Computing the depth of an arrangement of axis-aligned rectangles in parallel*

Helmut Alt† Ludmila Scharf†

Abstract

We consider the problem of computing the depth of the arrangement of n axis-aligned rectangles in the plane, which is the maximum number of rectangles containing a common point. For this problem we give a sequential $O(n \log n)$ time algorithm, and a parallel algorithm with running time $O(\log^2 n)$ in the classical PRAM model. We also describe how to adopt the parallelization to a shared memory machine model with a fixed number of processing units.

1 Introduction

In this paper we consider a basic geometric problem: how to compute the *depth* of the arrangement of n axis-aligned rectangles in $\mathbb{R}^2$. The depth of a point p is defined as the number of rectangles containing p, and the depth of the arrangement is the maximum depth over all points in $\mathbb{R}^2$, or, equivalently, it is the maximum number of rectangles that contain a common point, see Figure 1 for an example.

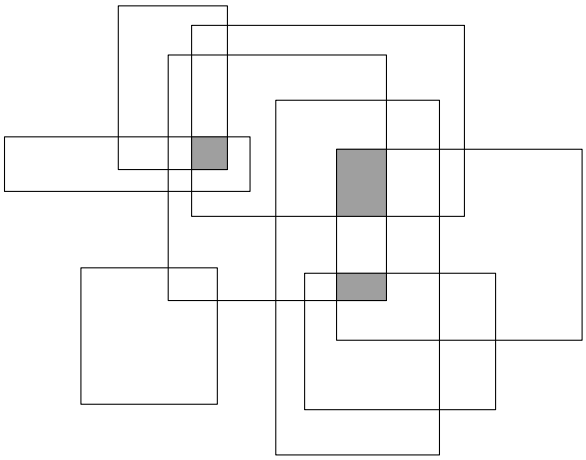

Figure 1: Arrangement of axis-parallel rectangles with cells of maximal depth 4 (shaded).

*This research was partially supported by the German Research Foundation (DFG) in the project "Parallel algorithms in computational geometry with focus on shape matching", contract number AL 253/7-1.

†Institute of Computer Science, Freie Universität Berlin, {alt,scharf}@mi.fu-berlin.de

We describe a parallel algorithm for this problem for a shared memory parallel machine model. The algorithm has a running time $O(\log^2 n)$ and total work $O(n \log^2 n)$ in the classical EREW-PRAM model. We also describe how to adopt the parallelization to a more realistic assumption of having a fixed number k of processing units in a shared memory machine, which fits better the modern multi-core processors.

The current trend in the microprocessor industry is to increase the performance in computing not by increasing the CPU clock rates but by multiple CPU cores working on shared memory and common cache. This trend in the hardware development makes the design of parallel algorithms once again an active topic in the algorithmic community.

In this paper we first describe a sequential $O(n \log n)$ time algorithm for computing the depth of the arrangement of axis-aligned rectangles in the plane. Namely, we construct a balanced search tree on the x-coordinates of the corners of rectangles and then perform a plane sweep along the y-axis, while updating the box coverage information in the tree.

To our knowledge no $O(n \log n)$ algorithm for the depth problem has been given before explicitly, although the result is somehow "folklore" knowledge in the computational geometry community. In fact, the scheme of algorithms given for solving Klee's measure problem (KMP), i.e., computing the volume of the union of n axis-parallel boxes, can be used for computing the depth of arrangements. For the two-dimensional KMP Bentley described but did not publish such an algorithm [2] the idea of which is given in [6], however. Our algorithm is similar to some extent. We develop and describe it in detail, mostly in order to derive from it in Section 3 the efficient parallel algorithm for the problem.

For the applications of the depth computation problem we mention two examples: One is a geometric pattern matching problem. For two m-point sets A and B in $\mathbb{R}^2$ finding a transformation minimizing the directed L_∞-Hausdorff distance from A to B can be reduced to finding the depth in an arrangement of $O(m^2)$ boxes. Another example is a clustering problem: For a given set of n points in $\mathbb{R}^2$ and a given radius r find a L_∞-disk of radius r containing the largest number of points, that is, the densest cluster of radius r. This clustering problem is dual to determining the deepest point in the arrangement of n boxes with side length $2r$.

