give some definitions. Section 3 contains the CREW PRAM algorithms. In § 4, we give the new characterization and use it to solve the MER problem on an EREW PRAM. In § 5, we discuss the mesh-of-trees algorithm. Finally, § 6 concludes the paper.

2. Definitions

An *isothetic rectangle* has its sides parallel to either the X or the Y axis. By a rectangle we always mean an isothetic rectangle. We denote the four sides of the enclosing rectangle BR by BR.left, BR.top, BR.right and BR.bottom. The cardinality of the point set P is n. We assume for ease of exposition that the points in the set P are in general position, i.e., no two points have the same X or Y coordinate. The X and Y coordinates of the point p_i are denoted by $p_i.x$ and $p_i.y$ respectively. We say a point p_i in P is a support for a side S of a rectangle R if S passes through p_i. Similarly, a side of BR, e.g., BR.left is the left support of a rectangle R if R.left and BR.left overlaps and R.left has length equal to or less than BR.left. A rectangle R is called a *Restricted Rectangle* (RR), if its sides are supported by either a point in P or a side of BR and R does not contain any point from the set P. It is easy to see that the maximum empty rectangle is a member of the set of restricted rectangles. It has been proved in Namaad *et al* (1984), that the number of restricted rectangles is at most $O(n^2)$. We use some terminology from Chazelle *et al* (1986). We refer to the maximum empty rectangle (MER) problem sometimes by the name of largest empty rectangle (LER) problem. Similarly, a subproblem in solving the LER problem is called the Largest Empty Corner Rectangle (LECR) problem.

The model of parallel computation PRAM stands for Parallel Random Access Machine. In this model, there are p identical processors executing same instructions in parallel on different data values. The processors can access a common shared memory. If two or more processors can read or write a memory word simultaneously,

the model of PRAM is called the Concurrent Read Concurrent Write (CRCW) PRAM. If only simultaneous reading is allowed, the model is called the Concurrent Read Exclusive Write (CREW) PRAM. In the weakest PRAM model, neither simultaneous read nor write is allowed. This is called the Exclusive Read Exclusive Write (EREW) PRAM. See the book by Gibbons & Rytter (1988) for more on PRAM algorithms. We use the following standard results from parallel algorithm literature.

Lemma 2.1. (Cole 1988) *A list of n elements can be sorted in O(logn) time on a CREW PRAM using O(n) processors.*

Lemma 2.2. (Kruskal *et al* 1985) *Given an array of integers* $A = \{a_1, a_2, \ldots, a_n\}$, *all the partial sums* $c_k = \Sigma_{j=1}^{K} a_j$, *i.e., the parallel prefix, can be computed on a CREW PRAM in O(logn) time using O(n/logn) processors.*

Both sorting and parallel prefix computation can be done on an EREW PRAM within the same processor and time complexities. Moreover, the parallel prefix computation can be done for any binary associative operator such as *max*, *min* etc.

The *nearest smallers* problem (Berkman *et al* 1988) is defined in the following way. The input to the problem is an array $A = \{a_1, a_2, \ldots, a_n\}$ of elements from a totally ordered domain. For each a_i, $1 < i < n$, find the nearest element to its left and the nearest element to its right, that are less than a_i, if such elements exist. In other words, for each $1 < i < n$, find the maximal $1 < j < i$ and the minimal $i < k < n$ such that $a_j < a_i$ and $a_k < a_i$.

Lemma 2.3. (Berkman *et al* 1988) *The nearest smallers problem can be solved on a CREW PRAM in O(logn) time using O(n/logn) processors.*

3. The CREW PRAM algorithms

3.1 *Background to the algorithm*

In this subsection, we develop some ideas which will be useful for the exposition of our algorithms for both the area and perimeter cases of the maximum empty rectangle problem.

Our algorithm is based on the divide-and-conquer strategy used by Chazelle *et al* (1986) for solving the MER problem in the sequential domain. The restricted rectangles whose two opposite sides are supported by the sides of the bounding rectangle BR can be found easily. We consider the case when the left and right sides of an RR are supported by BR.left and BR.right respectively. Notice that the top and bottom sides of such a rectangle should be supported by two consecutive points in the Y sorted order. The points in the set P can be sorted in the Y order in $O(logn)$ time using $O(n)$ processors on a CREW PRAM (Cole 1988). After this the area/perimeter of the RR can be found in $O(logn)$ time by $O(n/logn)$ processors by grouping them in $O(n/logn)$ groups and doing the computation sequentially within a group. The time and processor requirement is well within the overall complexity of our algorithm. The other type of RR whose top and bottom supports are BR.top and BR.bottom, respectively, can also be found in a similar way. From now on we will not consider these RR in our discussion.

Let v_l be a vertical line that splits the points in the set P into two roughly equal

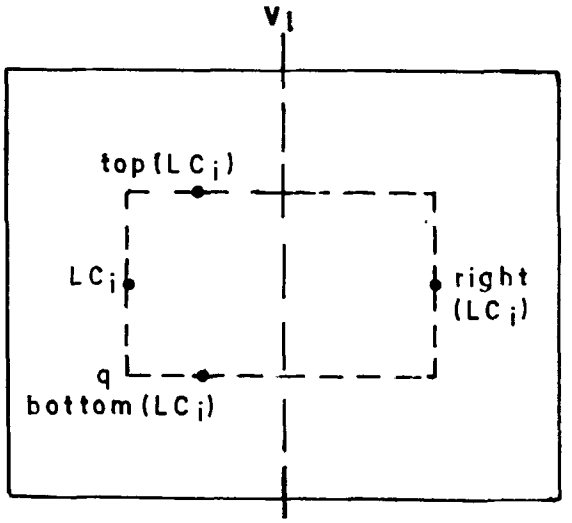

Figure 1. Some definitions related to the algorithm.

halves, namely $CL = \{LC_1, \ldots, LC_s\}$ and $CR = \{RC_1, \ldots, RC_t\}$. Let the supports of the top, bottom, left and right edges of a restricted rectangle R be top_sup(R), bottom_sup(R), left_sup(R) and right_sup(R) respectively. In the divide-and-conquer approach in Chazelle *et al* (1986), the problem is solved separately in the two sets CL and CR. The restricted rectangles whose supports lie completely within one of the sets are found in this recursive step. Then the two solutions are merged to find the maximum empty rectangle which has supports from both CL and CR. We also follow a similar approach for our parallel algorithm. In the merge step, we consider three different types of rectangles, (a) with three supports from CL and one support from CR, (b) with two supports from CL and two from CR and (c) with one support from CL and three supports from CR. Since types (a) and (c) are symmetrical, we consider only type (a). Hence, in the merge step of the divide-and-conquer algorithm, we have to find the largest of all the candidate rectangles of types (a) and (b).

