PARALLEL FRACTIONAL CASCADING ON A HYPERCUBE MULTIPROCESSOR

FRANK DEHNE[1], AFONSO FERREIRA[2], AND ANDREW RAU-CHAPLIN[1]

[1] *Center for Parallel and Distributed Computing, School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6.*

[2] *Laboratoire de l'Informatique Parallele, Ecole Normale Superieure de Lyon, 69364 Lyon cedex 07, France.*

ABSTRACT: In this paper we present a parallel implementation of fractional cascading for hypercube multiprocessors. We show that, if the underlying catalog graph is *monotone* (as defined in the paper), N multiple look-up queries (including catalog look-ups) can be executed independently, in parallel, in time $O(t_S \log N)$ on a hypercube multiprocessor of size N, where t_S is the sequential time complexity for one multiple look-up query. We apply our method to various problems for which we obtain new efficient hypercube algorithms.

1 INTRODUCTION

For the design of parallel algorithms, many researchers prefer the PRAM model over processor network models. Efficient parallel algorithms for the PRAM are not constrained by the need for efficient data routing mechanisms; furthermore, the PRAM memory can be used to store and access data structures in essentially the same way as on a standard sequential machine. Once the processors have collectively built a data structure, each of them can individually execute a query on this structure (as if it was a single processor architecture) without interfering with the other query processes. This method allows to apply results from sequential data structures to the design of PRAM algorithms (see, e.g., [ACG], [ACGD], [AG], [DK], [G]).

For processor networks, the parallel execution of independent queries on one joint data structure is not as straight forward. Algorithms designed for processor networks are usually not as elegant; they use only very simple data structures, if any, and are mainly concerned with solving the routing and collision avoidance problem. A more elegant approach is to simulate PRAM algorithms on processor networks; the obtained results are however in most cases less efficient than algorithms designed directly for specific networks.

In this paper we show that for hypercube multiprocessors it is also possible to design elegant yet extremely efficient algorithms based on parallel implementations of data structures.

We study the class of those data structures that are special cases of fractional cascading [CG1]. Consider a catalog graph of size N and bounded (fixed) degree, and a set of N multiple look-up queries along paths of length at most p [CG1]. We show that, if the graph is *monotone* (to be defined in Section 2), such N multiple look-up queries (including catalog look-ups) can be executed independently, in parallel, in time $O(t_S \log N)$ on a hypercube multiprocessor of size N, where $t_S = O(p + \log N)$ is the sequential time complexity for one multiple look-up query as described in [CG1]. Note that, our solution allows an arbitrary number of search queries to access the same node and its catalog at the same time. (This can not be achieved by, e.g., embedding graphs into hypercubes.) The requirement that the graph needs to be monotone is not overly restrictive; such graphs include e.g. all k-nary trees (for fixed k).

In Section 3 we apply our method to the problem of determinating all intersections of n lines with a simple polygonal path of length n; the Multi Slanted Range Search problem is solved in Section 4. In Section 5, we implement a segment tree [BW], [M], [PS] for next element search (n segments, m queries, $N = \max\{n,m\}$) on a hypercube of size N log N. Our approach provides $O(\log^2 N)$ time hypercube algorithms for the next element search problem, the trapezoidal map construction problem, and the triangulation problem.

2 PARALLEL FRACTIONAL CASCADING

2.1 REVIEW OF SEQUENTIAL FRACTIONAL CASCADING

Consider a directed graph G=(V,E) with bounded (fixed) degree; i.e., there exist constants μ_{out} and μ_{in} such that for every vertex $v \in V$, the out-degree and in-degree of v are at most μ_{out} and μ_{in}, respectively. Assume that G is connected and has n vertices (and therefore O(n) edges). As in [CG1], we associate with each vertex v of G a *catalog* C_v consisting of an

F. Dehne, A. Ferreira, and A. Rau-Chaplin, "Parallel fractional cascading on a hypercube multiprocessor," in Proc. *Allerton Conference on Communication, Control and Computing*, 1989, pp. 1084-1093.

ordered collection of records from a totally ordered domain U. The graph G together with its catalogs is referred to as a *catalog graph* of size N=max{n,s}, where s is the sum of the sizes of all catalogs.

A *path* π in G (of length p) is a sequence of vertices $v_1, v_2, ..., v_p$ such that for each $1 \le i < p$, either $(v_i, v_{i+1}) \in E$ [*forward edge*] or $(v_{i+1}, v_i) \in E$ [*backward edge*]. A *multiple look-up query* is a pair (q,π) where q is a value of U and π is a path in G. For each catalog C we denote by $\sigma(q,C)$ the *successor of q in C*, that is the first record of C whose value is greater than or equal to q. The *iterative search problem* consists of executing a multiple look-up query (q,π) by following the path π in G and determining for every vertex v on the path the successor of q in C_v. The path π is assumed to be specified *on-line*, that is the successor v_{i+1} of the vertex v_i in π is only known after the query has reached v_i and determined $\sigma(q,C_{v_i})$.