For general shape (algebraic) regions, not just axis-aligned rectangles, there is no better algorithm known for computing the depth of their arrangement than to construct the complete arrangement and then to traverse it. For arrangements of disks the problem is known to be 3-SUM hard [1], so sub-quadratic algorithms are not likely to exist. For an arrangement of axis-aligned boxes in higher dimensions Chan [3] describes a sequential algorithm with running time $O((n^{d/2}/\log^{d/2-1} n)\ \log^{d/2} \log n)$ for $d \geq 3$.

2 Sequential Algorithm

In this section we describe the sequential algorithm for computing the depth of the arrangement of axis-aligned rectangles.

The general idea is the following: For a given set of n axis-aligned rectangles we build a balanced binary search tree T on the x-coordinates of the vertical sides of the rectangles, so that all x-coordinates are in the leaves of the tree. Let $x_1, x_2, \ldots, x_{2n}$ be the x-coordinates of the vertical sides of the rectangles in sorted order. With the leaf labeled with x_i, $i = 1, \ldots, 2n-1$, we associate the interval $[x_i, x_{i+1})$. The last leaf, labeled with x_{2n}, is associated with the interval $[x_{2n}, x_{2n}]$. With an internal node v we associate the union of the intervals

of its children. Space requirement for the tree is linear in the number of rectangles.

Next we perform a top-down sweep along the y-axis. Each sweepline event, i.e., a y-coordinate of the top or bottom of a rectangle, has two x-coordinates a and b, with $a, b \in \{x_1, \ldots, x_{2n}\}$ and $a < b$, of the vertical sides of the corresponding rectangle and an event value d associated with it. The event value is $d = 1$ if it is the top of the rectangle (the rectangle is "opened") and $d = -1$ if it is the bottom (the rectangle is "closed").

To process a sweepline event we traverse the tree T from the root node to the leaves labeled a and b. In the nodes of the tree we want to count the number of rectangles covering the associated x-interval, and to update this information with each y-event. Of course, we cannot store this information directly and update it for all covered nodes for each rectangle, since that could make up to linear time per update.

Instead, we maintain in every internal node v for the current state of the sweepline in counters l and r the number of rectangles covering the interval of the left and right child of v that were opened minus the number of ones closed since the last traversal of that child. In other words, the counters l and r store the additive update of the information about how many open rectangles cover the interval of the left and right child, respectively, at the current position of the sweepline. Additionally, counters l_m and r_m store the maximum of this additive update for the left and right child, respectively, since the last traversal of the corresponding child node. Every leaf node contains counters c and c_m, which keep track of the current and maximum coverage of the associated interval during the sweep.

The information in counters l, r, l_m and r_m is exactly as described above if the node v is traversed, and thus updated, by the current sweepline event. For subsequent events that do not traverse v the information may get outdated. Thus, the counters of the internal node v store the updates that happened between the last traversal of the corresponding child node and the last traversal of v. The counters c and c_m in the leaf nodes are only updated for the open- and close-events of the corresponding rectangle.

During each traversal of v by one of the searches the counter values are propagated from v to its child on the search path in temporary counters t and t_m, which are initially set to 0. I.e., once we updated v as described below and move to its left (right) child, t and t_m are set to the values of $v.l$ and $v.l_m$ ($v.r$ and $v.r_m$), and then $v.l$, $v.l_m$ ($v.r$, $v.r_m$) are reset to 0. Thus, when we enter the child node w the counter t is the additive change since the last update of w of the number of open rectangles that completely cover the interval of w; t_m is either the maximum value of that change between the last update of w and the current event, or 0 if the additive updates were all negative.