Let $bottom(LC_i)$ be the point in CL with the highest Y coordinate which is to the right of and below the point LC_i. Similarly, let $top(LC_i)$ be the point in CL with the lowest Y coordinate which is to the right of and above the point LC_i. Also, let $right(LC_i)$ be the point in CR which has the lowest X coordinate such that $\text{bottom}(LC_i).y < \text{right}(LC_i).y < \text{top}(LC_i).y$. In figure 1, for a point LC_i, the points $\text{bottom}(LC_i)$, $\text{top}(LC_i)$ and $\text{right}(LC_i)$ are illustrated.

3.1a *Type (a) rectangles*: First we consider a rectangle R of type (a). Clearly, the three supports of R that are from CL are the left, top and bottom supports. Let the left support be LC_i. Since the rectangle R is empty, top_sup(R) is the point from CL with the least Y coordinate which is above and to the right of LC_i. In other words, top_sup(R) is $\text{top}(LC_i)$. Similarly, bottom_sup(R) is $\text{bottom}(LC_i)$. The emptiness of R implies that right_sup(R) is the leftmost point from CR whose Y coordinate lies between those of $\text{top}(LC_i)$ and $\text{bottom}(LC_i)$. Consequently, right_sup(R) is the same as $\text{right}(LC_i)$. Hence, for each point LC_i, there is exactly one type (a) rectangle with LC_i as the left support. The other three supports are immediately fixed. So, there are at most $O(n)$ type (a) rectangles, of which the maximum area/perimeter rectangle has to be found.

3.1b *Type (b) rectangles*: A rectangle of type (b) can have either the left and top supports or the left and bottom supports from CL. Wihtout loss of generality, we consider only those type (b) rectangles with the left and bottom supports from CL (the other case being similar). From now on, we use the term type (b) rectangle to refer only to rectangles with the left and bottom supports from CL. If the left support of a type (b) rectangle R is LC_i, the emptiness of R implies that the bottom support of R will be bottom(LC_i). Unlike type (a) rectangles, however, the number of type (b) rectangles is not restricted to $O(n)$, since the top and right supports are not immediately fixed by the left and bottom supports.

First we transform the problem of finding the largest empty type (b) rectangle to that of finding the largest empty type (b) corner rectangle. A *corner rectangle* is a rectangle such that any two of its opposite corners coincide with points from a given set of points. Given a rectangle BR and a set of points P_1 inside it, the largest empty corner rectangle is therefore the maximum area/perimeter empty corner rectangle with respect to the set of points P_1. The transformation can be done in the following way. Consider a set of points CL' obtained as follows. Include all elements of CL in CL'. In addition, for every point LC_i in the set CL, include in CL' the point $(LC_i.x, \text{bottom}(LC_i).y)$, shown as the point q in figure 1. Similarly, we can obtain a new set of points CR' from CR. Let the elements of CL' be $LC_1', LC_2', \ldots, LC_r'$ and those of CR' be $RC_1', RC_2', \ldots, RC_s'$ in order of increasing X coordinate. A corner rectangle whose bottom-left corner is from CL' and top-right corner is from CR' is called a type (b) corner rectangle. Chazelle *et al* (1986) reduced the problem of finding the largest type (b) rectangle to that of obtaining the largest type (b) corner rectangle over the two sets CL' and CR'. We again use divide-and-conquer to find the largest type (b) corner rectangle. We divide the sets CL' and CR' into two subsets CL_1', CL_2' and CR_1', CR_2' respectively (where CL_1' lies above CL_2' and CR_1' lies above CR_2') using a horizontal line h_1 such that $|CL_1'| + |CR_1'|$ is approximately equal to $|CL_2'| + |CR_2'|$. A corner rectangle with one corner from CL_2' and the other corner from CR_1' is called a *special corner rectangle.* To compute the largest special corner rectangle, we use the arguments given in Chazelle *et al* (1986) and discard a point of CR_1' (respectively CL_2') that *dominates* (is dominated by respectively) some other point in this set. Similarly, for CL_1' (respectively CR_2') we discard a point p_i if there exists p_j from CL_1' (CR_2' respectively) such that $p_i.x < p_j.x$ and $p_i.y > p_j.y$ (respectively $p_i.x > p_j.x$ and $p_i.y < p_j.y$). Let $CL_2'' = \{q_1, q_2, \ldots, q_v\}$ and $CR_1'' = \{m_1, m_2, \ldots, m_u\}$ denote the sets after they have been trimmed such that q_i's and m_j's are in increasing order of X coordinates (which implies that they are in decreasing order of Y coordinates). To find the largest empty special corner rectangle, for each point q_i in CL_2'', we need to determine the set of points of CR_1'' that can be paired with it to form empty rectangles. Clearly, this is a contiguous set of m_j's and there exist indices $l(i)$ and $r(i)$ such that the set $\{m_{l(i)}, m_{l(i)+1}, \ldots, m_{r(i)}\}$ contains exactly the points of CR_1'' that can be paired with q_i to form an empty corner rectangle.

3.2 *The algorithm*

The divide-and-conquer strategy for the largest empty rectangle problem can be naturally parallelised. The parallel algorithm executes in two stages. In the first stage, all the subproblems for all the recursive steps are produced, ready to be solved. These subproblems originate at each step of the recursive division. Once all the subproblems are generated, all of them are solved in parallel. We first formalize the concept of

generating a subproblem, i.e., specify the input which should be provided for each subproblem.

