IMPLEMENTING DATA STRUCTURES ON A HYPERCUBE MULTIPROCESSOR, AND APPLICATIONS IN PARALLEL COMPUTATIONAL GEOMETRY

FRANK DEHNE* AND ANDREW RAU-CHAPLIN

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

Abstract. In this paper, we study the problem of implementing standard data structures on a hypercube multiprocessor. We present a technique for efficiently executing multiple independant search processes on a class of graphs called ordered h-level graphs. We show how this technique can be utilized to implement a segment tree on a hypercube, thereby obtainig $O(\log^2 n)$ time algorithms for solving the next element search problem, the trapezoidal decomposition problem, the triangulation problem, and the (multiple) planar point location problem.

1 INTRODUCTION

One of the main differences, besides the difference in communication delay, between the parallel random access machine (CREW-PRAM) and the hypercube processor network is that the PRAM has one large (shared) memory similar to a standard sequential computer, whereas the hypercube has its memory divided into pieces of constant size and distributed over the network.

The fact that the PRAM memory resembles the structure of the standard sequential machine memory has been extensively used for the design of efficient PRAM algorithms. It allows the implemention, on a PRAM, of well established data structures like, e.g., segment trees [ACG], [ACGD], [G] or subdivision hierarchies [DK]. Once such a data structure has been built, each processor can search in it, independently of the others, in the standard manner.

In this paper, we show that a similar paradigm can also be used for a hypercube multiprocessor of size N; i.e., a set of N 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 define a class of graphs called *ordered h-level graphs* which includes most of the standard data structures (in particular, all k-nary search trees for $k=O(1)$) and show that for such a graph with n nodes stored on a hypercube multiprocessor, O(n) search processes can be efficiently executed independantly and in parallel.

We apply this method to implement a segment tree [BW], [M], [PS] for next element search on a hypercube. Since the total length of all lists attached to the nodes of a segment tree (for n segments) is $O(n \log n)$, we can not construct the entire segment tree before starting the search processes (as in [ACG], [ACGD], [G]), since this would require a hypercube of size $n \log n$. Instead, we first build the segment tree without the node lists, and then show how during the execution of the search processes these lists can be dynamically created only for those nodes currently visited by the search queries (thus, not exceeding a total length of O(n)).

Our approach provides $O(\log^2 n)$ time hypercube algorithms for the next element search problem, the trapezoidal map construction problem, the triangulation problem, and the planar (multiple) point location problem.

The paper is organized as follows: In Section 2, we define ordered h-level graphs and the associated m-way search problem, and also review some standard data movement operations. In Section 3, we present an efficient hypercube algorithm for m-way search on ordered h-level graphs and, in Section

* Research partially supported by the Natural Sciences and Engineering Research Council of Canada under Grant A9173.

F. Dehne and A. Rau-Chaplin, "Implementing data structures on a hypercube multiprocessor and applications in parallel computational geometry," in Proc. *International Workshop on Graphtheoretic Concepts in Computer Science (WG'89)*, Aachen (W.-Germany), 1989, M. Nagl (Ed.), Springer Verlag, Lecture Notes in Computer Science, Vol. 411, pp. 316-329.

4, we show how to use this method to implement a segment tree for next element search on a hypercube. Finally, in Section 5, we describe how this algorithm can be used to obtain efficient hypercube algorithms for the other geometric problems listed above.

2 DEFINITIONS AND BASIC HYPERCUBE OPERATIONS

In this section, we will first define ordered h-level graphs and the associated m-way search problem. Then, some basic standard hypercube data movement operations are reviewed, as these operations will be used in the remainder of this paper.

2.1 ORDERED H-LEVEL GRAPHS

Assume we are given a directed graph G=(V,E) with vertex set V and edge set E. An *ordered h-partitioning* of G is a partitioning of V into an ordered sequence of h disjoint sets $L_1,...,L_h$ together with an ordering of the elements in each subset L_i $(1 \le i \le h)$.

For every ordered h-partitioning of G we define, for every $v \in V$, three numbers *Level(v)*, *Levelindex(v)* and *Index(v)* as follows:

- Level(v) = i if and only if $v \in L_i$,
- Levelindex(v) is the rank of v with respect to the ordering of the vertices in $L_{Level(v)}$, and
- $Index(v) = \left(\sum_{1 \le i \le Level(v)-1} |L_i| \right) + Levelindex(v)$

A directed acyclic graph G = (V,E) is called an *ordered h-level graph* if it has the following properties (see Figure 1 for an illustration):

(1) There exists a constant k = O(1) such that every node of G has an out-degree of at most k.