An update of an internal node v is performed slightly differently depending on whether both x-coordinates a and b associated with the event are contained in the subtree rooted at v (see Procedure 1: SearchBoth), or the search paths for a and b split earlier in the tree and the subtree of v contains only a (see Procedure 2: SearchLeft) or only b (procedure SearchRight).

If both a and b are contained in the subtree rooted at v we need to update the counters of v only with the values propagated from the parent node, without considering the event value. For l and r we simply add the value of t (lines 3 and 4 of procedure SearchBoth). The max-counters (l_m and r_m) are set to the maximum of their old value, and the sum of the old counter (l or r, resp.) and t_m (lines 1 and 2 of procedure SearchBoth).

If both a and b are contained in the same, say left, subtree of v then t and t_m are set to l and l_m, the search for both x-coordinates is continued in the left subtree, and the counters l and l_m are reset to 0. If the paths to a and b split in the node v, we perform two separate

searches in the left and right subtrees (lines 12 and 13).

```
 1 v.l_m = max(v.l_m, v.l + t_m)
 2 v.r_m = max(v.r_m, v.r + t_m)
 3 v.l = v.l + t
 4 v.r = v.r + t
 5 if a < b ≤ v.x then
 6     SearchBoth (v.left, a, b, v.l, v.l_m)
 7     v.l = v.l_m = 0
 8 if v.x < a < b then
 9     SearchBoth (v.right, a, b, v.r, v.r_m)
10     v.r = v.r_m = 0
11 if a ≤ v.x < b then
12     SearchLeft(v.left, a, v.l, v.l_m)
13     SearchRight(v.right, b, v.r, v.r_m)
14     v.l = v.l_m = v.r = v.r_m = 0
```

Procedure 1 : SearchBoth(v, a, b, t, t_m)

Notice that the update instructions for the counters make sure that the stored values are not the absolute numbers of simultaneously opened rectangles but an additive update to the information stored in the corresponding subtree. To illustrate this remark we consider an example of a (sub-)sequence of update events for a node v starting from some event, that resets the counters l and l_m to 0, that is, the event updates the left child of v with the information gathered so far, see Figure 2. Assume that the node v now gets the following events: two closing, three opening, and one closing, where the left child is not traversed by the events, but the corresponding rectangles cover the interval of the left child. Then the values of l and l_m counters are updated as shown in Figure 2. At the end of this sub-sequence the interval of the left child is covered by exactly as many rectangles as at the beginning: two old rectangles were closed, three new opened and one new closed, thus $l = 0$. However, during this sub-sequence, between the last two events, the interval of the left child was covered by one more rectangle than in the beginning. This additional coverage is maximal over the whole sub-sequence, therefore, $l_m = 1$.

Once the search path for a and b splits in some vertex, we know that the current rectangle spans all intervals of the right subtrees of the left search path, i.e., path to a, and all intervals of the left subtrees of the right search path, i.e., path to b. Therefore, for a node v on the left (right) search path we also add the event value d to the counter r (l). When we reach the leaves containing a and b we can update the current and maximum depth of the associated intervals. The updates are described in the procedure SearchLeft for the search path of a. The search for b is performed in a procedure called SearchRight, which is analogous to procedure SearchLeft and, therefore, is not explicitly given here.

After all sweepline events have been processed, the depth of the arrangement is determined as the maximum of the c_m counters of the leaf nodes.

The correctness of the algorithm is based on the following observation: Once we reach the bottom of a rectangle, the counter c_m in the leaf node labeled with the x-coordinate of its left vertical boundary contains the maximum coverage of the associated x-interval between the highest y-coordinate and the y-coordinate of the bottom of the rectangle. Since, clearly,