The above definition of *fractional cascading* follows the one given in [CG1], but assumes fixed degree catalog graphs (instead of catalog graphs with locally bounded degree, see [CG1]) and uses a restricted definition of the considered type of paths. For the above case it follows from [CG1] that in O(N) time and space it is possible to construct, for the standard sequential machine model, a data structure that allows to solve the iterative search problem for one single multiple look-up query with path length p in time O(p + log N).

2.2 A Hypercube Implementation of Fractional Cascading

We now study how to obtain a parallel implementation of fractional cascading on a hypercube multiprocessor of size N; that is a set of N synchronized processors PE(i), $0 \le i \le N-1$, where two processors PE(i) and PE(j) are connected by a communication link if the binary representations of i and j differ in exactly one bit. (We assume that every processor has the same constant number of registers; for every register *A* available at each processor, *A(i)* will refer to register A at processor PE(i).)

Compared to the sequential methods presented in [CG1], we need to impose a restriction on the type of catalog graphs. Consider a catalog graph G=(V,E) of size N with n vertices. G is called a *monotone catalog graph* if there exists a (one-to-one) index function *Index*: $V \Rightarrow \{1, ..., n\}$ with the following property: if (v,v') and (w,w') are two edges of G with $Index(v) < Index(w)$ then $Index(v') \le Index(w')$.

In the remainder of this section we will show how, given a *monotone* catalog graph of size N and a set $Q=\{q_1,...,q_N\}$ of N multiple look-up queries along paths of length at most p, these N multiple look-up queries can be executed independantly, in parallel, in time $O((p + \log N) \log N)$ on a hypercube multiprocessor of size N. We assume, w.l.o.g., that $N=2^d$; all results obtained can be easily generalized. We first give an overview of the algorithm, including the assumed initial configuration of the hypercube, and how the results of the N multiple iterative search queries are reported. We will then present some more details for the different phases of the algorithm.

Algorithm Overview

The graph G is assumed to be stored in the hypercube such that each vertex v with Index(v)=i is stored in register v(i) of processor PE(i). For every vertex v, let *pred(v)* and *succ(v)* be the sets of predesessors and successors in G; i.e., the sets of at most μ_{in} and μ_{out} vertices w such that $(w,v) \in E$ and $(v,w) \in E$, respectively. We assume that register v(i) contains fields v.index(i), $v.successor_1(i)$, ..., $v.successor_{\mu_{out}}(i)$ and $v.predecessor_1(i)$, ..., $v.predecessor_{\mu_{in}}(i)$ storing Index(v) and (in sorted order) the indices of the vertices in succ(v) and pred(v), respectively.

We assume that each PE(i) also has a register c(i) to store a catalog record. The catalogs are stored in sorted order with respect to the index of the associated vertices, and each catalog is internally sorted with respect to the order on U. Each record c(i) contains a field c.index(i) storing the index of the associated vertex of G.

Every register v(i) also has a field v.EndCat(i) storing the address (processor number) of the last record of the associated catalog.

The set $Q=\{(q_1,\pi_1), ..., (q_N,\pi_N)\}$ of N multiple look-up queries is given as follows: Every processor PE(i) stores in its register q(i) one arbitrary query value q_j. The search paths π_j ($1 \le j \le N$) are determined *on-line* by the following two functions

- start: $U \Rightarrow \{1, ..., n\}$
- g: $V \times U \times U \Rightarrow \{-\mu_{in},...,-1,1,...,\mu_{out}\}$.

For each q_j, start(q_j) is the index of the first vertex v_1 in its path π_j.

Assume that the x^{th} vertex v_x of π_j is stored in register v(i) of processor PE(i), and that the successor $\sigma(q_j,C_{v_x})$ of q_j in C_{v_x} has been determined; let $y = g(v(i), q_j, \sigma(q_j,C_{v_x}))$. If $y<0$ then $v.predecessor_{-y}(i)$ is the index of the $(x+1)^{st}$ vertex v_{x+1} of π_j; otherwise, $v.successor_y(i)$ is the index of the $(x+1)^{st}$ vertex.

It is required that both functions, start and g, can be calculated in constant time.

Multi Iterative Search:

(1) Phase 1: Match every query with the 1st node in its search path and perform the respective catalog lookup.
(2) For x := 2 to p do
(3) Phase x: Match every query with the x^{th} node in its search path and perform the respective catalog lookup.

Figure 1: Global Structure of the Multi Iterative Search Algorithm

The global structure of the multi iterative search algorithm is described in Figure 1. The iterative search processes for all N queries $q_1, ..., q_N$ are executed in p phases; each phase moves all queries one step ahead in their search paths.

The general idea is that, in Phase x ($1 \le x \le p$), instead of routing the queries to the respective nodes (possibly resulting in collisions), these nodes are duplicated and routed to the respective queries. In order to obtain the desired time complexity, the algorithm first permutes the queries (in registers q(i)) such that they are sorted with respect to the index of the x^{th} node in their search path. It then creates, in registers v'(i), copies of the respective nodes such that each processor PE(i) containing a query q_j in its register q(i), contains in its register v'(i) a copy of the x^{th} node in the search path of q_j (we will call this a *match* of q_j with the x^{th} node in its search path). Finally, for each node v all queries that have v as the x^{th} vertex in their search path and C_v are merged into one sorted list, and for each query its successor in C_v is determined.

