Optimal Cooperative Search in Fractional Cascaded Data Structures[1]

R. Tamassia[2] and J. S. Vitter[3]

Abstract. Fractional cascading is a technique designed to allow efficient sequential search in a graph with catalogs of total size n. The search consists of locating a key in the catalogs along a path. In this paper we show how to preprocess a variety of fractional cascaded data structures whose underlying graph is a tree so that searching can be done efficiently in parallel. The preprocessing takes $O(\log n)$ time with $n/\log n$ processors on an EREW PRAM. For a balanced binary tree, cooperative search along root-to-leaf paths can be done in $O((\log n)/\log p)$ time using p processors on a CREW PRAM. Both of these time/processor constraints are optimal. The searching in the fractional cascaded data structure can be either explicit, in which the search path is specified before the search starts, or implicit, in which the branching is determined at each node. We apply this technique to a variety of geometric problems, including point location, range search, and segment intersection search.

Key Words. Parallel computing, Fractional cascading, PRAM, Search, Cooperative search, Point location, Computational geometry.

1. Introduction. Fractional cascading is a preprocessing technique that allows efficient searching of the same key in a collection of catalogs (sorted lists) [3], [4]. More formally, there is a catalog associated with each node of a graph G. Given a search argument y and a search path in G, the goal is to find the smallest entry $\geq y$ in each of the catalogs of the nodes on the search path. The search path can be either explicit, in which it is specified before the search starts, or implicit, in which the branching is determined at each node. In many important applications of fractional cascading the graph G is a balanced binary tree and the search path is from the root to a leaf.

Assume for simplicity that the graph G has bounded degree. Each entry in each node's catalog has a "bridge" pointer to an entry in the catalog of each adjacent node. Searching proceeds as follows: Starting at the first node on the path, the desired entry is found via a binary search in the node's catalog. The bridge from this entry to the next catalog on the search path points to an entry within a constant-time walk of the desired entry in that catalog. This last step is iterated for the remaining nodes on the search path. The search

time is thus $O(\log n + m)$, where n is the total size of the catalogs and m is the length of the search path.

The search described above in the fractional cascaded data structure is sequential in nature. In this paper we consider a rooted tree T with $O(n)$ nodes storing catalogs of total size n, and explore optimum ways to preprocess T so that searches along root-to-leaf paths can be performed optimally in parallel. Our data structures support efficient *cooperative search* with p processors for any value of p in the range $1 \le p \le n$. In a cooperative search, all p processors work together in parallel on the same search. The preprocessing takes $O(\log n)$ time with $n/\log n$ processors on an EREW PRAM. For a balanced binary tree, cooperative search along explicit or implicit root-to-leaf paths can be done in $O((\log n)/\log p)$ time using p processors on a CREW PRAM. Both of these time/processor tradeoffs are optimal.

One motivation for this work is the problem of cooperative point-location search in an n-vertex planar subdivision. Dadoun and Kirkpatrick [7] show how to perform cooperative point location on their hierarchical data structure [12] in $O((\log n)/\log p)$ time, which is optimal. Unfortunately, the preprocessing requirements are high: they use $O(\log^2 n)$ time with $O(n^3)$ processors on a CREW PRAM. In [17] we show how to construct the bridged separator tree data structure [13], [9] for a monotone subdivision in $O(\log n)$ time using $n/\log n$ processors on an EREW PRAM. The bridged separator tree is very efficient in practice [8]. The search path used to process a point-location query is "highly" implicit, due to the space-saving nature of the bridged separator tree, which makes cooperative search seem especially difficult. In this paper we show how to modify the bridged separator tree data structure of [17] within the same optimal time-processor bound so that it supports optimal cooperative point location.

In the next section we sketch our cooperative search algorithm for trees. We focus on balanced binary trees since the majority of applications of fractional cascading fall into this category. In Section 3 we give optimal preprocessing and cooperative search algorithms for planar point location, and we extend them to spatial point location. In Section 4 we sketch algorithms for orthogonal range search, orthogonal segment intersection, and point enclosure. Finally, open problems are discussed in Section 5.

2. Cooperative Search in Trees.

In this section we show how to preprocess a rooted, ordered, balanced binary tree T with $O(n)$ nodes storing catalogs with a total of n items into a fractional cascaded data structure, so that cooperative searches with p processors can be done in optimal $O((\log n)/\log p)$ time.

For ease of exposition, we distinguish two types of search paths in the tree T: *explicit* and *implicit*. An explicit search path is a path in tree T that is determined before the search begins. In an implicit search the search path is not specified beforehand, but rather each branch taken is determined by the result of a secondary comparison.

Each node in the tree T contains an ordered sequence of distinct catalog entries; for convenience, we let each catalog contain the terminal entry $+\infty$. The total number of catalog entries is n, although individual catalogs may contain as many as $\Theta(n)$ entries. Given a search argument y and a node v in T, we denote by *find*(y, v) the smallest entry in v's catalog that is not smaller than y. The output of the search is the list of catalog entries *find*(y, v), for each node v on the search path.

In the basic form of implicit search, each catalog entry is associated with a primary value, as before, but it also contains auxiliary secondary information. The search argument is a pair $q = (x, y)$. The search starts at the root. The branch taken at each node v on the search path is determined by the function call $branch(q, find(y, v))$, which returns either *left* or *right*, based on the secondary information of the catalog entry $find(y, v)$. If *branch* returns *left*, the left branch is taken; otherwise *branch* returns *right*, and the right branch is taken. We make the following *consistency assumption* for basic implicit search: for each node w not on the search path, $branch(q, find(y, w))$ returns *right* if the search path is to the right of w and *left* if the search path is to the left of w, and at the leaf node v on the search path, *branch* returns *left*.

More intricate forms of implicit search can be used, in which the branching at the current node also depends on the previous branches taken and the consistency assumption does not hold. This is the case for point location, which we discuss in Section 3.

The following theorem states the main result of this section.

THEOREM 1.   *A balanced binary tree with catalogs of total size n and $O(n)$ nodes can be preprocessed in $O(\log n)$ time using $n/\log n$ processors on an EREW PRAM so that subsequent cooperative searches (either explicit or implicit) along root-to-leaf paths can be done in $O((\log n)/\log p)$ time using a CREW PRAM with p processors, for any $1 \le p \le n$, on an $O(n)$-space data structure.*

The time $O\big((\log n)/\log p\big)$ for the cooperative search is optimal, since we can reduce the problem of dictionary searching to cooperative search in a binary tree with catalogs, and thus the lower bound from [16] applies. If cooperative search is done within the EREW PRAM model, the lower bound increases to $\Omega\big(\log(n/p)\big)$.

Intuitively we form a preprocessed version T' of T by starting with the fractional cascaded data structure constructed by [1] and introducing certain substructures. For cooperative searches, we use p processors to traverse $\Theta(\log p)$ levels of the appropriate substructure in constant time, by simulating p-way branching. The hard part is applying this idea to a tree with catalogs, in which the key value in each tree node is not a single value, but rather a variable number of catalog entries. The fact that the catalogs can have a variable number of entries is the main downfall of our first approach at a solution in Section 2.1.