(2) There exists an ordered h-partitioning $L_1,...,L_h$ of G such that
   (a) every source of G is contained in L_1,
   (b) if (v,w) is an edge of G then Level(w) = Level(v) + 1, and
   (c) if (v,w), (v',w') are two edges of G with Index(v) < Index(v'), then Index(w) ≤ Index(w').

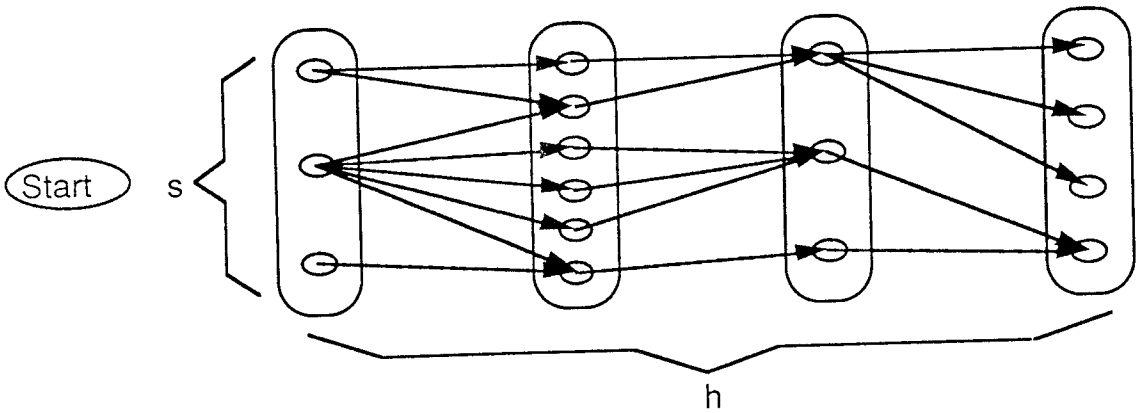

Figure 1. An Ordered h-Level Graph.

We observe that ordered h-level graphs are acyclic and planar, and that any k-nary tree (k=O(1)) is an ordered h-level graph.

2.2 THE M-WAY SEARCH PROBLEM FOR ORDERED H-LEVEL GRAPHS

Let $G = (V=L_1 \cup ... \cup L_h, E)$ be an ordered h-level graph (with maximum out-degree k), and let U is a universe of possible search queries on G.

A *search path* for a query $q \in U$ is a sequence $path(q)=(v_1, ..., v_h)$ of h vertices of G defined by a *successor* function $f: (V \cup \{start\}) \times U \Rightarrow \mathbf{N}$ (i.e., a function with the property that $f(start,q) \in L_1$ and for every vertex $v \in V$, $(v,f(v,q)) \in E$) as follows:

- $Index(v_1) = f(start, q)$
- $Index(v_{i+1}) = f(v_i, q)$, $1 \le i < h$.

We also define an associated *successor rank* function $g: (V \cup \{start\}) \times U \Rightarrow \{1,...,k\}$ as follows:

- $g(start, q)$ is the rank of $f(start, q)$ in the set $\{Index(v) | v \in L_1\}$, and
- $g(v_i, q)$ is the rank of $f(v_i, q)$ in the set $\{Index(w) | (v_i,w) \in E\}$, $1 \le i < k$.

For example, if G is a binary search tree then, for every query q, f(start,q) is the root of the tree; for every node v, $g(v,q) \in \{1,2\}$ indicates whether the left or right child is to be visited next and f(v,q) is the index of (or pointer to) that child.

Given an ordered h-level graph G with n nodes stored in a hypercube multiprocessor such that the node v with Index(v)=i is stored in processor PE(i), then a *search process* for a query q with search path $(v_1, ..., v_h)$ is a process divided into h time steps $t_1 < t_2 < ... < t_h$ such that at time t_i, $1 \le i \le h$, there exists a processor which contains a description of both, the query q and the node v_i. Note, however, that we do not assume that the search path is given in advance; we assume that it is constructed during the search by successive applications of the functions f and g.