In the following we will present some details of Phase 1 and Phase x ($2 \le x \le p$), respectively. The first phase is different from the remaining phases. When ordering the queries with respect to the index of the first node in their search path, the first phase has to start with an arbitrary permutation of the queries, whereas each subsequent phase will utilize the ordering of the previous phase (in order to improve the time complexity of the algorithm).

The algorithm assumes that, in addition to the registers mentioned above, every processor PE(i) also has a register v'(i) to store another vertex of G as well as other auxiliary registers R(i), N(i), q'(i), q"(i), c'(i), N'(i), N"(i), LS(i), Shift(i) and Dest(i).

Phase 1 of the Multi Iterative Search Algorithm

An outline of Phase 1 is given in Figure 2. The algorithm consists of five steps; consult Appendix A for a describtion of the standard hypercube operations referred to in the figure.

Phase 1:

(1) Every PE(i): N(i):=f(Start,q(i),∞)
(2) Sort([q(i), N(i)], N(i))
(3) MoveVerticesToQueries
(4) SelectCatalogs
(5) SearchCatalogsForQueries

Figure 2. Outline of Phase 1.

First, every processor PE(i) calculates the index of the first node in the search path of its query q(i), and stores this value in the register N(i). Then in Step 2, the queries are sorted by the index of the first node in the search path, i.e. N(i). In Step 3 (see Figure 3), the source nodes are copied to the queries for which they are the first node in their search path. Finally, in Steps 4 and 5 (see Figures 4 and 5, respectively), the catalogs associated with the current vertices are selected and, for each query the successor record in the respective catalog is determined, respectively. Steps 3-5 can be performed in O(log N) time.

```
MoveVerticesToQueries:
(1)  Every PE(i): N'(i):=N(i); Dest(i) :=-1
(2)  Route([N'(i)], i-1, i>0)
(3)  PE(N): N'(N):=N(N) + 1
(4)  Every PE(i) with N'(i)≠N(i): Dest(i):=i
(5)  Route([Dest(i)], N(i), N'(i)≠N(i))
(6)  Every PE(i): v'(i):=v(i)
(7)  RouteAndCopy([v'(i)], Dest(i), Dest(i)≠-1)
```

Figure 3. Sketch of Procedure MoveVerticesToQueries.

```
SelectCatalogs:
(1)  Every PE(i): N'(i):=N(i); Dest(i) :=-1
(2)  Route([N'(i)], i-1, i>0)
(3)  PE(N): N'(N):=N(N) + 1
(4)  Every PE(i) with N'(i)≠N(i): Dest(i):=v'.EndCat(i)
(5)  RouteAndCopy([Dest(i)], Dest(i), Dest(i)≠-1)
```

Figure 4. Sketch of Procedure SelectCatalogs.

```
SearchCatalogs:
(1)  Every PE(i) with Dest(i)≠-1 : c'(i):=c(i)
(2)  Number(H(i), Dest(i)≠-1)
(3)  Concentrate([c'(i),Dest(i)], Dest(i)≠-1)
(4)  Reverse([c'(i), Dest(i)], N, H(i)+N-1)
(5)  BitonicMerge2([q'(i),c'(i),r'(i)],v'index(i),q'(i),c'index(i),c'(i),0,N-1,0,H(i)-1)
(6)  Every PE(i)  : flag'(i):=flag(i)
(7)  Route([flag'(i)], i+1, i<N)
(8)  RouteAndCopy([c'(i)],(i),flag'(i)≠flag(i) and flag(i)=c)
```

Figure 5. Sketch of Procedure SearchCatalogsForQueries.

Phase x ($2 \le x \le p$) of the Multi Iterative Search Algorithm

As indicated above, the purpose of each subsequent phase is to advance, in time O(log N), all queries by one step in their search paths. After Phase x-1 has been completed, all queries are sorted (in registers q(i)) with respect to the index of the $(x-1)^{th}$ node in their search path. Each processor PE(i) contains in its register v'(i) a copy of the $(x-1)^{th}$ node in the search path of the query stored in q(i). In register c'(i), PE(i) stores a copy of the successor catalog element of query q(i) in catalog $C_{v'(i)}$. The desired effect of Phase x is to have all queries sorted (in registers q(i)) with respect to the index of the x^{th} node in their search path, and have each processor PE(i) contain (in its register v'(i)) a copy of the x^{th} node in the search path of query q(i) and (in register c'(i)) a copy of the successor of q(i) in $C_{v'(i)}$.

An outline of the algorithm for Phase x is given in Figure 6. First (in Step 1), every PE(i) computes for the query currently stored in its register q(i) which edge to use for the next step in the search path as well as the index of the next node, and stores these two numbers in the auxiliary registers R(i) and N(i), respectively. Note that if the query has to be routed backward in the graph G then a negative value is stored in the register R(i). In Step 2, all queries are sorted by the index of the next node in their search paths. By sorting first the backward moving queries and then the forward moving queries, this sorting operation can use the properties of the previous permutation of the queries and be performed by a procedure OrderQueriesByNextVertex in time O(log N). Once this ordering has been obtained, the nodes can be matched with the queries, and the respective catalogs can be selected and searched, in time O(log N) in the same way as above.

```
Phase x , 2≤x≤p:
(1)  Every PE(i):   R(i):=g(v'(i),q(i),c'(i));
                    If R(i)>0
                        THEN N(i):=v'.successor_R(i)(i)
                        ELSE N(i):=v'.predecessor_-R(i)(i)
(2)  OrderQueriesByNextVertex
(3)  MoveVerticesToQueries
(4)  SelectCatalogs
(5)  SearchCatalogsForQueries
```

Figure 6. Overview of Phase x, 2≤x≤p.