2.1. *Preprocessing.*   In this section we describe the preprocessing needed for converting the balanced binary tree T into the cooperative search data structure T'. The preprocessing can be done in $O(\log n)$ time with $n/\log n$ processors on an EREW PRAM. It consists of two main steps:

Step 1.   Form the fractional cascading data structure of [1].

Step 2.   Form $\lceil \log \log n \rceil - 1$ search substructures T_i, for $0 \le i \le \lceil \log \log n \rceil - 1$. Each data structure T_i is designed to handle cooperative searches for values of p in the range $2^{2^i} < p \le 2^{2^{i+1}}$, so that the log p speedup factor does not vary by more than a multiplicative factor of 2 in each range.

Step 1 can be done within the desired time bounds [1]. The rest of this subsection is devoted to the design and analysis of Step 2.

The difficulty in implementing Step 2 is twofold: The first problem is how to modify the fractional cascading data structure of [1] so that $\Theta(\log p)$ levels can be traversed in one "hop" in constant time, even when the search is implicit and thus the path is not known *a priori*. The second problem is that individual catalogs may be large, even though the total size of all catalogs is $O(n)$, and this impacts the space needed for the data structures.

Let S be the fractional cascaded data structure constructed by the parallel algorithm of [1] when applied to a bidirectional version of tree T, in which each parent-to-child edge is replaced by two directed edges. For convenience, we use S to refer to the portion of S corresponding only to parent-to-child directed edges. The tree S has auxiliary nodes that are not in T, but it can be regarded, with a slight compression of levels, as a directed tree with the same node and edge sets as T, in which each catalog is augmented with "dummy" catalog entries.

From now on, we use the term "catalog" to refer to the augmented catalogs that include the dummy entries. For each node v, edge (v, w), and catalog entry c in v's catalog, there is a "bridge" *bridge*$[v, w, c]$ that points to a catalog entry in w's catalog. The fractional cascaded data structure has the following properties:

1. ("Fan out" property) There is some positive constant b such that, given any two nodes v and w consecutive on a possible search path, *find*(y, w) is within b entries in w's catalog from *bridge*$[v, w, find(y; v)]$.

2. An implication of property 1 is that any two adjacent entries c and c' in v's catalog have bridge pointers to entries at most $2b + 1$ entries apart in w's catalog.

3. Bridges do not "cross over" one another; that is, if $c < c'$, where c and c' are two entries in the catalog for some node v, then *bridge*$[v, w, c] \leq$ *bridge*$[v, w, c']$ for each neighbor w of v.

First Approach. We begin with a naïve attempt to construct T_i, the search structure used when p is in the range $2^{2^i} < p \leq 2^{2^{i+1}}$. We construct T_i by first partitioning S into subtrees of height $h_i = \lfloor \alpha 2^i \rfloor = \Theta(\log p)$, for some sufficiently small constant $0 < \alpha < 1$, rooted at the nodes in S at levels $0, h_i, 2h_i, \ldots$. The idea is to traverse S by traversing $O((\log n)/\log p)$ of these subtrees, each in constant time.

Let U be one of these subtrees of height h_i; we denote U's root by u. We assume that we know the catalog entry $c = find(y, u)$. (If u is the root of S, we can determine c by a parallel binary search in $O(\log_p n) = O((\log n)/\log p)$ time [16]; otherwise, we will be able to determine c as a result of the search of the ancestor subtree of U.) We define the *reach* in U of catalog entry c, denoted by *reach*(c, U), to be the set of all possible catalog entries that can be returned during a search in U; more formally, *reach*(c, U) consists of all (c', u'), where c' is an entry in the catalog of some $u' \in U$, such that there is some search argument y in which *find*$(y, u') = c'$ and *find*$(y, u) = c$. By the "fan out" property of fractional cascading, the reach of c has size $O((2(2b + 1))^{h_i})$, which for any $\beta < 1$ is $O(p^\beta)$, if α is made sufficiently small, by definition of h_i. The reach of c is pictured in Figure 1.

If we store *reach*(c, U) for each catalog entry c in U's root u, then we might be able to traverse U in constant time by exhaustively assigning the p processors to each element
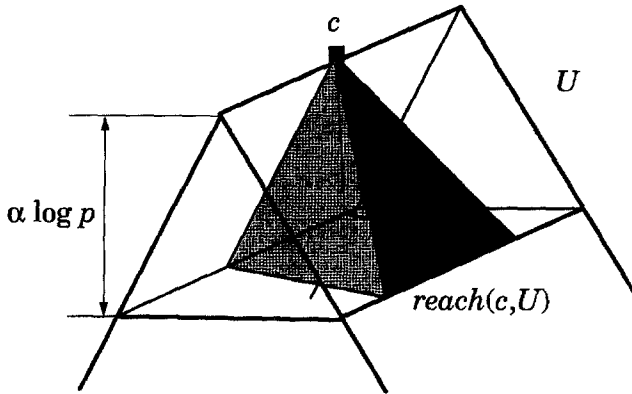
Fig. 1. The reach in U of catalog entry c, denoted *reach*(c, U), consists of $O(p^\beta)$ catalog entries, with $\beta < 1$.

in *reach*(c, U). The problem that arises, however, is that the reaches of adjacent catalog entries c and c' in u's catalog can overlap, and the resulting space usage might be $\Theta(n^2)$, which is nonoptimal by a multiplicative factor of n. The space usage would be $O(n)$ if the number of catalog entries in each node were the same, namely, $O(1)$. In reality, however, the number of catalog entries per node can be quite variable, subject to the constraint that the total number of catalog entries is $O(n)$.

In addition we need $\lceil \log \log n \rceil - 1$ data structures, one for each value of i, which further increases the space usage by a multiplicative factor of $\log \log n$.

Second Approach. A possible remedy to reduce the space of T_i is to prune the reach sets so that each entry (c', u') is stored physically in the pruned reach set of only one catalog entry in u's catalog, as illustrated in Figure 2. Instead of processing only the pruned *reach*(c, U) during a search, we must now also process the pruned *reach*(c', U) of $O(p^\gamma)$ catalog entries c' preceding c, for some $\gamma < 1$, in order to guarantee that each entry in the nonpruned *reach*(c, U) is processed.

However, a new problem now arises: In order to assign processors to a pruned reach set in constant time, the pruned reach set must be stored contiguously in memory, such
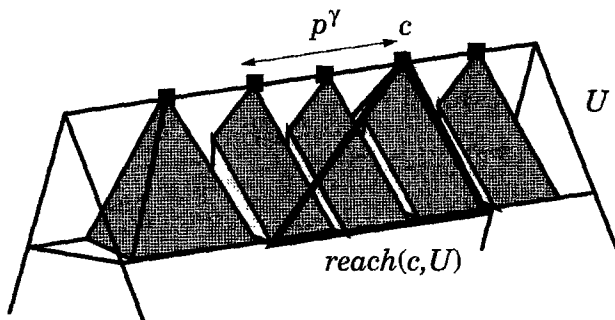

Fig. 2. The reaches of the catalog entries in u's catalog are pruned to avoid overlap, so that each entry is stored in only one pruned reach set.

as in depth-first or breadth-first order, for example. Such representations seem to be difficult to construct efficiently in parallel, since the pruned sets are defined naturally by a sequential process, as shown in Figure 2. In addition, as pointed out at the end of our first approach, the space usage can still be off by a multiplicative $\log \log n$ factor.