We explain the input sequence with respect to the first stage of recursion, i.e., when the point set P is divided into two parts by the vertical line v_1. The largest empty rectangle (LER) problem is specified by giving a list of points of P sorted by the X coordinate. The largest empty type (b) rectangle (LETCR) problem is specified by giving the lists of points of CL and CR sorted by the Y coordinate. The largest empty special corner rectangle (LESCR) problem is specified by giving the lists of points of CL_1', CL_2', CR_1' and CR_2' each sorted by the Y coordinate. The inputs in the successive recursive stages are defined in a similar way. The subproblems are generated in the following way. First, we sort the points of P (in the subsequent stages of recursion, this set also gets divided into $n/2$, $n/4$... number of points) by increasing X coordinate. Now, we can easily generate all the LER problems in $O(logn)$ time using $O(n)$ processors. This is done in the following way. First we sort the points according to X coordinate in $O(logn)$ time using $O(n)$ processors (Cole 1988). Since the recursion has $O(logn)$ stages, the point set has to be divided into smaller point sets to provide for the input of each recursive stage. The boundaries for these smaller sets can be found by a single processor in $O(logn)$ time by a binary search. Each point in the set P can be the input of at most $O(logn)$ LER subproblems. We assign processor Pr_i for the point p_i. Pr_i creates a separate copy of p_i for each subproblem in which p_i participates. Clearly, each processor Pr_i takes $O(logn)$ time. So, the overall time and processor complexities for generating all LER subproblems is $O(logn)$ time and $O(n)$ processors. From now on, for each LER subproblem, we denote the input set by the generic name P. By the size of a subproblem, we mean the number of points involved in it.

Now, for each LER subproblem, we need to solve the corresponding LETCR and LESCR subproblems in order to find the largest empty type (b) rectangle. For each LER subproblem, we assign m processors. In $O(logn)$ time, for all the LER subproblems, the corresponding set P is sorted by the Y coordinate. For each LER subproblem of size m we can now generate all the LETCR subproblems using m processors in $O(logm)$ time. Again, for each LETCR subproblem of size k, we assign k processors. For each such LETCR subproblem, we can now easily generate all the LESCR subproblems in $O(logk)$ time using k processors. The details of the complexity of generation of these subproblems will be given later. Finally, in $O(logn)$ time we find the overall maximum rectangle among the maximum rectangles found in parallel for each subproblem. Later on in this paper, we discuss in detail the processor and time complexities resulting from these steps. We first provide a method to find the largest type (a) rectangle for a subproblem of the LER problem. This method is common to both the area and perimeter cases and uses $O(n)$ processors and $O(logn)$ time.

3.2a *Finding the largest type (a) restricted rectangle*: First we show how to obtain top(LC_i), bottom(LC_i) and right(LC_i). We assume that the sets of points CL and CR form the input. We sort the points of CL by increasing the Y coordinate using $O(n)$ processors and $O(logn)$ time (Cole 1988). Consider a point p_i in the sorted array. It is easy to see that top(p_i) is the first point in the array after p_i with an X coordinate greater than that of p_i. The problem of finding top(p_i) is analogous to the nearest smallers problem as defined in § 2. This problem has been solved by Berkman *et al* (1988) taking $O(logn)$ time using $O(n/logn)$ processors on a CREW PRAM. In the problem of finding top(p_i), the condition 'less than p_i' is replaced by the condition 'greater

than p_i'. This change can be accommodated by a minor modification in the nearest smallers algorithm. By a similar method, we obtain bottom(LC_i) for each point LC_i. Now, to find right(LC_i), a processor is assigned to each point LC_i. The processor assigned to LC_i has to find the leftmost point in CR whose Y coordinate is in the range top(LC_i) to bottom(LC_i). We build a binary search tree containing points from CR with the key as the Y coordinate. This tree can be built starting from the leaves upwards, in $O(logn)$ time using $O(n)$ processors. This is done in the fashion of the parallel merge sort algorithm of Cole (1988). At each interior node of the binary search tree, we store the point in the subtree which has the least X coordinate. Once the tree is built, the processor assigned to LC_i can perform binary search to find the leftmost point in the range top(LC_i) to bottom(LC_i). The leftmost point so obtained is the required right(LC_i). Having found top(LC_i), bottom(LC_i) and right(LC_i) for each LC_i, we now proceed to find the maximum area/perimeter type (a) rectangle as follows. For each point LC_i, there is exactly one rectangle R_i with left support LC_i. We already know the other three supports by the method described above. So in $O(1)$ time we can find the area/perimeter of the rectangle R_i. Hence, in $O(logn)$ time using $O(n)$ processors, we can find the maximum area/perimeter type (a) rectangle.

3.2b *Finding the largest type (b) rectangle*: We consider the method of finding the type (b) rectangle for a subproblem of the LER problem. First we show how to find the sets of points $CL_2'' = \{q_1, q_2, \ldots, q_v\}$ and $CR_1'' = \{m_1, m_2, \ldots, m_u\}$ from CL_2' and CR_1' respectively. Consider an array containing the Y coordinates of the points in CL_2'. A point LC_i' belongs to CL_2'' if there is no point to its right with a larger Y coordinate. Hence we obtain the set of points $CL_2'' = \{q_1, q_2, \ldots, q_v\}$ by solving the nearest smallers problem for this array. Similarly, we obtain the set of points $CR_1'' = \{m_1, m_2, \ldots, m_u\}$. Similarly, the other two sets CL_1'' and CR_2'' can also be found. For every point q_i, we now find the corresponding $l(i)$ ($r(i)$ is found using a similar method). Suppose, L_1 is an imaginary vertical line through q_i and the first point in the set CL_1'' to the right of L_1 is r_i (i.e., r_i is the first point having greater X coordinate than L_1). Now, consider a horizontal line L_2 through r_i. Let s_i be a point in CR_1'' such that it is the first point below L_2. It is easy to see that s_i is the point $l(i)$. For each point q_i in CL_2'', we can find the corresponding point r_i by a binary search in the set CL_1''. This is possible because the points in the set CL_1'' are already sorted according to both X and Y coordinates. This computation can be done by assigning one processor for each point. The overall complexity is $O(n)$ processors and $O(logn)$ time. The point s_i, i.e., $l(i)$ can also be found in a similar way. The point $r(i)$ is found by binary searches in CR_2'' and CR_1''.