Given an ordered h-level graph G with n nodes and a set $Q = \{q_1,...,q_m\} \subseteq U$ of m queries, m=O(n), then the *m-way search problem* consists of executing (in parallel) all m search processes induced by the m queries.

2.3 BASIC HYPERCUBE OPERATIONS

The m-way search algorithm described in the next section uses slightly generalized versions of eight well-defined hypercube data movement operations; in addition to those registers listed below, their implemenation requires a constant number of auxiliary registers. In the following, for every register *A* available at every processor, *A(i)* refers to register A at processor PE(i).

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 $\le$ i $\le$ 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).

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)).

3 AN O(MIN{S LOG N, LOG^{2}N} + H LOG N) TIME HYPERCUBE ALGORITHM FOR M-WAY SEARCH ON ORDERED H-LEVEL GRAPHS

Let G = (V =L$_1$,...,L$_h$, E) be an ordered h-level graph (with h-partitioning V =L$_1$,...,L$_h$), where |V| = n, and such that every node has an out-degree of at most k=O(1) and can be stored using O(1) space. For the remainder, *s* denotes the number of sources of G (i.e., the number of v$\in$ V with Level(v)=1).

Furthermore, let U be a universe of search queries, each of which can also be stored using O(1) space, and let f: (V $\cup$ {start}) x U $\Rightarrow$ **N** and g: (V $\cup$ {start}) x U $\Rightarrow$ {1,...,k} be the successor function and successor rank function, respectively, describing the search path in G associated with every search query. We assume that f(x,q) and g(x,q) can be computed in constant time for any x$\in$ V $\cup$ {start}, q$\in$ U.

In this section, we consider the problem of solving, on a hypercube multiprocessor, the m-way search problem for a set Q={q$_1$,...,q$_m$} $\subseteq$ U of m queries. We present an O(min{s log N, log^{2}N} + h log N) time algorithm for solving the m-way search problem on a hypercube of size N, where N=max{n,m}.

For the remainder we assume, w.l.o.g., that n=m=N=2^d; all results obtained can be easily generalized. In the following Section 3.1 we give an overview of the algorithm, including the assumed initial configuration of the hypercube, and how the result, i.e. the m-way search, is reported. In Sections 3.2 and 3.3 we will then present the details of the algorithm. Section 3.4 summarizes the results.

3.1 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); register v(i) contains fields v.data(i), v.Level(i), v.Levelindex(i), and v.Index(i), storing a constant amount of data associated with vertex v, its level, levelindex and index, respectively. The edges of G are stored as adjacency lists. That is, for every vertex v stored in register v(i), the indices of the at most k successors of v (i.e.; the indices of the vertices w such that (v,w)$\in$ E) are stored in the fields v.successor1(i), ..., v.successork(i), respectively; see Figure 2.

The set Q={q$_1$,...,q$_m$} of m queries is stored in arbitrary order, and such that every processor PE(i) stores one query in its register q(i).

Figure 2 shows the set of registers necessary at every processor PE(i). In addition to the registers v(i) and q(i) mentioned above, the algorithm assumes that every processor also has a register v'(i) to store another vertex of G as well as other auxiliary registers which will be described later.