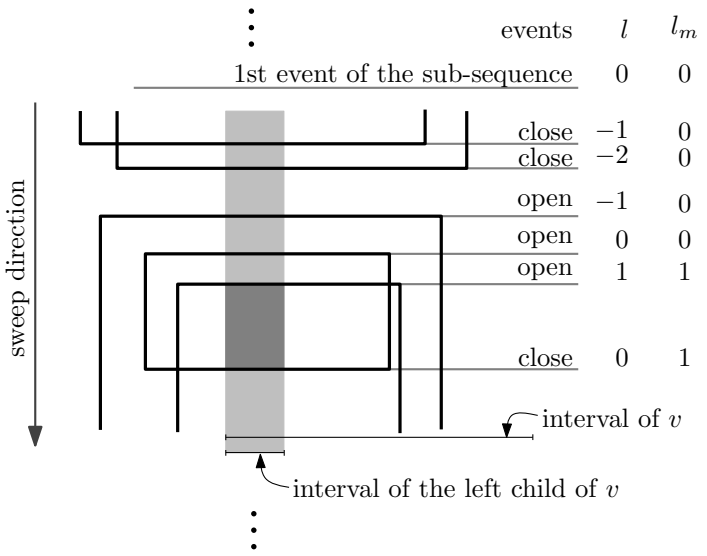

Figure 2: An example of a sub-sequence of sweepline events and the corresponding updates of the counters l and l_m of an internal node v.

```
1  if a ≤ v.x and v is an internal node then
2      v.r_m = max(v.r_m, v.r + t_m, v.r + t + d)
3      v.r = v.r + t + d
4      SearchLeft(v.left, a, v.l + t, max(v.l_m, v.l + t_m))
5      v.l = v.l_m = 0
6  if a > v.x then
7      v.l_m = max(v.l_m, v.l + t_m);        v.l = v.l + t
8      SearchLeft(v.right, a, v.r + t, max(v.r_m, v.r + t_m))
9      v.r = v.r_m = 0
10 if a = v.x and v is a leaf then
11     c_m = max(c_m, c + t_m, c + t + d)
12     c = c + t + d
```

Procedure 2 : SearchLeft(v, a, t, t_m)

every maximally covered cell has a left vertical boundary that is a part of a left boundary of a rectangle, and thus covers at least one leaf interval, we capture at least one cell with maximum depth this way, i.e., store its depth in a counter c_m of one of the leaves.

If we want not only to compute the depth, but also get a point with maximum depth, we can additionally store a y-coordinate for each max-counter. This y-coordinate has to be updated with the y-value of that event which results in the counter update.

The time needed to construct the tree and to sort the y-events is $O(n \log n)$. Each of the $2n$ events is processed in $O(\log n)$ time.

Theorem 1 summarizes the result of this section:

Theorem 1. *The depth of an arrangement of n axis-aligned rectangles in $\mathbb{R}^2$ can be computed in time $O(n \log n)$ with $O(n)$ additional memory.*

3 Parallel Algorithm

To enable a parallel execution of the algorithm we maintain so-called "history lists" in the nodes of the tree T. A history list of a node v contains an entry for each event of the sweepline that traverses the node v, i.e., for each y-coordinate of a top or bottom side of a rectangle spanning the interval $[a, b]$ in x-direction, such that a or b is contained in the subtree rooted at v.

A history entry α of an internal node contains its "timestamp" – the y-coordinate (or the rank of the y-coordinate) of the sweepline event, the corresponding x-values, the event value d and the counters l, r, l_m, r_m, as described in Section 2, and reset-flags ρ_l, ρ_r. The reset flags indicate whether the values of the left or right counters, respectively, "survive" until the next event: The value of ρ_l/ρ_r in the entry α is 0 if the event of α causes the traversal of the left/right subtree, since in this case the counter values are propagated to the subtree and will be reset in the node v. Otherwise, the value is 1. Additionally, every history event has a pointer to the corresponding event, i.e., the event with the same y-coordinate, in the parent node. A history entry of a leaf node contains only its y-coordinate, the event value d, and two counters c and c_m.

Every y-event appears in at most two nodes of each level of the tree and requires constant space. Thus, the space for the tree is $O(n \log n)$.