What remains to be discussed is procedure OrderQueriesByNextVertex. This procedure, which is sketched in Figure 7, creates in time O(log N) the new ordering of the queries with respect to the indices of the next nodes in the search paths.

We first consider all forward edges to next vertices in the search paths; the backward edges are handled analogously. Let (v,w) and (v',w') be two such edges for two queries q and q', respectively, with the property that $g(v,q,\sigma(q,C_v))=g(v,q,\sigma(q,C_{v'}))$. From the monotonicity of G it follows that if Index(v)<Index(v') then Index(w)≤Index(w'). Therefore, the subsequence of queries q for which $g(v,q,\sigma(q,C_v))$ has the same value r is already sorted with respect to the index of the next vertex. Furthermore, since each node has an outdegree of at most μ_{out}, there are at most $\mu_{out}=O(1)$ such subsequences. The new ordering of the queries can therefore be created in time $O(\mu_{out} \log N)=O(\log N)$ by extracting these μ_{out} ordered subsequences and mergeing them in μ_{out} bitonic merge steps. For the backward edges the same idea applies because a monotone graph has the same monotonic properties forwards or backwards.

```
OrderQueriesByNextVertex:
(1)  Initialize all shift registers.
(2)  Every PE(i): Shift(i):=0
(3)  FOR r:= -μin,...,-1,1,...,μout DO
(4)       Every PE(i) with R(i)=r: q'(i):=q(i), N'(i):=N(i)
(5)       Concentrate([q'(i), N'(i)], N'(i)=r)
(6)       Number(LS(i), N(i)=r)
(7)       ls := LS(0)
(8)       shift := Shift(0)
(9)       Reverse([q'(i), N'(i)],0,ls)
(10)      Route([q'(i), N'(i)], i+Shift(i), N'(i)=r)
(11)      Every PE(i) with N'(i)=r: q''(i):=q'(i), N''(i):=N'(i)
(12)      BitonicMerge([q''(i), N''(i)],N''(i),0,shift,shift+ls)
(13)      Every PE(i): Shift(i):=Shift(i)+LS(i)
(14) Every PE(i): q(i):=q''(i), N(i):=N''(i)
```

Figure 7. Sketch of Procedure OrderQueriesByNextVertex.

Theorem 1. *For a monotone catalog graph of size N (and bounded degree), N iterative search queries along paths of length at most p can be executed independantly, in parallel, in time O((p + log N) log N) on a hypercube multiprocessor of size N.*

For the algorithm presented above, it is in fact not necessary that all catalogs are given a priori. It is easy to see that for each phase only those catalogs to be accessed by at least one query need to be available, provided that these catalogs are stored in sorted order with respect to the indices of the respective nodes.

3 Determination of all Intersections of n Lines with a Simple Polygonal Path of Length n

Consider the problem of determining, for a given simple polygonal path P of length n, all intersections with a given query line l. Chazelle and Guibas [CG2] have presented an O(n) space, O(n log n) preprocessing time, and $O(k \log \frac{n}{k})$ query time sequential solution (where k is the number of reported results) based on sequential fractional cascading. They introduce a convex hull decomposition tree T with a root node representing the convex hull of P; recursively, P is then split into two halves represented by each subtree. The convex hull at each

vertex is stored as follows: at every leaf, the respective line segment is stored; for every internal node the at most two merge lines for creating the convex hull for this node (from the convex hulls of the two direct children) are stored. Line queries are answered in a branch-and-bound fashion; the subtree H_l of T traversed by a query line l is shown to have at most $O(k \log \frac{n}{k})$ vertices. The decision to be made at each node of T consists mainly of locating the slope of l in a sorted list of slopes of convex hull edges, and hence fractional cascading can be applied.

We observe that for a line l, instead of traversing H_l in a branch-and-bound fashion as in [CG2], we can also traverse H_l in an inorder traversal with exactly the same type of decision to be made at every node. Such a traversal (as well as the tree T) meet the requirements for parallel fractional cascading on a hypercube described in Section 2.

It is easy to see that on a hypercube of size n, the convex hull decomposition tree T can be constructed in time $O(\log^2 n)$. Applying Theorem 1, we obtain

Theorem 2. *Given a simple polygonal path P of length n and a set of n arbitrary query lines, then for all query lines all intersections with P (with a maximum of k results per query) can be determined on a hypercube multiprocessor of size N in time $O((k \log \frac{n}{k} + \log n) \log n)$.*

Note: We assume that at the end of each phase of parallel fractional cascading, every processor can report a result without having to store it. Otherwise, after each phase the reported results need to be concentrated in order to obtain an even data distribution, and the number of processors and time complexity increase to O(n+M) and $O((k \log \frac{n}{k} + \log n) \log (n+M))$, respectively, where M denotes the total output size for all queries. A similar argument holds for Section 4.