Our Final Approach. To construct the search substructure T_i, we limit ourselves to the hypothetical subtree S' consisting of levels $0, 1, 2, \ldots, \lceil (1 - 2^{-i}) \log n \rceil$ of S. This truncation of the lower levels reduces the cumulative space used for the $\lceil \log \log n \rceil - 1$ search substructures T_i to at most a small constant factor times the size of the largest substructure $T_{\lceil \log \log n \rceil - 1}$, as discussed later in Lemma 2. For each i, the truncated lower levels can be traversed sequentially (with one processor) in $O(2^{-i} \log n) = O((\log n)/\log p)$ time. Similar truncation ideas appear often in the parallel computing literature, for example, in evaluating n-node complete binary trees in optimal $O(\log n)$ time with $n/\log n$ processors, and in a cooperative search context in [7].

The problem that remains is how to reduce the storage requirement for each individual T_i. As in our first two approaches we consider the complete binary subtrees of S' of height $h_i = \lfloor \alpha 2^i \rfloor = \Theta(\log p)$, rooted at the nodes in S' on levels $0, h_i, 2h_i, \ldots$. (The subtrees rooted on the lowermost level might be truncated and have height less than h_i.) The value of the fixed constant $0 < \alpha < 1$ will be specified later.

Let U denote one of the subtrees of S', having root u, and let $t \geq 1$ be the number of catalog entries in u's catalog. We denote the catalog entries by

$$c_1, c_2, \ldots, c_t,$$

where $c_t = +\infty$. We define s_i to be $(2b+2)(2b+1)^{h_i}$, where b is the fractional cascading parameter and $h_i = \lfloor \alpha 2^i \rfloor = \Theta(\log p)$. We choose the fixed constant $\alpha > 0$ so that $(2(2b + 1)^2)^\alpha = 2$; in particular, it follows that $0 < \alpha < 0.25$, $s_i = \Theta(2^{(1-\alpha)2^{i-1}})$, $s_i = \Omega(p^{(1-\alpha)/4})$, and $s_i = O(p^{(1-\alpha)/2})$.

One solution to the problem arising in our second approach is to store the nonpruned reaches in an implicit way. We "store" $reach(c, U)$ by associating with each node $u' \in U$ a single catalog entry; the other entries in the catalog of u' that are in c's reach are sufficiently close to this single catalog entry. It is still possible, though, for the stored entries for $reach(c, U)$ and $reach(c', U)$ to overlap, resulting in the same worst-case storage inefficiency as before. For example, if node $u' \in U$ contains only one catalog entry, then that entry will be stored as part of $reach(c, U)$ for every entry c in the catalog of u.

Our final solution is to store the implicitly stored reach sets of only a *sample* of the catalog entries of u. We use the value of s_i defined above as our "sampling factor." We form $m = \lceil t/s_i \rceil$ complete binary trees $U_1, U_2, \ldots, U_m$ having the same skeleton as U (that is, the same shape and nodes), but without catalogs at each node, as illustrated in Figure 3. Each node in each tree has a key value equal to some catalog entry in the corresponding node in U. Let $key[z, U_j]$ denote the key value of node z in U_j. The key values of the root nodes of $U_1, U_2, \ldots, U_m$ are chosen to form an equally spaced sample of the t catalog entries of the root of U:

$$key[u, U_j] = \begin{cases} c_{js_i} & \text{if} \quad 1 \leq j < m, \\ +\infty & \text{if} \quad j = m. \end{cases}$$
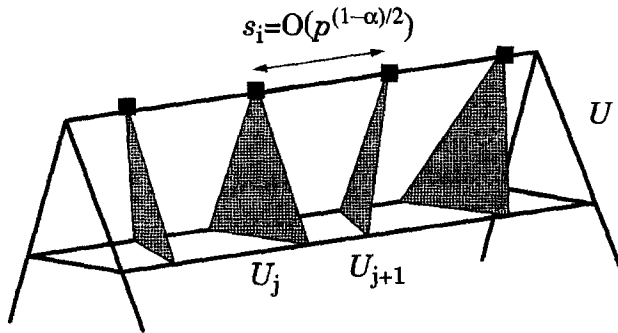
Fig. 3. The key values of the roots of trees $U_1, U_2, \ldots, U_m$ are evenly spaced entries from the catalog of U's root u, where U is a subtree of S of height h_i. For each node z in U, the key values of z's node in $U_1, U_2, \ldots, U_m$ are distinct, although the reaches of the roots of $U_1, U_2, \ldots, U_m$ may still overlap.

If $m = 1$, then the catalog of U's root is too small to sample, and we refer to U_1's root node, which has key value $+\infty$, as a *sparse node*. The remaining key values for each U_j are induced by the bridge structure: for each child w of z in U_j, we define $key[w, U_j] := bridge[z, w, key[z, U_j]]$, which is the entry in w's catalog in U pointed to by the bridge pointer from the entry $key[z, U_j]$ in z's catalog in U.

The purpose of the sampling is to guarantee that the m trees $U_1, U_2, \ldots, U_m$ are disjoint:

LEMMA 1.   *Let z be any node in U, and let $key[z, U_j]$ denote the key value of z when considered as a node in U_j. Then the values $key[z, U_1], key[z, U_2], \ldots, key[z, U_m]$ are distinct entries from z's catalog in U.*

PROOF.   Let w be a node in U and let c_1' and c_2' be two entries that are r entries apart in w's catalog. Let c_1 and c_2 be entries in w's parent v whose bridges point to c_1' and c_2', respectively. Let y_1 and y_2 be search arguments so that $c_1 = find(y_1, v)$ and $c_2 = find(y_2, v)$. We designate $c_1'' = find(y_1, w)$ and $c_2'' = find(y_2, w)$. By property 1 of fractional cascading, c_1'' and c_1' are at most b entries apart in w's catalog, and similarly c_2'' and c_2' are at most b entries apart in w's catalog. By property 2, the reverse bridge pointers in the bidirectional version of S (which we do not use in T') link adjacent entries in w's catalog to entries at most $2b + 1$ apart in v's catalog. Figure 4 illustrates that the entries c_1 and c_2 are at most $(2b + 1)(2b + r + 1) - 1$ entries apart in v's catalog.