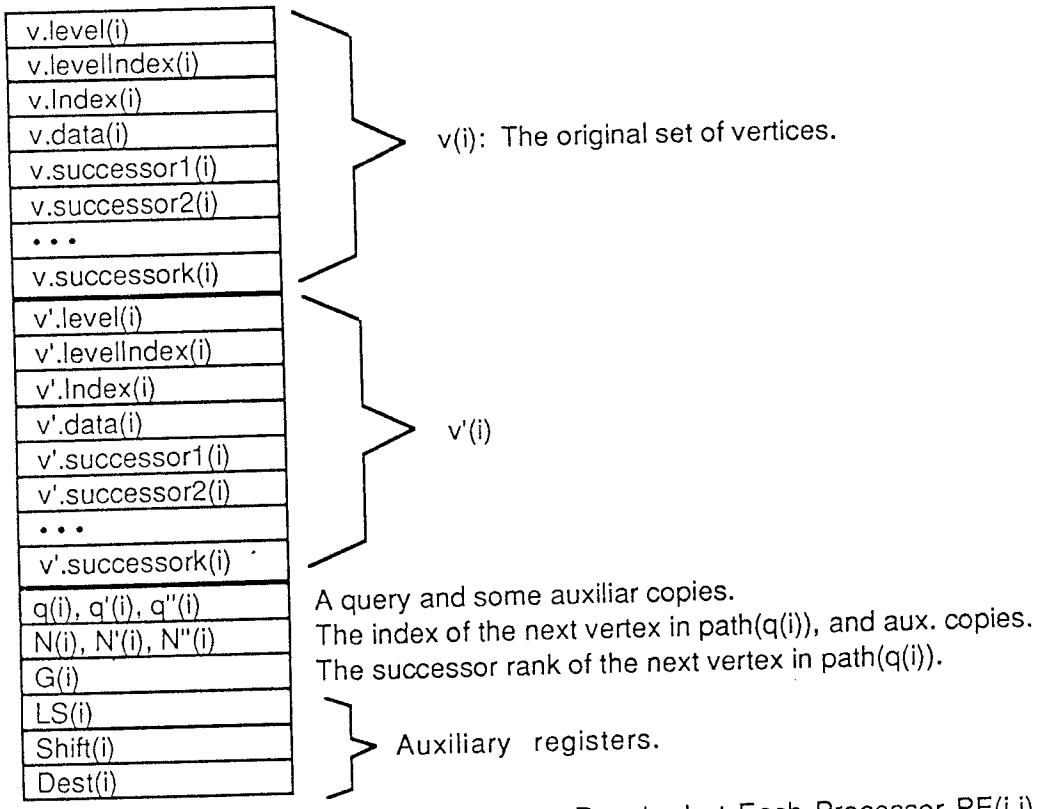

Figure 2. The Registers Required at Each Processor PE(i,j).

The global structure of the m-way search algorithm is described in Figure 3. The m search processes for all m queries q_1, ..., q_m are executed in h phases; each phase moves all queries one step ahead in their search paths.

```
Procedure M-Way-Search:
(1)  Phase1 {Match every query with the 1st node in its search path.}
(2)  For x := 2 to h do
(3)        Phasex {Match every query with the xth node in its search path.}
```

Figure 3: Global Structure of the M-Way-Search Algorithm

The algorithm permutes the queries (in registers q(i)) and copies some nodes into the registers v'(i) such that at the end of phase x ($1 \le x \le h$):

- all queries are sorted with respect to the index of the x^{th} node in their search path, and
- each processor PE(i) containing a query q in its register q(i), contains in its register v'(i) a copy of the x^{th} node in the search path of q (this is called a *match* of q and the x^{th} node in its search path).

A typical situation at the end of a phase is depicted in the following Figure 4; each vertical column represents the registers q(i) and v'(i) of a processor PE(i).

	PE(0)						...									PE(15)
q(i)	q5	q7	q15	q1	q4	q6	q8	q9	q12	q14	q16	q2	q3	q10	q11	q13
v'.Index(i)	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3
v'.Data(i)	v1	v1	v1	v2	v2	v2	v2	v2	v2	v2	v2	v3	v3	v3	v3	v3

Figure 4. A Typical Situation at the End of a Phase.

In the following Sections 3.2 and 3.3, we will describe the details of Phase 1 and Phase x ($2 \le x \le h$), 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).

3.2 PHASE 1 OF THE M-WAY SEARCH ALGORITHM

An outline of Phase 1 is given in Figure 5. The algorithm consists of four basic steps. 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 an auxiliary register N(i). Then, in Step 2, the number of sources is calculated (here represented by a variable s). In Step 3, the queries are sorted by the index of the first node in the search path, i.e. N(i); this ordering is performed in one of two possible ways depending on the number of sources. If $s \ge \log(N)$ then bitonic sorting [B] is used; if $s < \log(N)$, a procedure called SortBySourceIndex is used which sorts the queries in O(s log N) time and will be described below. Note that in many application s is a constant (e.g. for search trees) and, thus, the sorting step is performed in time O(log N).

Finally, in Step 4, the source nodes are copied to the queries for which they are the first node in their search path. Because of the ordering of the queries, this step can be performed in O(log(N)) time using a procedure MoveVerticesToQueries which will also be described below.

```
Procedure Phase1:
(1)  Every PE(i): N(i):=f(Start,q(i))
(2)  Number(LS(i), v.Level(i)=1)
     s:=LS(0)                          {Note: LS(i)=LS(i') for all i,i'}
(3)  IF s≥log(N) THEN
          Sort([q(i), N(i)], N(i))
     ELSE
          SortBySourceIndex(s)
(4)  MoveVerticesToQueries(1)
```

Figure 5. Outline of Phase 1.