4 Multi Slanted Range Search

Consider a set S of n points in the Euclidean plane and n slanted range queries as described in [CG2]. Chazelle and Guibas present a sequential fractional cascading algorithm that allows to answer one slanted range queries in time O(k + log n), where k is the size of the output, with O(n) and O(n log n) space and preprocessing, respectively. Their algorithm is based on a tree T representing S as follows: the root of T represents the lower hull of S; the left and right subtree represent the lower hulls of the left and right half, respectively, of the set S minus its lower hull points. A slanted range search query is again executed by a branch-and-bound type search on T, where the decision at each node reduces to a catalog look-up of the angle of a border segment of the range in the sorted list of angles of the lower hull edges for the that node.

We observe that if we replace this type of traversal by an inorder traversal of length $O(k \log \frac{n}{k})$ we meet the requirement of parallel fractional cascading on a hypercube multiprocessor described in Section 2. Since the convex hull of n points can be obviously computed on a hypercube of size n in time $O(\log^2 n)$ by using the standard divide and conquer approach together with bitonic merging, it follows that T can be computed in time $O(\log^3 n)$. Applying Theorem 1, we obtain

Theorem 3. *Given a set of n points in the Euclidean plane then, with a preprocessing of $O(\log^3 n)$, n slanted range search queries [CG2] (with a maximum of k results per query) can be solved on a hypercube of size N in time $O((k + \log N) \log N)$.*

5 A Hypercube Implementation of a Segment Tree for Next Element Search, and Applications

We will now use the results obtained in Section 2 to present an efficient parallel implementation, for the hypercube multiprocessor, of a well known data structure: the segment tree [BW]. The segment tree is a widely used structure which has for example been used to obtain efficient implementations of plane sweep algorithms in computational geometry [BW], [M], [PS]. Here, we consider an application of the segment tree to the next element search problem.

The *next element search* problem is a well known problem in computational geometry. Given a set S of n non intersecting line segments $I_1,...,I_n$ and a direction D_{next} (without loss of generality we will assume that D_{next} is the direction of the positive Y-axis), the next element search problem consists of finding for each point p_i of a set of m query points $p_1,...,p_m$ the line segment I_j first intersected by the ray starting at p_i in direction D_{next} (m=O(n)).

A well known method for solving the next element search problem is to apply a plane sweep in direction D_{next} using a *segment tree* [BW], [M], [PS]. Let $I_i^{(x)}$ [$p_i^{(x)}$] be the projection of line segment I_i [point p_i, respectively] onto the x-axis, and let $(x_1,x_2,...,x_{2n})$ be the sorted sequence of the projections of the 2n endpoints of $I_1,...,I_n$ onto the x-axis. The segment tree $T(S) = (V_S,E_S)$ for S is the complete binary tree with leaves $x_1,...,x_{2n}$. For every node v of T(S), an interval xrange(v) is defined as follows:

- if v is a leaf x_i, then xrange(v) = $[x_i,x_{i+1})$. ($[x_{2n},x_{2n+1})=[x_{2n},x_{2n}]$)
- if v is an internal node, then xrange(v) is the union of all intervals xrange(v') such that v' is a leaf of the subtree of T(S) rooted at v.

With every node v of a segment tree T(S) there is associated a node list NL(v) $\subseteq$ S which is defined as follows: NL(v) = { $I \in S$ | xrange(v) $\subseteq I^{(x)}$ and not (xrange(father of v) $\subseteq I^{(x)}$)}. For the remainder let h denote the height of T(S).

A segment tree T(S) is a monotone catalog graph with respect to the following index function: assume all nodes being sorted by height as major key and xrange as minor key, then the index of each node is its rank with respect to this ordering.

For every query point p, we define path(p) to be the path in T(S) from the root to the leaf v such that $p^{(x)} \in$ xrange(v). In order to solve the next element search problem, each query point p is routed along path(p). At every node v on the path, the successor of p in NL(v) is determined (this process will be referred to as *locating* p in NL(v)). Hence, the next element search problem for n query points reduces to n multiple look-up queries on T(S). We show next how to build T(S), in particular how to build the node lists (catalogs) NL(v).

Note that each line segment can occur in O(log n) node lists and, thus, the sum of the lengths of all node lists is O(n log n) [M]. Hence, storing the segment tree with all its node lists in a hypercube multiprocessor requires O(n log n) processors.

For a segment $I \in S$ with $I^{(x)} = [a,b]$ we define *l-path(I)* to be the path from the root of T(S) to the leaf v of T(S) with $a \in$ xrange(v). Likewise we define *r-path(I)* to be the path from the root of T(S) to the leaf v of T(S) with $b \in$ xrange(v). We observe that, if a line segement I is contained in a node list NL(v), then exactly one of the following four cases applies:

(1) $v \in$ l-path(I)
(2) v is the right child of a node $v' \in$ l-path(I)
(3) $v \in$ r-path(I)
(4) v is the left child of a node $v' \in$ r-path(I)

We define $NL_r(v)$, $r \in \{1,2,3,4\}$, to be the set of all $I \in$ NL(v) for which case r applies.