To complete the proof, we consider a node z in U that has the same key value in two different binary trees U_j and U_k; that is, $key[z, U_j] = key[z, U_k]$. By the formula we derived in the preceding paragraph, with $r = 0$ and $c_1' = c_2' = key[z, U_j] = key[z, U_k]$, we find that the entries in the catalog of z's parent that point to c_1' and c_2', respectively, are at most $(2b + 1)^2 - 1$ entries apart. Again reapplying the formula, with $r = (2b + 1)^2 - 1$, the entries in the catalog of z's grandparent that point to c_1' and c_2' are at most $(2b + 1)^3 + (2b + 1)^2 - (2b + 1) - 1$ entries apart. By continuing this argument for the maximum h_i levels, we find that any two entries in u's catalog whose bridge pointers lead to a common entry at most h_i levels down in z's catalog must be
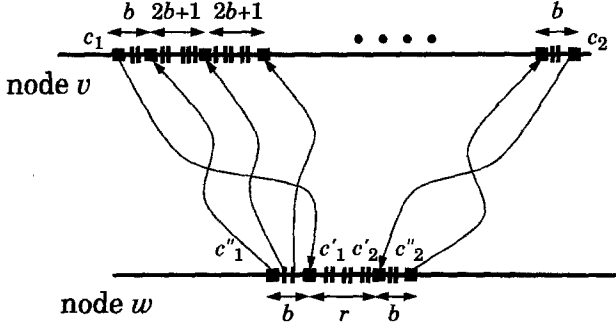
Fig. 4. Proof of Lemma 1. The two catalog entries c_1 and c_2, whose bridges point to entries c'_1 and c'_2 that are r entries apart in w's catalog, can be at most $(2b + 1)(2b + r + 1) - 1$ entries apart in node v's catalog, since each consecutive pair of the $2b + r$ reverse bridges from w's catalog can "fan out" to entries at most $2b + 1$ entries apart in v's catalog.

at most

$$(2b + 1)^{h_i+1} + (2b + 1)^{h_i} - (2b + 1) - 1 < (2b + 2)(2b + 1)^{h_i} \le s_i$$

entries apart in u's catalog. Since the sampled entries in u's catalog are at least s_i entries apart, the nodes corresponding to z in two different binary trees U_j and U_k cannot have the same key value. This proves the lemma. □

We store each of the m trees in compacted form, for example, by using a heap (breadth-first search) ordering. The search substructure T_i consists of the compacted depth-first search forest $U_1, U_2, \ldots, U_m$ described above, for each subtree U of S'. There is a pointer to the root nodes of each compacted forest from the corresponding catalog entry in S, and each node in the forests points to the corresponding node in S.

The following main lemma shows that the sizes of the substructures T_i sum geometrically to $O(n)$:

LEMMA 2. *The total storage space used for T' is $O(n)$.*

PROOF. First we consider the storage space used by T_i. The total size of the trees U_j having a sparse root is bounded by the total number of nodes in S', which is $O(2^{(1-2^{-i})\log n}) = O(n^{1-2^{-i}})$. The total size of trees U_j with nonsparse roots is bounded by the product of the total number of sampled catalog entries, which is $O(n/s_i) = O(n/2^{(1-\alpha)2^{i-1}})$, times the storage space for each U_j, which is $O(2^{h_i}) = O(2^{\alpha 2^i})$. Thus, the total storage required for T_i is

$$O\left(n^{1-2^{-i}} + 2^{\alpha 2^i}\left(\frac{n}{2^{(1-\alpha)2^{i-1}}}\right)\right).$$

Since $\alpha < 0.25$, summing on $0 \le i \le \lceil \log \log n \rceil - 1$ gives $O(n)$. □

By Lemmas 2 and 1, the compacted trees in T' can be formed easily using an EREW PRAM in $O(\log n)$ time with $n/\log n$ processors.

2.2. *Explicit Cooperative Search.* Searching the data structure T' is confined to the substructure T_i for which the number of processors satisfies $2^{2^i} < p \leq 2^{2^{i+1}}$. First we consider explicit searches with query point y. We discuss how to handle implicit searches in Section 2.3. The search procedure works as follows, using a CREW PRAM:

Step 1. Set u to the root of S. We do a cooperative binary search for y in u's catalog to get $\bar{c} = find(y, u)$.

Step 2. [Move to next sampled catalog entry.] Let U be the hypothetical subtree of height h_i rooted at u. We set c to the smallest sampled catalog entry $\geq \bar{c}$. This can be done by assigning $s_i = O(p^{1-\alpha})$ processors to $\bar{c}$ and its $s_i - 1$ successors in u's catalog. Exactly one will be a sampled catalog entry. We denote by U_j the compacted tree whose root has key value c.

Step 3. [Jump $h_i = \Theta(\log p)$ levels.] We assign processors to ranges of catalog entries for each of the nodes v on the search path in U. For $v \in U$ on level l, we assign a total of $s_i(2b + 1)^l$ processors, one processor to each of the following catalog entries of v:

$$c_{k-q-r}, c_{k-q-r+1}, \ldots, c_k, \ldots, c_{k+q},$$

where $c_k = key[v, U_j]$, $q = \frac{1}{2}((2b + 1)^l - 1)$, and $r = (s_i - 1)(2b + 1)^l$. The processor assigned to catalog entry c_g tests whether $c_{g-1} < y \leq c_g$. The entry of the unique processor whose test succeeds is returned as $find(y, v)$.

Step 4. Let u be the leaf of U on the search path. If u is a leaf of S', we go to Step 5. Otherwise, we set $\bar{c}$ to be u's catalog entry returned for $find(y, w)$, and we return to Step 2.

Step 5. Search the remainder of the path sequentially in the fractional cascaded data structure S from u to the leaf.

The following lemma shows that in Step 3 each catalog entry in $reach(c, U_j)$ corresponding to a node along the search path is assigned a processor.

LEMMA 3. *In Step 3 of the explicit search procedure, for each node v on the search path in U, there is a processor assigned to each possible catalog entry that is a candidate value of $find(y, v)$.*

PROOF. For any node u in U and u's left or right child w, the bridge pointers from two catalog entries that are r entries apart in the u's catalog can point to entries in w's catalog that are at most $2b + 1$ entries apart. The parameter q is chosen large enough to compensate for the "fan out" of fractional cascading search. The parameter r biases the assignment of processors to the left to compensate for the fact that we shifted right from $\bar{c}$ to c because of the sampling.   □

We finish this section by completing the proof of Theorem 1 for the case of explicit cooperative search. The total number of processors used in Step 3 is

$$\sum_{1 \leq l \leq h_i} s_i(2b + 1)^l \leq 2s_i(2b + 1)^{h_i} \leq s_i^2.$$

By definition of α, this bound is $O(p^{(1-\alpha)}) = O(p)$. The cooperative binary search in Step 1 takes $O((\log n)/\log p)$ time on a CREW PRAM. In Steps 2–4 a subtree of height $h_i = \lfloor \alpha 2^i \rfloor = \Theta(\log p)$ is processed in constant time. The total number of iterations is $O((\log n)/\log p)$. In Step 5 the number of nodes in the search path searched sequentially using S is at most $2^{-i}\log n \le (\log n)/\log p$, thus giving us the desired running time.