For every point q_i, we now have to find the point in the range $m_{l(i)}, \ldots, m_{r(i)}$ with which it should be paired to give the maximum area/perimeter empty rectangles. Since the methods are different for the area and perimeter cases we present each method separately.

3.2c *Algorithm for the area problem*: *Lemma* 3.1. (Mckenna *et al* 1985) *Let q_i, q_j belong to CL_2'' with $i < j$ and let m_k, m_l belong to CR_1'' with $k < l$. Furthermore, assume that q_i and q_j can be individually paired with both m_k and m_l to form empty corner rectangles. Then, the sum of the areas of the rectangles formed by the corner pairs (q_i, m_l) and (q_j, m_k) is no more than the sum of the areas of the rectangles formed by the corner pairs (q_i, m_k) and (q_j, m_l).*

DEFINITION 3.2 (Aggarwal *et al* 1987)

Let M denote any $p \times q$ matrix containing real entries. Let $j(i)$ be the smallest column index j such that $M(i,j)$ equals the maximum value in the ith row of M. Aggarwal *et al* (1987) call M monotone if, for $1 \leqslant i_1 \leqslant i_2 \leqslant v$, we always have $j(i_1) \leqslant j(i_2)$. Furthermore, they call M totally monotone if every 2×2 submatrix of M is monotone. The row-maximum problem for a matrix requires the determination of the maximum value (or the leftmost maximum value if there are several maxima in a row) in every row of the matrix.

Consider a $v \times u$ matrix A that contains in location $A(i,j)$ the area of the rectangle formed by q_i as the lower-left corner and m_j as the upper-right corner when these points form an empty corner rectangle; otherwise, $-\infty$ is stored in $A(i,j)$. Now, $u + v \leqslant n$, and for $i \leqslant k$, since $l(i) \geqslant l(k)$ and $r(i) \geqslant r(k)$, A contains at most two sets of entries that are $-\infty$. The periphery of each set forms a staircase inside A and the two sets of entries lie in the top-left and bottom-right corners of A (figure 2). From lemma 3.1, if none of the entries of a 2×2 minor of A is $-\infty$, then this minor is monotone. Consequently, we call such a matrix a *monotone double-staircase matrix*, or simply an *mds-matrix*. We note that the largest area empty corner rectangle problem can be computed by solving the row maximum problem for A. A matrix of real entries is called a *single staircase matrix* if it contains one set of entries those are $-\infty$ and if the periphery of this set forms a staircase either in the top-left or bottom-right corner. A single staircase matrix is called *monotone single staircase matrix*, or simply an *mss matrix*, if its every 2×2 minor that does not contain any $-\infty$, is monotone.

DEFINITION 3.3

For any row i in a $v \times u$ mds matrix A, let next (i) be the last row j such that $r(j) \geqslant l(i)$. Then the list x is defined as follows: $x_1 = 1$, $x_i = \text{next}\,(x_{i-1})$, $i > 1$.

Lemma 3.4. *In the matrix A, consider the intervals $l(x_1),\ldots,r(x_1)$; $l(x_2),\ldots,l(x_1)-1$; $l(x_3),\ldots,l(x_2)-1$, etc. Then the non-zero part of any row never spans more than two intervals.*

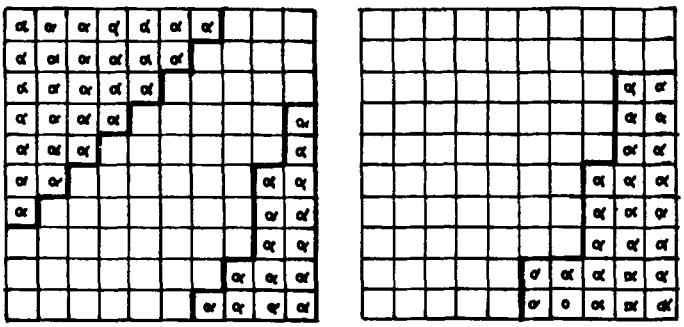

Figure 2. Two types of monotone matrices.

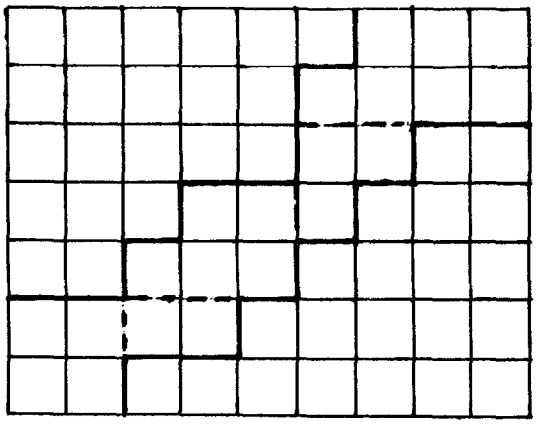

Figure 3. Partitioning of an mds matrix into mss matrices.

Proof. Let there be a row j such that the non-zero part of row j spans three intervals. Let the two nearest rows in the list x above and below j be x_k and x_{k+1} (where x_{k+1} may be j). Consider the intervals $l(x_k),\ldots,l(x_{k-1})-1$ and $l(x_{k+1}),\ldots,l(x_k)-1$. Now, $l(j)\geqslant l(x_{k+1})$. Since $r(x_k-1)<l(x_{k-1})$ and $x_k-1\leqslant j$, $r(j)<l(x_{k-1})$. Hence, the relevant entries of row j fall within the two intervals mentioned above.

Lemma 3.5. *By using $O(\log n)$ time and $O(n)$ processors, any $v\times u$ mds matrix A can be partitioned into at most $2n$ mss matrices, such that the non-zero entries of any row are in at most two mss matrices (see figure* 3).