The algorithm for constructing the segment tree T(S) consists of four parts. In Part r, $1 \le r \le 4$, all line segment are routed through T(S). When they arrive at the nodes of height i, $1 \le i \le h$, the node lists $NL_r(v)$ of all those nodes are created. In order to efficiently determine, for a query point, the next line segment in a node list NL(v), the segments have to sorted with respect to the above-below relation within the vertical slab defined by xrange(v). We will create every sublist $NL_i(v)$ in sorted order; at the end of Part 4, the node lists NL(v) in sorted order are obtained from the sublists $NL_i(v)$ by applying bitonic merge.

We will show how to execute Parts 1 and 2; Parts 3 and 4 follow by symmetry. We assume a hypercube of size N=max{n,m} where initially every processor stores one line segment and one query point; w.l.o.g., $m=n=N=2^d$.

We first present Part 1 of the algorithm, i.e. how to create the node list $NL_1(v)$ for all nodes v with Level(v)=i.

This problem is solved by taking as the set of queries (for our parallel fractional cascading algorithm) the set of line segments, and route every segment $I \in S$ along l-path(I). At the end of Phase i ($1 \le i \le h$) for every node v with Level(v)=i there exists a block of consecutively numbered processors containing all line segments s such that $v \in$ l-path(s). From these, we can immediately extract all line segments $s \in NL_1(v)$.

What remains to be discusses is how to obtain a sorted ordering of the node lists $NL_1(v)$. We observe that our fractional cascading algorithm applied to a segment tree T(S) is stable in the following sense: if two queries (line segments) q_1 and q_2 are initially stored in processors $PE(j_1)$ and $PE(j_2)$ with $j_1<j_2$, and the i_{th} node in l-path(q_1) is the same as the i^{th} node in l-path(q_2), then at the end of Phase i the queries q_1 and q_2 are stored in two processors $PE(j_1')$ and $PE(j_2')$ with $j_1'< j_2'$. Therefore, we initially sort all line segments by the y-coordinates of their left endpoints. Then, at the end of each Phase i all line segments which were routed to a

node v are ordered by y-coordinate; i.e., $NL_1(v)$ is in the desired order. Applying the results from Section 2.2 we therefore obtain the following.

Lemma 1. *Part 1 of the segment tree construction algorithm can be executed in time $O(\log^2 N)$ on a hypercube of size N log N.*

We now turn to Part 2 of the algorithm; i.e., constructing the node lists $NL_2(v)$. As in Part 1, each line segement I in a node list $NL_2(v)$ still has the property that xrange(v) $\subseteq I^{(x)}$, but in contrast to the former case a total ordering of the line segements in $NL_2(v)$ can not be obtained by using the sorted order of the left (or right) endpoints of the segments.

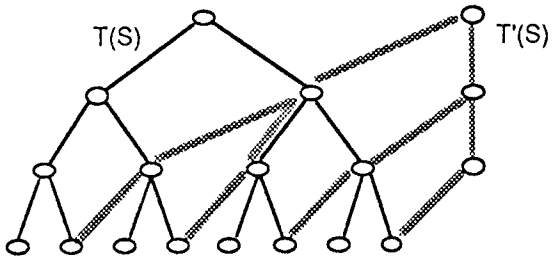

Figure 8. The Trees T(S) and T'(S).

Let T'(S) be the "shifted" version of T(S) as shown in Figure 8. For each line segement $I \in S$ let l-path'(I) be the path in T'(S) either from the last node v of l-path(I), if v is a right child in T(S), or otherwise from the right sibling of v in T(S), to the root of T'(S).

From the above definitions it follows that if a segment $I \in S$ is in a node list $NL_2(v)$ then v is a node in l-path'(I).

We also observe that if l-path'(I)=$(w_1, ..., w_h)$ and $I \notin NL_2(w_i)$, then $I \notin NL_2(w_j)$ for all $j \geq i$. Thus, for three nodes w_0, w_1, and w_2 in T'(S) such that $(w_1,w_0) \in E_S'$ and $(w_2,w_0) \in E_S'$ it follows that $NL_2(w_0) \subseteq NL_2(w_1) \cup NL_2(w_2)$. Let xrange($w_0$)=[a,b) and consider the ordering of $NL_2(w_0)$ obtained by sorting the line segments by the y-coordinate of their intersection with the line x=a. This ordering can be constructed from the analogous orderings of $NL_2(w_1)$ and $NL_2(w_2)$ by eliminating from these sequences the elements not contained in $NL_2(w_0)$ and merging the so obtained subsequences.

The idea for Part 2 of the algorithm is to route all line segements along l-path'(I). Since T'(S) is a monotone (catalog) graph with O(n) sources, this can be implemented in time $O(\log^2 N)$. It is easy to see that during this search, it is possible to delete some line segments (i.e., eliminate them from further consideration) in any phase of the parallel fractional cascading algorithm without changing the time complexity. In this particular case, we delete a line segment $I \in S$ if it has been routed to some node v with $I \notin NL_2(v)$.