All information of a history event, except for the counter values l, l_m, r, r_m, is known at the construction time of the tree and can be set during the tree construction. Now we can "fill out" the counter values in the history lists starting with the root node down to the leaves. The left/right counters in all root node events are set to 0. For an internal node v let $\alpha^{(i)}$ be the i-th history event of v and let $\alpha^{(j)}$ be the corresponding history event in the parent node of v, i.e., $y^{(i)} = y^{(j)}$. Then the counters of $\alpha^{(i)}$ are computed according to the following rules: Let $t^{(i)}$ and $t_m^{(i)}$ denote the values of $r^{(j)}$ and $r_m^{(j)}$ of the event $\alpha^{(j)}$ if v is a right child of its parent node, and values of $l^{(j)}$ and $l_m^{(j)}$ otherwise. If $\alpha^{(i)}$ contains both x-coordinates a and b

associated with $y^{(i)}$ then set

$$l^{(i)} = l^{(i-1)} \cdot \rho_l^{(i-1)} + t^{(i)} \tag{1}$$

$$l_m^{(i)} = \max\left\{ l_m^{(i-1)} \cdot \rho_l^{(i-1)},\ l^{(i-1)} \cdot \rho_l^{(i-1)} + t_m^{(i)} \right\} \tag{2}$$

$$r^{(i)} = r^{(i-1)} \cdot \rho_r^{(i-1)} + t^{(i)} \tag{3}$$

$$r_m^{(i)} = \max\left\{ r_m^{(i-1)} \cdot \rho_r^{(i-1)},\ r^{(i-1)} \cdot \rho_r^{(i-1)} + t_m^{(i)} \right\}, \tag{4}$$

where the values with the high-index $(i-1)$ denote the values of the history event preceding $\alpha^{(i)}$ in the node v.

Also if a history event $\alpha^{(i)}$ of a node v contains only a, and a is in the right subtree of v, or if $\alpha^{(i)}$ contains only b, and b is in the left subtree of v, the counters are set as above. That is, the values from the parent node v are propagated to the corresponding child node but the interval associated with the child is not completely covered by the rectangle causing the event, and thus, we do not need to consider the event value d.

In case $\alpha^{(i)}$ contains only a, and a is in the left subtree of v, then the complete right subtree is covered by the current rectangle. Therefore, the right counters are incremented by the event value $d^{(i)}$:

$$r^{(i)} = r^{(i-1)} \cdot \rho_r^{(i-1)} + t^{(i)} + d^{(i)} \tag{5}$$

$$r_m^{(i)} = \max\left\{ r_m^{(i-1)} \cdot \rho_r^{(i-1)},\ r^{(i-1)} \cdot \rho_r^{(i-1)} + t_m^{(i)},\ r^{(i)} \right\}. \tag{6}$$

The left counters are updated as in equations (1), (2). In case $\alpha^{(i)}$ contains only b, and b is in the right subtree of v the left counters must be adjusted analogously:

$$l^{(i)} = l^{(i-1)} \cdot \rho_l^{(i-1)} + t^{(i)} + d^{(i)} \tag{7}$$

$$l_m^{(i)} = \max\left\{ l_m^{(i-1)} \cdot \rho_l^{(i-1)},\ l^{(i-1)} \cdot \rho_l^{(i-1)} + t_m^{(i)},\ l^{(i)} \right\}. \tag{8}$$

and the right counters are updated as in equations (3), (4).

The counters of the i-th entry in a leaf node are updated as follows:

$$c^{(i)} = c^{(i-1)} + t^{(i)} + d^{(i)} \tag{9}$$

$$c_m^{(i)} = \max\left\{ c_m^{(i-1)}, c^{(i-1)} + t_m^{(i)}, c^{(i-1)} + t^{(i)} + d^{(i)} \right\} \quad . \tag{10}$$

Then, after all events have been processed, each leaf node stores the maximal coverage of its associated interval up to the position where the (last) rectangle with the corresponding vertical side was closed.