Proof. The function next (i) can be found in $O(\log n)$ time using $O(n/\log n)$ processors by converting it into a nearest smallers problem (Berkman *et al* 1988). The intervals defined in the previous lemma can be found in $O(\log n)$ time using $O(n)$ processors by using list compression techniques. The rest of the proof is similar to the proof of lemma 2·3 in Aggarwal & Suri (1989).

Lemma 3.6. *The maximum problem for a $v\times u$ mss matrix A'' can be solved in $O(\log v\log u)$ time using $O(u/\log u)$ processors.*

Proof. It can easily be shown that an mss matrix is monotone. Now, we find the maximum of the middle row in $O(\log u)$ time using $O(u/\log u)$ processors. Let the position of the maximum be $j(v/2)$. We now know that the maximum for any row between 1 and v/w must lie between 1 and $j(v/2)$ and that the maximum for any row between $v/2$ and v must lie between $j(v/2)$ and u. Hence we can use $j(v/2)/\log u$ processors for the minor $A''(1\ldots(v/2)-1, 1\ldots j(v/2))$ and $(u-j(v/2))/\log u$ processors for the minor $A''((v/2)+1\ldots v, j(v/2)\ldots u)$. We end up with the recurrence equations

$$P(u,v)=P(u,v/2)+(u/\log u),$$

$$T(u,v)=T(u,v/2)+(\log u),$$

where $P(u,v)$ and $T(u,v)$ are the processor and time complexities respectively. The solutions of these two recurrences give the result stated in the lemma.

Theorem 3.7. *The maximum problem for a $v\times u$ mds matrix A can be solved in $O(\log u\log v)$ time using $O(v/\log v)$ processors.*

Proof. The proof follows directly from the previous lemmas.

3.2d *Algorithm for the perimeter problem*: Consider a matrix A' whose (i,j)th entry is $-\infty$ if and only if the (i,j)th entry of the matrix A, defined in the previous section, is $-\infty$. Otherwise, $A'(i,j)$ contains the perimeter of the rectangle with its bottom-left corner as q_i and top-right corner as m_j. Clearly, the maximum entry in A' corresponds to the pair of points that form the largest perimeter empty rectangle. Also, the perimeter of the rectangle formed by q_i and m_j equals $2(y_j + x_j) - 2(y_i + x_i)$ and $(y_i + x_i)$ is a constant throughout the ith row of A'. Hence, if we split the matrix column-wise such that the finite part of each row is split into at most two parts (figure 3), we could do a parallel prefix computation to obtain partial maxima in each section and then find the maximum in each row in constant time.

Theorem 3.8. *The row-maximum problem for the matrix A' can be solved in $O(\log n)$ time using $O(n)$ processors.*

Proof. From lemma 3.5 it follows that the required intervals can be found in $O(\log n)$ time using $O(n)$ processors. The partial maxima in each interval can then be found by doing a parallel prefix computation in $O(\log n)$ time using $O(n/\log n)$ processors. Once the partial maxima in each interval is available, the maximum of any one row can be found in constant time and the row-maximum problem for the matrix can be solved in $O(\log n)$ time using $O(n \log n)$ processors. The overall complexity is $O(\log n)$ time using $O(n)$ processors.

3.3 *Complexities of the algorithms*

As described in § 3.2, the parallel algorithm executes in two stages. In the first stage, all the subproblems due to all the recursive calls are generated. In the second stage, all the subproblems are solved in parallel. We will now analyse the number of processors required to solve these subproblems in parallel.

3.3a *The perimeter problem*: In the case of the perimeter problem, let $A(n)$ be the number of processors required to solve the problem. The merge step consists of sorting which requires n processors, finding the largest empty type (a) rectangle which takes n processors and finding the largest empty type (b) rectangle which takes, say, $B(n)$ processors. Since the two subproblems and the merge step are performed in parallel, the recurrence of $A(n)$ is as follows,

$$A(n) = 2^* A(n/2) + B(n) + n. \tag{1}$$

Let $C(n)$ be the number of processors required to solve the largest empty special corner rectangle (LESCR) problem. To solve the largest empty type (b) corner rectangle (LETCR) problem, $O(n)$ processors are required for sorting and $C(n)$ processors are required for the LESCR problem. Hence, the recurrence for $B(n)$ is,

$$B(n) = 2^* B(n/2) + C(n) + n. \tag{2}$$

We have already shown that $O(n)$ processors and $O(\log n)$ time are required to solve the LESCR perimeter problem. Hence, $C(n)$ is n. From (2), $B(n)$ is $O(n \log n)$ and further,

from (1), $A(n)$ is $O(nlog^2 n)$. Hence, our algorithm for the perimeter case takes $O(nlog^2 n)$ processors and $O(logn)$ time.

3.3b *The area problem*: The complexity analysis for the area problem is similar to that for the perimeter problem. Equation (1) remains the same. But to solve the LETCR problem, we sort using $O(n/logn)$ processors and $O(log^2 n)$ time. Hence, (2) is modified as follows,

$$B(n) = 2*B(n/2) + C(n) + n/logn. \tag{3}$$

The LETCR problem for the area case was solved in $O(log^2 n)$ time using $O(n/logn)$ processors. Substituting $C(n) = n/logn$ in (3), we obtain

$$B(n) = n/logn + 2*(n/2)/log(n/2) + 4*(n/4)/log(n/4) + ...$$

or,

$$B(n) = n(1/logn + 1/(logn - 1) + 1/(logn - 2) + \cdots + (1/2 + 1),$$

or,

$$B(n) = nH(logn), \quad \text{where } H(m) \text{ is the harmonic function.}$$

It is well known that $H(m)$ is $O(logm)$ as m tends to infinity. Hence, $B(n)$ is $O(nH(logn))$ or $O(nloglogn)$. So, from (1), $A(n)$ is $O(nlognloglogn)$. The time complexity remains $O(log^2 n)$. Hence, our algorithm for the area case seems to use $O(nlognloglogn)$ processors and $O(log^2 n)$ time. Now, we reduce the processor complexity to $O(nlogn)$ without affecting the time complexity.