At the end of Phase i, $1 \leq i \leq h$, for each node w_0 in G with Level(w_0)=h-i+1 there exists a consecutive sequence of processors containing all line segments $I \in NL_2(w_0)$. In Phase i-1, these line segments have been routed to at most two different nodes w_1 and w_2. If $NL_2(w_1)$ and $NL_2(w_2)$ where previously ordered as described above, then the same ordering for $NL_2(w_0)$ can be obtained by extracting the two subsequences of segments previously routed to $NL_2(w_1)$ and $NL_2(w_2)$, respectively, and merging these subsequences using a bitonic merge. Since only two line segements where initially routed to every source of T'(S), the orderings of all lists $NL_2(v)$ can be maintained through all phases with an overhead of O(log N) steps per phase.

This yields

Lemma 2. *Part 2 of the segment tree construction algorithm can be executed in time $O(\log^2 N)$ on a hypercube of size N log N.*

Summarizing, we obtain

Theorem 4. *The segment tree construction problem can be solved on a hypercube of size N log N in time $O(\log^2 N)$; N=max{m,n}.*

Our strategy for solving the next element search problem has five phases. In phase one, we construct the segment tree T(S) as described above. In phase two, we route the query points down T(S), constructing a query list QL(v) for each node v by storing copies of all queries that visit it. The query lists are stored in the same manner as the catalogs (see Section 2.2). Phase

three converts in $O(\log^2 N)$ time the catalogs NL(v) into a forest of binary trees, one for each catalog. Each NL(v), which is sorted as described above, is represented by a balanced binary search tree. This step can be completed by having each catalog element calculate its level and position in its tree. A single $O(\log^2 N)$ sort can then be used to construct the forest of binary trees. Each Catalog lookup can now, in part four, be performed by executing a multiple binary search in which each query in each query list is routed through the respective binary search tree. Finally, in phase five, the partial results generated in phase four, are sorted by the original query points, and the final results are calculated by choosing the best answer for each query point.

Summarizing, we obtain

Theorem 5. *The next element search problem for a set of n disjoint line segments and m query points can be solved on a hypercube of size N log N in time $O(\log^2 N)$; N=max{m,n}.*

Theorem 5 implies an efficient hypercube solution for another fundamental geometric problem: the construction of the *trapezoidal map* [TW].

Given a set S of n disjoint line segments in the plane; for any endpoint p of a segment in S, the trapezoidal segments for p are the (at most two) line segments first intersected by the rays emanating from p in direction of the positive and negative y-axis, respectively. The construction of the trapezoidal map consists of finding for each endpoint of the segments in S its trapezoidal segments.

This problem is fundamental in computational geometry and is frequently used to solve other geometric problems; see e.g. [G], [TW], and [Y]. Atallah, Cole, and Goodrich [ACG], [G] presented an O(log n) time algorithm for computing the trapezoidal decomposition on a PRAM with O(n) processors (and O(n log n) space). As a consequence of Theorem 5, we obtain

Corollary 1. *For a set of n disjoint line segments, the trapezoidal map can be computed on a hypercube with n log n processors in time $O(\log^2 n)$.*

Yap [Y] has shown that on a PRAM with O(n) processors (and O(n log n) space), the *triangulation* of a simple polygon (see [TW]) can be computed in time O(log n) by essentially applying two calls of the trapezoidal map algorithm (of [ACG], [G]). By combining the result in [Y] with Corollary 1, we obtain

Corollary 2. *An n-vertex simple polygon can be triangulated on a hypercube multiprocessor of size n log n in time $O(\log^2 n)$.*

REFERENCES

[ACG] M.J. Atallah, R. Cole, and M.T. Goodrich, "Cascading divide-and-conquer: a technique for designing parallel algorithms", SIAM J. Comput. 18:3, pp. 499-532, 1989.

[ACGD] A. Aggarwal, B. Chazelle, L. Guibas, C. O'Dunlaing, and C. Yap, "Parallel computational geometry", Algorithmica 3:3, 1988, pp. 293-327.

[AG] M.J. Atallah, M.T. Goodrich, "Efficient plane sweeping in parallel", in Proc. ACM Symp. Computational Geometry, 1986, pp. 216-225.

[B] K.E. Batcher, "Sorting networks and their applications", in Proc. AFIPS Spring Joint Computer Conference, 1968, pp. 307-314.

[BW] J.L. Bentley and D. Wood, "An optimal worst case algorithm for reporting intersections of rectangles", IEEE Transactions on Computers 29:7, 1980, pp. 571-576.

[CG1] B. Chazelle, L.J. Guibas, "Fractional cascading: I. A data structuring technique", Algorithmica 1:2, 1986, pp. 133-162.

[CG2] B. Chazelle, L.J. Guibas, "Fractional cascading: II. Applications", Algorithmica 1:2, 1986, pp. 163-192.

[DK] N. Dadoun and D.G. Kirkpatrick, "Parallel processing for efficient subdivision search", in Proc. ACM Symp. on Computational Geometry, 1987, pp. 205-214.

[DR] F. Dehne, A. Rau-Chaplin, "Implementing data structures on a hypercube multiprocessor, and applications in parallel computational geometry", Tech. Rep. SCS-TR-152, School of Computer Science, Carleton Univ., Ottawa, Canada K1S5B6.