2.3. *Implicit Cooperative Search.*

In the basic form of implicit search, as defined in Section 2, the search path is defined implicitly. For search argument $q = (x, y)$, the left bridge is taken at node v if the function call $branch(q, find(y, v))$ returns *left*; otherwise, $branch(q, find(y, v))$ returns *right*, and the right branch is taken.

The search procedure in Step 3 in Section 2.2 is modified as follows: Processors must be assigned to all nodes of U. The function evaluations $branch(q, find(y, v))$ for the nodes are stored in an array. By the consistency assumption for basic implicit search, as stated in Section 2, the node on the search path at level l can be identified as the unique node v on level l for which $branch(q, find(y, w)) = right$ and $branch(q, find(y, v)) = left$, where w is the leaf node preceding v. This can be determined in constant time on a CREW PRAM.

The number of processors needed increases to

$$\sum_{1 \le l \le h_i} s_i 2^{l+1} (2b + 1)^l \le 2^{h_i} s_i^2.$$

By definition of α, this bound is $O(p^\alpha p^{(1-\alpha)}) = O(p)$.

More complicated forms of implicit search, such as the type that arises in point location, are discussed in Section 3.

2.4. *General Trees.*

First, we consider trees with bounded degree.

THEOREM 2. *A bounded-degree rooted tree with catalogs of total size n and $O(n)$ nodes can be preprocessed in $O(\log n)$ time using $n/\log n$ processors on an EREW PRAM so that a subsequent explicit cooperative search along a path of length k can be done in time $O((\log n)/\log p + k/(p^{1-\varepsilon}\log p))$ using a CREW PRAM with p processors, for any $1 \le p \le n$, on an $O(n)$-space data structure (for any positive constant $0 < \varepsilon \le 1$).*

PROOF. We partition the search path into $k/\log n$ subpaths of length $\log n$ each. We assign p^ε processors to each subpath, so that a group of $p^{1-\varepsilon}$ subpaths can be searched concurrently in time $O((\log n)/\log p^\varepsilon) = O((\log n)/\log p)$. Since there are $\lceil k/(p^{1-\varepsilon}\log n) \rceil$ groups of subpaths, the total time complexity is $O((\log n)/\log p + k/(p^{1-\varepsilon}\log p))$. □

When the maximum degree d of the tree is not bounded, we can use standard techniques (see, for example, [1]) to replace each level of the tree by $\log d$ levels of a binary tree. This gives us the following modifications of Theorems 1 and 2:

THEOREM 3. *Trees of degree d can be preprocessed as in Theorems 1 and 2, except that the cooperative search time increases to $O(\log n (\log d)/\log p)$ and $O((\log n)/\log p + k(\log d)/(p^{1-\varepsilon}\log p))$, respectively.*

3. Point Location. In this section we consider the fundamental point-location problem of computational geometry. We first discuss planar point location, and then extend our results to spatial point location.

3.1. *Planar Point Location.* The well-known planar point-location problem is defined as follows:

PLANAR POINT LOCATION. Let S be a planar subdivision with n vertices. Find the region of S containing a given query point $q = (x, y)$.

We assume that the subdivision S is monotone and is given with a standard representation such as doubly connected edge lists [15]. Nonmonotone subdivisions are handled via a preliminary triangulation that takes $O(\log n)$ time using a CREW PRAM with n processors [1], [18], [10], or a CRCW PRAM with $n/\log n$ processors [11]. Parallel triangulation can also be performed using a randomized CREW PRAM algorithm that runs in $O(\log n \log \log n)$ time and does $O(n)$ expected work [6], [5].

The *bridged separator tree* [13], [9] uses $O(n)$ space and supports point-location queries in S in $O(\log n)$ time. It is a balanced binary tree T with catalogs where searches are performed implicitly. We recall that a *separator* σ of S is a monotone chain from $-\infty$ to $+\infty$ in the vertical direction. Let $r_1, r_2, \ldots, r_f$ be the regions of S numbered such that $i < j$ whenever region r_i shares an edge with r_j and is to the left of r_j (see Figure 5(a)). The common boundary of the regions with index $\leq i$ and of the regions with index $> i$ is a separator of S, which we denote σ_i (see Figure 5(a)). Each leaf of T represents a region of S, and each internal node represents a separator, such that the inorder sequence of the nodes of T is given by (see Figure 5(b))

$$r_1, \sigma_1, r_2, \sigma_2, \ldots, \sigma_{f-1}, r_f.$$

We denote with σ_0 and σ_f fictitious separators at $-\infty$ and $+\infty$ in the horizontal direction, respectively, which represent the left and right boundary of S (see Figure 5(a)). For convenience, we often equate a node of T with the corresponding separator or region.

Point location in S can be done sequentially by tracing a path in T from the root to the leaf r_i containing the query point $q = (x, y)$. At each internal node σ_j we discriminate q against separator σ_j and branch left or right according to whether q is to the left or right of σ_j (see Figure 5(b)). The discrimination of q against σ_j is performed by locating y in the set of y-coordinates of the edges of σ_j, which identifies the edge of σ_j horizontally visible from q.

Each edge e of S belongs to a range of separators $\{\sigma_j : i \leq j \leq k - 1\}$, where r_i and r_k are the regions to the left and right of e, respectively. For space efficiency, edge e is stored only once at the least common ancestor σ_l of r_i and r_k in T, and is called a *proper edge* of σ_l. (Node σ_l is also the least common ancestor of nodes σ_j, for $i \leq j \leq k - 1$.) The set of proper edges of a separator consist of a collection of chains and "gaps," sorted from bottom to top. Figure 5(a) shows the sets of proper edges for each separator. The catalog of a node σ_j of T is the set of y-coordinates of the proper edges of σ_j.