3.3c *Reallocation of the processors for the area problem*: We now show how to reallocate the processors in the area problem to reduce the processor requirement to $O(nlogn)$. First, we prove that the total amount of work done in the algorithm is $O(nlog^3 n)$. The work done in solving the LETCR problem is $(n/logn)*(log^2 n)$, or $nlogn$. Using recurrence equations similar to (1) and (2) with $C(n) = (nlogn)$, it is easy to prove that the total work done is $O(nlog^3 n)$.

The parallel algorithm works in $t = log^2 n$ time intervals which we call stages. At the ith stage, let C_i be the number of processors required by the algorithm. From § 3.3b, we know that $max\{C_i | 1 < i < t\} = nlognloglogn$. Since the total work done is $nlog^3 n$, $C_1 + C_2 + \cdots + C_t = nlog^3 n$. Let us assume that we have only $O(nlogn)$ processors. At the ith stage, the algorithm requires C_i processors. According to Brent's theorem (Gibbons & Rytter 1988), the ith stage can be executed in $upper(C_i/nlogn)$ time with $O(nlogn)$ processors, where $upper(x)$ is the lowest integer greater than or equal to the real number x. Hence, the total time taken is,

$$\begin{aligned}
&upper(C_1/nlogn) + upper(C_2/nlogn) + \cdots + upper(C_t/nlogn) \\
&\quad < C_1/nlogn + 1 + C_2/nlogn + 1 + \cdots + C_t/nlogn + 1 \\
&\quad < (C_1 + C_2 + \cdots + C_t)/nlogn + t \\
&\quad < nlog^3 n/nlogn + log^2 n \\
&\quad = O(log^2 n).
\end{aligned}$$

Hence, the time complexity of the algorithm is still $O(log^2 n)$, while the processor complexity is reduced to $O(nlogn)$. So, we can solve the area case of the maximum empty rectangle problem in $O(log^2 n)$ time using $O(nlogn)$ processors.

4. The EREW PRAM algorithm

4.1 *A new characterization*

From now on we consider the set $P = \{p_1, p_2, \ldots, p_n\}$ sorted according to the X coordinate. R is the bounding rectangle. Given two points p_i and p_j such that $p_i.x < p_j.x$, we define the following sets.
Above $(p_i, p_j) = \{p_k | p_k.y > p_i.y$ and $p_i.x < p_k.x < p_j.x\}$.
Below $(p_i, p_j) = \{p_k | p_k.y < p_i.y$ and $p_i.x < p_k.x < p_j.x\}$.
Above$(p_i$, R.right$) = \{p_k | p_k.y > p_i.y$ and $p_i.x < p_k.x <$ R.right$\}$.
Below$(p_i$, R.right$) = \{p_k | p_k.y < p_i.y$ and $p_i.x < p_k.x <$ R.right$\}$.
Above (R.left, p_i) $= \{p_k | p_k.y > p_i.y$ and R.left $< p_k.x < p_i.x\}$. Below (R.left, p_i) is defined similarly. There is a unique point $p_m \in$ Above(p_i, p_j) such that $p_m.y < p_n.y$ for all $p_n \in$ Above(p_i, p_j). We denote this point p_m by $MA(p_i, p_j)$. Similarly, there is a unique point $p_q \in$ Below(p_i, p_j) such that $p_q.y > p_n.y$ for all $p_n \in$ Below(p_i, p_j). Such a point p_q is called $MB(p_i, p_j)$. We classify all RR in three categories. (i) *Type* 1 – Left support is R.left and right support is R.right. (ii) *Type* 2 – Left support is R.left and right support is a point $p_i \in P$. (iii) *Type* 3 – Left support is a point $p_i \in P$ and the right support is either a point $p_j \in P$ or R.right. It is well-known that there are $O(n)$ RR of type 1 and type 2 and $O(n^2)$ RR of type 3. Consider two points p_i and p_j with $p_i.x < p_j.x$.

Property 1. *If an RR R_i of type 3 exists with p_i as left support and p_j as right support, the top and bottom supports of R_i are $MA(p_i, p_j)$ and $MB(p_i, p_j)$ respectively.*

Proof. Suppose some other point p_k is the top support and $p_m = MA(p_i, p_j)$. From definition, $p_k.y > p_m.y > p_i.y$ and at least p_m will be within R_i. Hence R_i is not an RR. The proof for the bottom support is similar.

COROLLARY

If either Above (p_i, p_j) or Below (p_i, p_j) or both are empty, the top and bottom supports are R.top and R.bottom respectively.

Suppose, $p_k = MA(p_i, p_j)$ and $p_m = MB(p_i, p_j)$.

Property 2. *An RR R_i exists with p_i, p_k, p_j and p_m as left, top, right and bottom supports, iff $p_k.y > p_j.y > p_m.y$.*

Proof. Simple.
These two properties together characterize the type 3 RR.

Property 3. *The top and bottom supports for a type 2 RR with p_i as the right support are $MA(R.left, p_i)$ and $MB(R.left, p_i)$.*

4.2 *The EREW PRAM algorithm*

We enumerate and compute the three types of rectangles separately.

Algorithm MER. Procedure type 1: First sort the points in P in y order by the algorithm

in Cole (1988) and find out the RR. This can be done with $O(n)$ processors and $O(\log n)$ time.

Procedure type 2: (1) Sort the points in P according to X coordinates. This takes $O(n)$ processors and $O(\log n)$ time (Cole 1988).
(2) For each point $p_i \in P$, find $MA(p_i, p_j)$ and $MB(p_i, p_j)$ for $j = i+1, \ldots, n$, R.right.

Since max and min are binary associative operators, this can be done by $O(n/\log n)$ processors and $O(\log n)$ time by the algorithm for parallel prefix computation in Kruskal *et al* (1985). This computation finds all RR with p_i as the left support. For each p_i allocate $O(n/\log n)$ processors. So, this step can be done by $O(n^2/\log n)$ processors in $O(\log n)$ time. For each pair p_i, p_j which form an RR, i.e., satisfy property 2, compute the area.