[G] M.T. Goodrich, "Efficient parallel techniques for computational geometry", Ph.D. thesis, Department of Computer Science, Purdue University, 1987.

[M] K. Mehlhorn, "Data structures and algorithms 3: multi-dimensional searching and computational geometry", Springer Verlag, 1984.

[NS] D. Nassimi, S. Sahni, "Data broadcasting in SIMD computers", IEEE Trans. on Computers 30:2, 1981, pp. 101-106.

[PS] F.P. Preparata and M.I. Shamos, "Computational geometry - an introduction", Springer Verlag, 1985.

[TW] R.E. Tarjan and C.J. Van Wyk, "An O(n log log n) time algorithm for triangulating a simple polygon", SIAM Journal of Computing 17, 1988, 143-178.

[Y] C.-K. Yap, "Parallel triangulation of a polygon in two calls to the trapezoidal map", Algorithmica 3:2, 1988, pp. 279-288.

APPENDIX A: BASIC HYPERCUBE OPERATIONS

The following is a list of slightly generalized versions of well known hypercube operations, as used in Section 2. (In addition to those registers listed below, the actual implementation of these operations may require a constant number of auxiliary registers.)

Rank(Reg(i),Cond(i)): Compute, in time O(log N), in register Reg(i) of every processor PE(i) the number of processors PE(j) such that $j < i$ and Cond(j) is true [NS].

Number(Reg(i),Cond(i)): Compute, in time O(log N), in register Reg(i) of every processor PE(i) the number of processors PE(j) such that Cond(j) is true.

Concentrate([Reg_1(i),...,Reg_z(i)],Cond(i)): This operation includes an initial Rank(R(i), Cond(i)) operation. Then for each PE(i) with Cond(i) = true, registers Reg_1(i),...,Reg_z(i) are copied to PE(R(i)), z=O(1). The time complexity of this operation is also O(log N) [NS].

Route([Reg_1(i),...,Reg_z(i)],Dest(i),Cond(i)): Every processor PE(i) has z=O(1) data registers Reg_1(i),...,Reg_z(i), a destination register Dest(i), and a boolean condition register Cond(i). It is assumed that the destinations Dest(i) are monotonic; i.e., if $i<j$ then Dest(i)<Dest(j). This operation routes, for every processor PE(i) with Cond(i) = true, all registers Reg_1(i),...,Reg_z(i) to processor PE(Dest(i)); it can be implemented with an O(log N) time complexity by using a Concentrate operation followed by a Distribute operation described in [NS].

RouteAndCopy([Reg_1(i),...,Reg_z(i)],Dest(i),Cond(i)): Under the same assumptions as for the Route operation, this operation routes, for every processor PE(i) with Cond(i) = true, a copy of registers Reg_1(i),...,Reg_t(i) to processors PE(Dest(i - 1) + 1), ..., PE(Dest(i)), each; it can be implemented with an O(log(N)) time complexity by using a Concentrate followed by a Generalize operation described in [NS].

Reverse([Reg_1(i),...,Reg_z(i)],Start,End): This operation routes for every PE(i) with Start $\leq i \leq$ End, its registers Reg_1(i), ..., Reg_z(i), z=O(1), to PE(Start + End - i); i.e., it reverses the contents of those registers for the sequence of processors between PE(Start) and PE(End). Reversing, in the entire hypercube, a sequence of n values (each stored in one processor) corresponds to routing each value stored at processor PE(i) to processor PE(i'), where i' is obtained from i by inverting all bits in its binary representation. Hence, this operation can be implemented in time log(n) similarly to the Concentrate/Distribute operation described in [NS].

BitonicMerge([Reg_1(i),...,Reg_z(i)],Key(i),Left,Peak,Right): This operation is the well known bitonic merge [B]. It converts in time O(log N) a bitonic sequence (with respect to register Key(i)) into a sorted sequence; it simultaneously permutes the registers Reg_1(i),...,Reg_t(i) (z=O(1)). Here, we apply it to a particular bitonic sequence consisting of an increasing sequence starting at PE(Left) and ending at PE(Peak) followed by a decreasing sequence starting at PE(Peak+1) and ending at PE(Right).

BitonicMerge2([Reg_1(i),Reg_2(i),Reg_3(i)],Key_{11}(i),Key_{12}(i),Key_{21}(i),Key_{22}(i),$Left_1$,$Right_1$,$Left_2$,$Right_2$): This operation simulates a $2N$ hypercube. It merges the sorted sequences stored in registers Reg_1 and Reg_2, and stores the resulting sequence into register Reg_3(1), ..., Reg_3(($Right_1$-$Left_1$ + 1) + ($Right_2$-$Left_2$ + 1)). The principal keys for merging are Key_{11}(i) and Key_{21}(i), while the secondary keys are Key_{12}(i) and Key_{22}(i). The first sequence is stored in processors PE($Left_1$) to PE($Right_1$) while the second sequence is stored in PE($Left_2$) to PE($Right_2$).

Sort([Reg_1(i),...,Reg_z(i)],Key(i)): This operation refers to $O(\log^2 n)$ time bitonic sort [B] with respect to Key(i); it simultaneously permutes the registers Reg_1(i),...,Reg_z(i) (z=O(1)).