The depth of the arrangement is then the maximum over the c_m counters of the leaves.

So we could build the tree T and then traverse it level-by-level starting from the root node to the leaves, and node-by-node within one level, setting the counters in all history events. Thus, we would have a sequential algorithm with running time in $O(n \log n)$ as before but with $O(n \log n)$ memory usage.

Parallel implementation on a PRAM. For the parallel algorithm we assume that there are $O(n)$ processors on a EREW-PRAM machine available. Then sorting of the corner points

of the rectangles once by y-coordinates and once by x-coordinates can be performed in $O(\log n)$ time, i.e., $O(n \log n)$ total work, using, for example, the sorting algorithm by Cole [4].

In the following we assume that all x-coordinates of the vertical sides of the rectangles are distinct, which simplifies the analysis and the description. With careful consideration of technical details the analysis can be extended to the general setting within the same time and total work bounds as below.

The tree T without the history lists can be build straightforwardly in time $O(\log n)$ on pre-sorted x-coordinates of the vertical sides of the rectangles.

The unsorted history lists for each level of the tree can be constructed in constant time per level with $O(n)$ processing units: We assign one processor to every history event. Every processor writes an entry for its event to the history lists of the corresponding two leaf nodes. Then, each processor creates level-by-level parent entries for its event in the history lists of the nodes on the path from the two leaves to the root node. At the end of this process the history lists contain entries with correctly set timestamps (the y-coordinates), pointers to the parent events, the event value d, and, for the internal nodes, the reset switches ρ_l, ρ_r. The counter values remain open.

Several processors can write their entries to the history list of the same node in parallel since, if we organize the history lists as arrays, every processor independently can easily compute the index of its entry in the list. The total time for the construction of the unsorted history lists is $O(\log n)$.

Now we can sort all history lists by timestamps in parallel. The total size of the history lists in one level is at this stage $2n$ and the total size of all history lists in the tree is $O(n \log n)$. We can sort the history lists level-by-level, processing all lists of one level in parallel in time $O(\log n)$ per level with $O(n)$ processors. Then, for the complete tree, the construction time of the sorted history lists is $O(\log^2 n)$.

The computation of the left/right counters in the event entries is performed level-by-level starting with the root node down to the leaves. We first observe that the computation of the counters $l^{(i)}$ and $r^{(i)}$ according to equations (1), (3), (5), (7) corresponds to a prefix sum computation: Consider an internal node v and its left counter of the i-th history entry $l^{(i)}$. Let j be the highest index $\leq i$ with $\rho_l^{(j)} = 0$. Then the value of $l^{(i)}$ is the sum $\sum_{k=j+1}^{i}(t^{(k)} + d^{(k)})$, where $d^{(k)}$ is set to 0 if the computation of $l^{(k+1)}$ follows equation (1). Thus, if we can subdivide the l-counters of a history list into subsequences corresponding to blocks of ones terminated by a zero of the ρ_l-switches, then we can in a first step set each $l^{(i)}$ to $t^{(i)}$ or $t^{(i)}+d^{(i)}$, respectively, and then perform parallel prefix sum computations on the subsequences.

For the subdivision into subsequences we can again use the parallel prefix sum computation. First, we invert the values in the ρ_l-sequence, shift it by one to the right, and add a preceding 0, i.e., $\hat{\rho}_l^{(i)} = 1 - \rho_l^{(i-1)}$ for $i > 1$ and $\hat{\rho}_l^{(1)} = 0$. Now we can compute the prefix sum of the $\hat{\rho}_l$-values. The blocks of equal values in this new sequence correspond exactly to the subsequences in the l-sequence. Then in the prefix sum computation of the l-values we only need to consider those entries that have the same $\hat{\rho}_l$-value.

The r-counter values are computed analogously. Prefix sum computation can be performed in $O(\log n)$ time [5].The total size of all prefix sum lists of one level is $O(n)$, and there are $O(\log n)$ levels. So we need $O(\log^2 n)$ time in total.