Procedure type 3: (1) Sort the points in P according to X coordinates.
(2) For each $p_i \in P$, we can find MA(R.left, p_i) and MB(R.left, p_j). Compute the area for the RR. This step can be done within our processor and time bound.

At the end of the execution of these three procedures, we have $O(n^2)$ RR and we can find out the MER with $O(n^2/\log n)$ processors in $O(\log n)$ time.

End of algorithm MER.
The algorithm can be modified easily when the points are not in general position. So, we state the following.

Lemma 4.1. The MER problem can be solved in $O(\log n)$ time with $O(n^2/\log n)$ processors on a CREW PRAM.

Now, we show how to modify this algorithm to run within the same processor and time bound in the EREW PRAM.

Notice that, sorting can be done in $O(\log n)$ time using $O(n)$ processors (Cole 1988) and parallel prefix of n elements can be found in $O(\log n)$ time using $O(n/\log n)$ processors (Kruskal *et al* 1985) on an EREW PRAM. But in procedure type 3, when we compute parallel prefix starting at each p_i and upto R.right simultaneously, there may be read conflicts. To avoid this, after sorting the sequence according to X coordinates, we create n copies of this sorted sequence. Then the parallel prefix computation starting at each point p_i in procedure type 3 is done on a different sorted sequence. This avoids read conflicts.

Lemma 4.2. We can create n copies of a sequence of n elements in $O(\log n)$ time using $O(n^2/\log n)$ processors on an EREW PRAM.

Proof. We assign $O(n/\log n)$ processors for each element p_i of the sequence. At the first instant, one processor reads p_i and writes it. So at the second instant, two processors can read two copies of p_i and write them. At the end of the second instant, four copies of p_i are available. Proceeding in this way, within $O(\log n)$ time $n/\log n$ copies of p_i will exist. Then onwards, $n/\log n$ processors create $n/\log n$ copies of p_i at each instant. So, within overall $O(\log n)$ time, p_i can be copied n times. The total processor requirement for copying the complete sequence is $O(n^2/\log n)$.

Obviously, the overall storage requirement is $O(n^2)$. So, we state the following.

Theorem 4.3. *The MER problem can be solved in $O(\log n)$ time using $O(n^2/\log n)$ processors on an EREW PRAM.*

5. The mesh-of-trees algorithm

The mesh of trees architecture (Ullman 1984) is a square grid of n^2 processors without any interconnection between them. There are n rows and n columns in this grid and n is assumed to be a power of 2. Each row and each column is connected as a complete binary tree. For details see Ullman (1984). Our algorithm for the MER problem again consists of recognizing type 1, type 2 and type 3 RR. We first sort the points according to increasing X coordinates before starting the main algorithm. This takes $O(\log n)$ time (Ullman 1984; Lodi & Pagli 1985). After the sorting, we set up the points $p_1, p_2, \ldots, p_n$ in each row of the mesh. This can be done in $O(\log n)$ time. We first describe how to perform the partial maxima computation in the mesh of trees.

5.1 *Partial maxima on a mesh-of-trees*

Let there be n numbers $a_1, a_2, \ldots, a_n$ in each row of a mesh of trees. At the ith row, we want to compute the partial maxima starting from a_i upto a_n. The maxima will be respectively $max(a_{i+1})$, $max(a_{i+1}, a_{i+2})$, $max(a_{i+1}, a_{i+2}, a_{i+3}), \ldots, max(a_{i+1}, a_{i+2}, \ldots, a_{n-1})$. We describe the algorithm for the ith row. After the computation is over, the leaves will get the partial maxima values. Each internal node n_i of the tree has three registers. ML and MR contain the maxima from the left and right subtrees respectively and MU gets a maxima from the father of n_i. The numbers $a_1, a_2, \ldots, a_n$ are stored in the leaves and the leaves $a_{i+1}, a_{i+2}, \ldots, a_{n-1}$ participate in the computation. The other leaves in the ith row are masked. The active leaves first find out the root of their subtree. This can be done in the following way. In the first phase, all active leaves send a signal up the tree. All the internal nodes at a particular level will receive this signal simultaneously. Whenever an internal node gets this signal, it remembers whether it got it from its left son, right son or both. If the root gets the signal from both of its sons, it is also the root of the active subtree. Otherwise, it initiates the second phase. The root sends a signal back to its son which sent its signal in the first phase. This process continues until this signal reaches a node n_i which received signals from both of its sons in the first phase. This internal node n_i is the root of the active subtree. This computation takes $O(\log n)$ time. From now on, by root we will always mean the root of the subtree of the active leaves. Initially, all the active leaves send their data to their respective fathers. The computation done at each time instant at an internal node is divided into two parts. In the first part, when an internal node gets the values in its ML and MR registers, it sends max(ML, MR) to its father. In the second part, when it gets the value in its MU register from its father, it sends MU down to its left son and max(MU, ML) to its right son. The root does not get the value MU. When it gets the ML value from its left son, it sends this to its right son. The root sends a value $-\infty$ to its left son, so that this value does not influence the partial maxima computation. The leaves $a_{i+1}, a_{i+2}, \ldots, a_{n-1}$ will get the respective partial maxima at the end of this computation. The overall time requirement is $O(\log n)$.