By Euler's formula, $f = O(n)$, so that T has $O(n)$ nodes. Each edge of S is stored only once as a proper edge, so that the overall size of the catalogs stored at the nodes of T is $O(n)$. Hence, the bridged separator tree T uses $O(n)$ space.
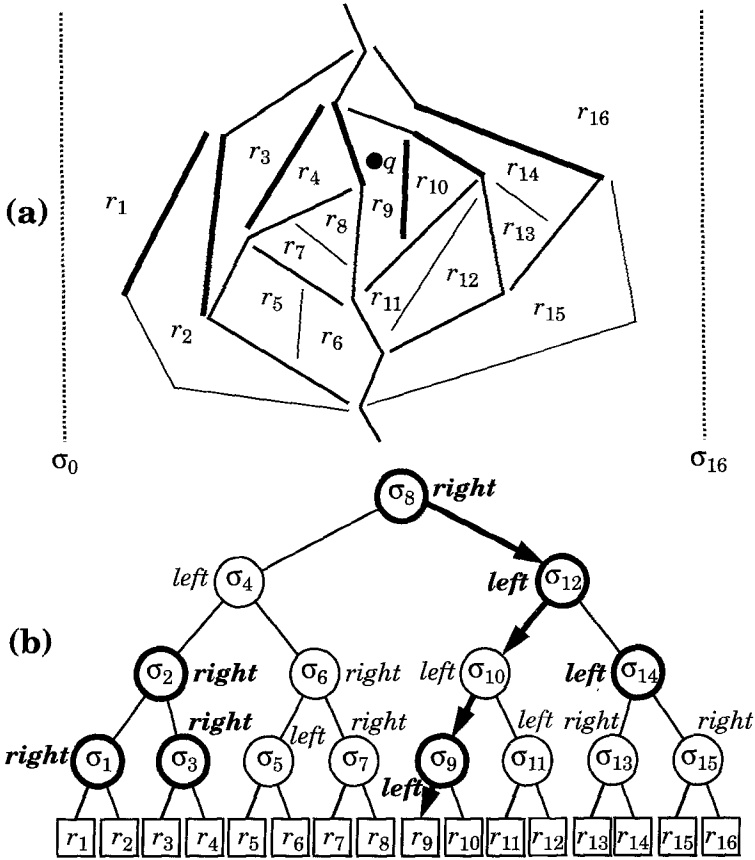
Fig. 5. Example of sequential point-location data structure. (a) Monotone subdivision and a query point $q = (x, y)$. The edges whose vertical span includes y are drawn with thick lines. The sets of proper edges for each separator are displayed as (contiguous) chains. (b) Bridged separator tree for the subdivision of part (a). Each node represents a separator and stores the proper edges of that separator, which are displayed as a chain in part (a). The search path and the active nodes for point q are shown with thick lines. The values of the *branch* function computed for point q are shown next to each node. Note that this *branch* function does not satisfy the consistency assumption, which is violated at nodes σ_4, σ_5, σ_{13}, and σ_{15}.

Given a query point $q = (x, y)$, we say that an internal node σ_l of $\mathcal{T}$ is *active* if separator σ_l has a proper edge whose vertical span includes y, i.e., the catalog entry $find(y, \sigma_l)$ represents a proper edge, and we denote such an edge with $e_l(q)$. If instead σ_l has a gap at the horizontal level of q, we say that node σ_l is *inactive*, and denote such a gap with $g_l(q)$. In the example of Figure 5 the active nodes are σ_1, σ_2, σ_3, σ_8, σ_9, σ_{12}, and σ_{14}.

The *branch* function for an active node σ_i is easily computed in constant time from q and $e_i(q)$. In a sequential point-location search the *branch* function for an inactive node σ_j can be stored in every gap of σ_j. Namely, if the search has reached the inactive node σ_j, the query point q has already been discriminated earlier on against separator σ_j. Hence, the value $branch(q, find(y, \sigma_j))$ depends only on the gap $g_j(q)$ given by $find(y, \sigma_j)$, as follows. Let σ_k be the lowest ancestor of σ_j such that some edge of gap $g_j(q)$ is a proper

edge of σ_k. We set *branch* = *left* if $j < k$, and *branch* = *right* if $j > k$. See the example of Figure 5.

However, the implicit cooperative search technique of Section 2.3 cannot be immediately applied to the bridged separator tree since the consistency assumption is not verified by the aforementioned *branch* function. For example, in Figure 5(b) node σ_4 is to the left of the search path for point q but its *branch* function returns *left* instead of *right*, and node σ_{13} is to the right of the search path for point q but its *branch* function returns *right* instead of *left*. We now discuss how to overcome this obstacle, and show how to compute a different *branch* function that verifies the consistency assumption.

Following the general approach of Section 2, the location of a query point consists of a sequence of "hops," each of which allocates processors to all the nodes of a subtree U of $\mathcal{T}$ of depth $\Theta(\log p)$. We store with each edge e of S the minimum and maximum indices of the separators containing e, denoted $\min(e)$ and $\max(e)$, respectively. Let $q = (x, y)$ be the query point. During the search we keep track of indices L and R such that q is between separators σ_L and σ_R. Initially, $L = 0$ and $R = f$.

Let $\mathcal{S}(U)$ denote the planar subdivision determined by the proper edges stored at the nodes of U. A hop is performed with the following steps (see Figure 6):

1. Allocate processors to all the nodes of U, and compute catalog entries $find(y, \sigma_l)$, for each node σ_l of U. Catalog entry $find(y, \sigma_l)$ identifies either proper edge $e_l(q)$ or gap $g_l(q)$, depending on whether node σ_l is active or inactive.
2. The processor allocated to an active node σ_i discriminates q against proper edge $e_i(q)$ of σ_i and sets $branch(q, find(y, \sigma_i))$ equal to *left* or *right* depending on whether q is to the left or to the right of $e_i(q)$.
3. Allocate processors to all the pairs of nodes of $U \cup \{\sigma_L, \sigma_R\}$ and determine the unique pair of nodes (σ_i, σ_j) such that:
   (a) Both σ_i and σ_j are active, with $i < j$.
   (b) $e_i(q)$ and $e_j(q)$ are on the boundary of the same region of $S(U)$.
   (c) $branch(q, find(y, \sigma_i)) = right, branch(q, find(y, \sigma_j)) = left$.
4. Set $L := i$ and $R := j$.
5. Allocate processors to all the nodes of U. For each inactive node σ_k of U, the processor allocated to σ_k compares index k with $\max(e_L(q))$. If $k \leq \max(e_L(q))$, it sets $branch(q, find(y, \sigma_k)) := right$; else, it sets $branch(q, find(y, \sigma_k)) := left$.
6. A unique pair of nodes of U (consecutive in the inorder sequence of U) is determined whose *branch* functions evaluate to *right* and *left*, respectively. Such a pair of nodes identifies the search path within the subtree U.