For the computation of the max-counters according to equations (2), (4) or (6), (8), e.g., for $l_m^{(i)}$, we need the values $l_m^{(i-1)}$, $l^{(i-1)}$, t_m, and possibly $l^{(i)}$. All of these values, except for $l_m^{(i-1)}$, are computed by now. Observe, that these equations are of the form $u^{(i)} =$

$\max\left(u^{(i-1)} \cdot \rho^{(i-1)},\ v^{(i)}\right)$, where, e.g., in (6) $u^{(i)} = r_m^{(i)}$ and $v^{(i)} = \max\left(r^{i-1} \cdot \rho^{(i-1)} + t_m^{(i)}, r^{(i)}\right)$. So the $u^{(i)}$ are prefix maxima of the previously computed $v^{(i)}$ and can be computed in $O(\log n)$ time analogously to the prefix sums.

We summarize the preceding sketch of the parallel algorithm and its analysis:

Theorem 2. *The depth of an arrangement of n axis-aligned rectangles in $\mathbb{R}^2$ can be computed on a EREW-PRAM with $O(n)$ processing units in time $O(\log^2 n)$.*

Parallel implementation for a fixed number k of processors with shared memory: Sorting of the x- and y-coordinates of the vertical and horizontal sides of the rectangles can obviously be performed on a k-processor machine in time $O(\frac{n}{k} \log n)$.

Then we have to split the work performed by the algorithm between k processors. For this purpose we split the tree construction into k subtrees, each containing at most $\lceil 2n/k \rceil$ x-coordinates. Each of the subtrees is constructed by one processor sequentially. Afterwards, the k subtrees are combined into a single tree by adding a tree of height $\lceil \log k \rceil$ on top of the subtrees. The tree construction includes the history lists except for the values of the counters r, l, r_m, l_m in the internal nodes, and the counters c, c_m in the leaves.

For the computation of the counters in the history lists we apply the same idea: for the top tree we apply parallel prefix sum computation by processing the history lists in blocks of at most k elements. There are $O(n/k)$ such blocks in each level, and each block is processed in $O(\log k)$ time. Thus, the $\lceil \log k \rceil$ levels of the top tree can be processed in $O(\frac{n}{k} \log^2 k)$ time. For the k subtrees we apply the sequential algorithm to find the maximum depth in each subtree. The size of the subtrees is $O(n/k)$, thus, the time for the remaining levels is $O(\frac{n}{k} \log n)$. The total time is then $O(\frac{n}{k}(\log^2 k + \log n))$.

Summarizing, we have:

Corollary 1. *The depth of an arrangement of n axis-parallel rectangles can be computed in parallel by k processors with shared memory in time $O(n/k(\log^2 k + \log n))$.*

4 Future Work

Although the depth computation of a set of rectangles in $\mathbb{R}^2$ is an interesting problem on its own, we plan to develop parallel algorithms for higher dimensional depth computation. Further, we are interested in an implementation of the algorithm presented here, and possibly algorithms for higher dimensional problems, and in their experimental evaluation. The implementations should be performed for currently available parallel hardware platforms, such as multicore CPUs and general purpose GPUs.

References

[1] B. Aronov and S. Har-Peled. On approximating the depth and related problems. *SIAM J. Comput.*, 38(3):899–921, 2008.

[2] J. L. Bentley. Algorithms for Klee's rectangle problems. Unpublished notes, 1977.

[3] T. M. Chan. A (slightly) faster algorithm for Klee's measure problem. In *SCG '08: Proceedings of the twenty-fourth annual symposium on Computational geometry*, pages 94–100, New York, NY, USA, 2008. ACM.

[4] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[5] D. W. Hillis and G. L. Steele. Data parallel algorithms. *Communications of the ACM*, December 1986.

[6] J. van Leeuwen and D. Wood. The measure problem for rectangular ranges in d-space. *J. Algorithms*, 2(3):282–300, 1981.