5.2 *The algorithm for MER*

The n points $p_1, p_2, \ldots, p_n$ and R.right are sorted according to increasing x coordinate. We assume that $n + 1 = 2^k$ for $k > 0$. The sorting can be done in $O(\log n)$ time (Ullman

1984; Lodi & Pagli 1985). After the sorting, we set up this sequence in each row of the mesh, so that all the processors in the first column will store p_1, those in the second column p_2 etc. We first describe the computation of type 3 RR. In the ith row, we find the RR with p_i as the left support. The algorithm is similar to partial maxima computation. The leaves containing the points $p_1, p_2, \ldots, p_i$ are masked and they do not participate in the computation. The point p_i is broadcast to all the active leaves. The leaf containing point p_j such that $p_j.x > p_i.x$, checks whether the point p_j is in Above(p_i) or Below(p_i). In other words, it checks whether $p_j.y > p_i.y$ or $p_j.y < p_i.y$. For any point p_k such that $p_k.x > p_i.x$, we have to find out $MA(p_i, p_k)$ and $MB(p_i, p_k)$. Notice that, the computation of $MA(p_i, p_k)$ is a partial minima computation for $k = i+1, i+2, \ldots, n$, R.right. Similarly, the computation of $MB(p_i, p_k)$ is a partial maxima computation. We describe the computation of $MB(p_i, p_k)$ first. The leaves containing points in the set Above(p_i) does not participate in this computation. These leaves send a value R.bottom and the leaves containing points in the set Below(p_i) send their points up the tree. From then onwards the computation is similar to the partial maxima computation described before. The only difference is that the root sends a value R.bottom to its left son in the second phase because this value does not influence the partial maxima computation. Similarly, in the computation of $MA(p_i, p_k)$, the leaves containing points in Below(p_i) send R.top and other leaves send their points up the tree. The root sends R.top to its left son in the second phase. When the leaf containing the point p_k gets both the values $MA(p_i, p_k)$ and $MB(p_i, p_k)$, it checks whether property 2 is satisfied and if so, computes the area of the type 3 RR with p_i as the left and p_k as the right support.

This computation is done simultaneously in all the rows and the time requirement is $O(\log n)$. Once the computation in each row is over, we can find the overall maximum in another $O(\log n)$ time. For type 2 RR, we perform another maxima and minima computation. In the ith row, the value p_i is broadcast to all processors $p_1, p_2, \ldots, p_{i-1}$. These processors decide whether they participate in MA(R.left, p_i) or MB(R.left, p_i) computation. The leaves send their points to their fathers and after a simple maximum and minimum computation, the root of the active subtree gets the values MA(R.left, p_i) and MB(R.left, p_i) and broadcast this to the leaf containing p_i which in turn computes the area of the type 2 RR with R.left, MA(R.left, p_i), p_i and MB(R.left, p_i) as the left, top, right and bottom supports. The overall time is $O(\log n)$. The type 1 RR, can be easily found out by sorting the points according to their y coordinates and the details are omitted. Since the lower bound for solving any problem on a mesh-of-trees is $\Omega(\log n)$ (Ullman 1984; Lodi & Pagli 1985), we state the following.

Theorem 5.1. *The MER problem can be solved on a mesh of trees in optimal $O(\log n)$ time.*

6. Conclusion

In this paper, we first presented efficient parallel algorithms for the area and perimeter cases of the maximum empty rectangle problem on a CREW PRAM. These algorithms have improved time complexities compared to the existing algorithms, but the processor-time product remains unchanged. After this, we have given a new characterization for the MER problem and solved it on EREW PRAM and mesh-of-trees architectures. No optimal parallel algorithm exists today for these problems. The processor-time products of the best parallel algorithms are a $O(\log n)$ factor worse

than those of the complexities of the best sequential algorithms. Hence, it remains an interesting open problem to find better parallel algorithms for the maximum empty rectangle problem. For the largest area empty rectangle problem, finding a parallel algorithm which runs in $O(\log n)$ time and uses a small number of processors is another open problem.

The authors are grateful to two anonymous referees for their valuable comments and suggestions which improved the presentation of this paper considerably.

References

Aggarwal A, Klawe M, Moran S, Shor P, Wilbur R 1987 Geometric applications of a matrix searching algorithm. *Algorithmica* 2: 195–208

Aggarwal A, Kravets D, Park J, Sen S 1989 Parallel searching in generalized Monge arrays with applications. Manuscript, IBM Research Division, T J Watson Research Center

Aggarwal A, Suri S 1987 Fast algorithms for computing the largest empty rectangle. *Proc. of the Third Annual ACM Symposium on Computational Geometry* (New York: ACM Press) pp. 278–290

Aggarwal A, Suri S 1989 Fast algorithms for computing largest empty rectangles (unpublished manuscript)

Atallah M J, Fredrickson G N 1986 A note on finding the maximum empty rectangle. *Discrete Appl. Math.* 13: 87–91

Atallah M J, Kosaraju S R 1989 An efficient algorithm for maxdominance, with applications. *Algorithmica* 4: 221–236

Berkman O, Schieber B, Vishkin U 1988 Some doubly logarithmic optimal parallel algorithms based on finding nearest smallers, Research report, IBM T J Watson Research Center

Chazelle B M, Drysdale R L, Lee D T 1986 Computing the largest empty rectangle. *SIAM J. Comput.* 15: 300–315

Cole R 1988 Parallel merge sort. *SIAM J. Comput.* 17: 770–785

Datta A 1991 Efficient algorithms for the largest rectangle problem. *Inf. Sci.* (to appear)

Datta A, Krithivasan K 1990 Efficient parallel algorithms for the maximum empty rectangle problem in shared memory and other architectures. *Proc. 1990 Int. Conf. Parallel Processing* vol. 3, pp. 344–345

Datta A, Krithivasan K 1991 An efficient algorithm for the maximum empty rectangle problem in three dimensions. *Proc. Third Canadian Conference on Computational Geometry*, Vancouver, Canada

Gibbons A, Rytter W 1988 *Efficient parallel algorithms* (Cambridge, NY: Cambridge University Press)

Kruskal C P, Rudolph L, Snir M 1985 The power of parallel prefix. *IEEE Trans. Comput.* C-34: 965–968

Lodi E, Pagli L 1985 A VLSI solution to the vertical segment visibility problem. *IEEE Trans. Comput.* C-35: 923–928

Mckenna M, O'Rourke J, Suri S 1985 Finding the largest rectangle in an orthogonal polygon. *Proc. of 23rd Annual Allerton Conference on Communication, Control and Computing*, Urbana-Champaign, Illinois

Namaad A, Hsu W L, Lee D T 1984 On maximum empty rectangle problem. *Discrete Appl. Math.* 8: 267–277

Orlowski M 1990 A new algorithm for the largest empty rectangle problem. *Algorithmica* 5: 65–73

Ullman J D 1984 *Computational aspects of VLSI* (Computer Science Press)