We apply this algorithm to the example of Figure 6. Each region of $\mathcal{S}(U)$ is the union of regions r_{2i-1} and r_{2i} of S, $i = 1, \cdots, 8$. The active nodes computed in Step 1 are σ_2, σ_8, σ_{12}, and σ_{14}, and are drawn with thick lines. The *branch* function computed at the active nodes in Step 2 is shown in a bold font. The pair of nodes (σ_8, σ_{12}) and their associated proper edges computed in Step 1 are drawn with extra-thick lines. In Step 4 we set $L = 8$ and $R = 12$. The *branch* function computed at the inactive nodes in Step 5 is shown in an italic font. For instance, the branch function at node σ_{10} is *left* since $\max(e_L(q)) = \max(e_8(q)) = 8$. Finally, the pair of nodes (σ_8, σ_{10}) is computed in Step 6, which identifies the search path within the subtree U, drawn with arrows.
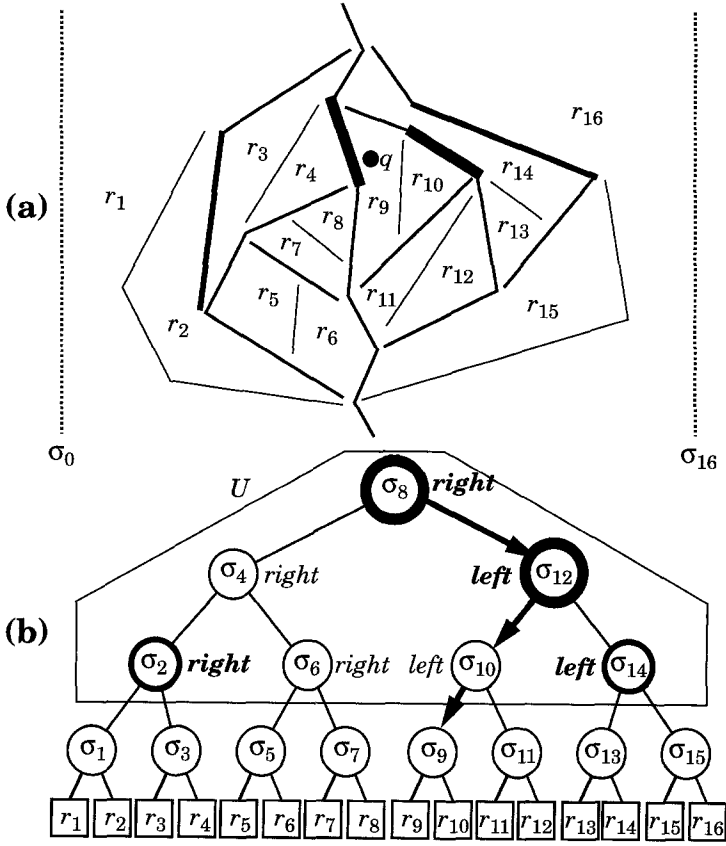
Fig. 6. Example of a parallel point-location data structure. (a) Monotone subdivision and a query point $q = (x, y)$. The sets of proper edges of each node of the tree are displayed as chains. The edges whose vertical span includes y and that are proper edges of a node of U are drawn with thick lines. The edges that verify condition 3(b) are drawn with extra-thick lines. (b) Bridged separator tree for the subdivision of part (a) and subtree U. The search path and the active nodes of U for point q are shown with thick lines. The values of the *branch* function computed for point q at the nodes of U are shown next to each node. Note that this *branch* function satisfies the consistency assumption.

THEOREM 4. *A planar subdivision $\mathcal{S}$ with n vertices can be preprocessed in $O(\log n)$ time on a CREW PRAM with n processors (or a CRCW PRAM with $n/\log n$ processors) so that subsequent cooperative point-location queries can be done in time $O((\log n)/\log p)$ using a CREW PRAM with p processors, for any $1 \le p \le n$, on an $O(n)$-space data structure. If the subdivision $\mathcal{S}$ is monotone, the preprocessing can be done with $n/\log n$ processors on an EREW PRAM.*

PROOF. Each step is executed in $O(1)$ time. We only discuss Step 3, since the remaining steps are straightforward to analyze. The allocation of processors in Step 3 can be done by suitably reducing the parameter α that controls the size of U (see Section 2). Condition 3(b) of Step 3 can be tested in $O(1)$ time by one processor without precomputing the embedding of $\mathcal{S}(U)$, as follows. Note that each region of $\mathcal{S}(U)$ is the union of 2^h

consecutive regions of S, for some integer h that depends on U, i.e., the region of $S(U)$ to the right of σ_{i-1} is the union of $r_i, r_{i+1}, \ldots, r_{i+2^h-1}$ (in the example of Figure 6, $h = 1$). Hence, the region of $S(U)$ immediately to the right of $e_i(q)$ is the same as the region of $S(U)$ immediately to the left of $e_j(q)$ if and only if $\min(e_j(q)) - \max(e_i(q)) \leq 2^h$. The parallel construction of the data structure uses a variation of the techniques described in [17].   □

3.2. *Spatial Point Location.* The result of Theorem 4 can be extended to spatial point location in a cell complex such that the vertical dominance relation among the cells is acyclic. This problem was previously solved using an $O(n)$-space data structure (called a *canal tree*) that supports sequential point-location search in $O(\log^2 n)$ time [2]. For cooperative search we use a data structure based on *separating surfaces*, a three-dimensional extension of separators. This data structure can be efficiently constructed in parallel if a topological ordering of the cells (with respect to the dominance relation) is available.

THEOREM 5. *Let C be a spatial cell complex with n facets such that the vertical dominance relation on the cells is acyclic, and assume that a topological ordering of the cells according to the vertical dominance relation is given. An $O(n)$-space data structure supporting cooperative point-location queries in $O((\log^2 n)/\log^2 p)$ time with a p-processor CREW PRAM (for any $1 \leq p \leq n$) can be constructed in $O(\log n)$ time by a CREW PRAM with n processors (or a CRCW PRAM with $n/\log n$ processors).*

PROOF. The data structure is a three-dimensional extension of the separator tree. Let $c_1, c_2, \ldots, c_r$ be the cells of C, labeled according to the topological ordering. We define χ_i as the set of facets f such that the cell below f has index $\leq i$ and the cell above f has index $> i$. It can be shown that χ_i is a continuous surface such that any vertical line intersects χ_i in at most one point. Hence, we call χ_i a *separating surface* of C. Let T be a balanced binary tree with n leaves. The ith leaf of T from left-to-right is associated with cell c_i, and each internal node is associated with a separating surface χ_j of C. Whenever convenient, we use the same name for a node of T and the associated cell or separating surface of C. A facet f belongs in general to a nonempty interval of separating surfaces. However, f is assigned to the least common ancestor χ_j of the cells above and below f, respectively, and is called a *proper facet* of χ_j.

Using T, spatial point location can be done analogously to the planar case by tracing a path from the root to a leaf, where at each internal node χ_j we determine whether the query point is above or below the separating surface χ_j. Discriminating the query point with respect to χ_j can be done by means of planar point location. Since each facet is stored exactly once, the space requirement is $O(n)$. The parallel construction of the above data structure is performed in two steps as follows:

1. We construct the sets of proper facets of each internal node of C. This can be done by performing least common ancestor queries in T.
2. For each j we consider the planar subdivision S_j generated by the projection of the proper facets of χ_j on the xy-plane, and we construct for S_j the cooperative point-location data structure of Theorem 4.

The cooperative search consists of a sequence of $O((\log n)/\log p)$ hops, each traversing $\Theta(\log p)$ levels of $\mathcal{T}$ within a subtree U. Each hop is performed in $O((\log n)/\log p)$ time by doing in parallel cooperative point-location queries in the planar subdivisions stored at the nodes of the current subtree U. $\qquad\square$

COROLLARY 1.   *Let C be the cell-complex with n facets induced by the Voronoi diagram of a set of sites in three-dimensional space. An $O(n)$-space data structure supporting cooperative point-location queries in $O((\log^2 n)/\log^2 p)$ time, for any $1 \leq p \leq n$, can be constructed by an EREW PRAM in $O(\log n)$ time using n processors.*

PROOF.   The cell complex induced by a Voronoi diagram is acyclic and a topological ordering of the cells can be obtained by sorting the associated sites according to their z-coordinates. $\qquad\square$

4. Further Applications.   The cooperative search technique described in Section 2 can be applied to a variety of geometric retrieval problems, including the ones defined below:

ORTHOGONAL RANGE SEARCH.   Let P be a set of n points in the plane. Report the points of P inside a given query range r (rectangle with sides parallel to the Cartesian axes).

ORTHOGONAL SEGMENT INTERSECTION.   Let V be a set of n vertical segments in the plane. Report the segments of V intersected by a given horizontal query segment h.

POINT ENCLOSURE.   Let R be a set of n ranges in the plane. Report the ranges of R containing a given query point q.

We consider two models of cooperative retrieval: *direct retrieval* consists of marking the items to be reported; *indirect retrieval* consists of returning a pointer to a linked list of the items to be reported.

THEOREM 6.   *There are $O(n \log n)$-space data structures for the above problems that can be constructed in time $O(\log n)$ on an EREW PRAM with n processors, such that cooperative retrieval can be done with the following time bounds, where k is the number of items reported:*

1. $O((\log n)/\log p + \log\log n + k/p)$ *for direct retrieval on a CREW PRAM with p processors.*
2. $O((\log n)/\log p)$ *for indirect retrieval on a CRCW PRAM with p processors.*

PROOF.   All of these problems can be solved by using data structures consisting of a balanced binary tree with $O(n)$ nodes, each containing a catalog, such that the overall size of the catalogs is $O(n \log n)$ [15].

   We describe only the data structure for the Orthogonal Segment Intersection Problem. The data structures for the other problems are constructed with a similar approach. We set up a segment tree $\mathcal{T}$ on the y-coordinates of the segments in V. Each node of $\mathcal{T}$ stores the catalog of the segments allocated to that node, sorted from left to right. The retrieval algorithm consists of identifying a root-to-leaf path by means of a dictionary search on the y-coordinate of the query segment h, and then performing two explicit iterative searches along such paths on the x-coordinates of the extremes of the endpoints of h. By Theorem 1, this procedure allows us to identify the range of items to be reported in each of the catalogs of the search path in time $O((\log n)/\log p)$.

   For direct retrieval we need to compute a prefix sum to allocate processors to the items to be reported, which takes $O(\log \log n)$ time. Note that if $p = o((\log n)/\log \log n)$, the prefix sum computation can be done in time $O((\log n \log \log n)/(p \log \log n))$, which is $O((\log n)/\log p)$. For indirect retrieval, we identify the catalogs that do not contain items to be reported in order to link the list correctly. Whenever $p = \Omega(\log^2 n)$, we use concurrent write to do this in $O(1)$ time. Otherwise, we perform a prefix computation, which takes $O((\log n)/\log p)$ time.   □

   The result of Theorem 6 can be extended to higher dimensions.

COROLLARY 2.   *For constant $d \geq 1$, there are $O(n \log^{d-1} n)$-space data structures for orthogonal range search and point enclosure in d dimensions that can be constructed in time $O(\log^{d-1} n)$ on an EREW PRAM with n processors, such that cooperative retrieval can be done with the following time bounds, where k is the number of items reported:*

1. $O(((\log n)/\log p)^{d-1} + \log \log n + k/p)$ *for direct retrieval on a CREW PRAM with p processors.*
2. $O(((\log n)/\log p)^{d-1})$ *for indirect retrieval on a CRCW PRAM with p processors.*

PROOF.   We outline the data structure for range searching. The data structure for point enclosure is similar. If $d = 2$, we use the result of Theorem 6. Otherwise $(d > 2)$ we construct a balanced tree $\mathcal{T}$ storing the points sorted by their first coordinate. Each node μ of $\mathcal{T}$ has a pointer to a $(d-1)$-dimensional range-searching data structure for the points in the subtree of μ projected along the first coordinate axis. The retrieval algorithm consists of $\Theta((\log n)/\log p)$ phases, each of which jumps $\alpha \log p$ levels of $\mathcal{T}$ by recursively and concurrently solving p^{α} $(d-1)$-dimensional subproblems associated with the nodes of a subtree of $\mathcal{T}$ of height $\alpha \log p$.   □

5. Conclusions.   We have presented efficient algorithms for cooperative search in fractional cascaded data structures whose underlying graph is a rooted tree. Also, we have given applications of our results to a variety of geometric search problems. Open problems include:

1. Develop optimal techniques for search paths of length $\Omega(\log n)$.
2. Extend the results to acyclic or general graphs with catalogs, without sacrificing storage or search efficiency.

3. Extend the results to generalized search paths (the catalogs to be searched are at the nodes of a subgraph).
4. Study cooperative update in dynamic data structures. Dynamic fractional cascading can be done sequentially with update time $O(\log \log n)$ [14].

References

[1] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: a technique for designing parallel algorithms. *SIAM J. Comput.*, 18:499–532, 1989.

[2] B. Chazelle. How to search in history. *Inform. Control*, 64:77–99, 1985.

[3] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[4] B. Chazelle and L. J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1:163–191, 1986.

[5] K. L. Clarkson, R. Cole, and R. E. Tarjan. Erratum: Randomized parallel algorithms for trapezoidal diagrams. *Internat. J. Comput. Geom. Appl.*, 2(3):341–343, 1992.

[6] K. L. Clarkson, R. Cole, and R. E. Tarjan. Randomized parallel algorithms for trapezoidal diagrams. *Internat. J. Comput. Geom. Appl.*, 2(2):117–133, 1992.

[7] N. Dadoun and D. G. Kirkpatrick. Cooperative subdivision search algorithms with applications. *Proc. 27th Allerton Conf. on Communication, Control, and Computing*, pp. 538–547, 1989.

[8] M. Edahiro, I. Kokubo, and Ta. Asano. A new point-location algorithm and its practical efficiency: comparison with existing algorithms. *ACM Trans. Graph.*, 3:86–109, 1984.

[9] H. Edelsbrunner, L. J. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM J. Comput.*, 15:317–340, 1986.

[10] M. T. Goodrich. Triangulating a polygon in parallel. *J. Algorithms*, 10:327–351, 1989.

[11] M. T. Goodrich. Planar separators and parallel polygon triangulation. *Proc. 24th ACM Symp. on Theory of Computing*, pp. 507–516, 1992.

[12] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12:28–35, 1983.

[13] D. T. Lee and F. P. Preparata. Location of a point in a planar subdivision and its applications. *SIAM J. Comput.*, 6:594–606, 1977.

[14] K. Mehlhorn and S. Näher. Dynamic fractional cascading. *Algorithmica*, 5:215–241, 1990.

[15] F. P. Preparata and M. I. Shamos. *Computational Geometry: an Introduction.* Springer-Verlag, New York, 1985.

[16] M. Snir. On parallel searching. *SIAM J. Comput.*, 14(3):688–708, 1989.

[17] R. Tamassia and J. S. Vitter. Parallel transitive closure and point location in planar structures. *SIAM J. Comput.*, 20(4):708–725, 1991.

[18] C. K. Yap. Parallel triangulation of a polygon in two calls to the trapezoidal map. *Algorithmica*, 3:279–288, 1988.