We first discuss the details of Procedure SortBySourceIndex; see Figure 6a. The procedure uses a register Shift(i) at each processor which stores the number of queries that have already been sorted. In Step 1, all registers Shift(i) are initialized to 0. Then, Steps 3 to 8 are executed for each source of the graph. In each iteration, the queries q(i) that need to be matched with that source [as well as the associated source indices N(i)] are copied into registers q'(i) [N'(i)] of the same processors, concentrated (Steps 3 and 4), and then appended at the end of the sequence of queries ordered so far (Steps 5 and 6); finally the registers Shift(i) are updated (Steps 7 and 8). These steps produce, in the registers q"(i), the correct permutation of the queries, which are then copied back into the registers q(i) [N(i)]; see Step 9. Obviously, each iteration takes O(log N) time and, hence, the time complexity of procedure SortBySourceIndex(s) is O(s log N).

```
Procedure SortBySourceIndex(s):
(1)  Every PE(i): Shift(i):=0
(2)  FOR r:=1 TO s DO
(3)       Every PE(i) with N(i)=r: q'(i):=q(i), N'(i):=N(i)
(4)       Concentrate([q'(i), N'(i)], N'(i)=r)
(5)       Route([q'(i), N'(i)], i+Shift(i), N'(i)=r)
(6)       Every PE(i) with N'(i)=r: q"(i):=q'(i), N"(i):=N'(i)
(7)       Number(H(i), N(i)=r)
(8)       Every PE(i): Shift(i):=Shift(i)+H(i)
(9)  Every PE(i): q(i):=q"(i), N(i):=N"(i)
```

Figure 6a. Detailed Description of Procedure SortBySourceSelected.

```
Procedure MoveVerticesToQueries(CurrentLevel):
(1)  Every PE(i): N'(i):=N(i)
(2)  Route([N'(i)], i-1, i>0)
(3)  PE(1): 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), v.Level(i)=CurrentLevel)
```

Figure 6b. Detailed Description of Procedure MoveVerticesToQueries.

Once the queries have been sorted by the index of the first vertex in their search path, the matching process between each query and the first node in its search path can be performed in time O(log N) using the procedure MoveVerticesToQueries described in Figure 6b. The parameter CurrentLevel (which is one for all sources) denotes the level of the nodes to which the queries are to be routed (i.e., to be matched with). The idea is to identify for each node, the largest address of a query to be matched with that node (Steps 1 to 5), and then use the procedure RouteAndCopy to broadcast each node to the block of queries to be matched with (Steps 6 and 7). The time complexity of this process is O(log N).

3.3 PHASE x (2≤x≤h) OF THE M-WAY SEARCH ALGORITHM

As indicated in Section 3.1, the purpose of each subsequent phase is to advance, in time O(log N) all queries by one step in their search path. After Phase x-1 has been completed, all queries are sorted with respect to the index of the $(x-1)^{th}$ node in their search path, and each processor PE(i) contains a query q in its register q(i) together with a copy of the $(x-1)^{th}$ node in the search path of q in its register v'(i). The desired effect of Phase x is to have all queries sorted 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 the query q(i).

```
Procedure Phase(x), 2≤x≤h:
(1)  Every PE(i): N(i):=f(v'(i),q(i)), G(i):=g(v'(i),q(i))
(2)  OrderQueriesByNextVertex
(3)  MoveVerticesToQueries(x)
```

Figure 7. Overview of Phase x, 2≤x≤h.

An outline of the algorithm for Phase x is given in Figure 7. First (in Step 1), every PE(i) computes for the query currently stored in its register q(i) the index of the next node in its search path as well as the successor rank of that node (see Section 2.2) and stores these two numbers in the auxiliary registers N(i) and G(i), respectively. In Step 2, all queries are sorted by the index of the next node in their search paths. This sorting operation is performed by a procedure OrderQueriesByNextVertex in time O(log N) by using the properties of the previous permutation of the queries. Once this ordering has been obtained, the nodes can be matched with the queries in time O(log N) in the same way as described in Section 3.2.

What remains to be discussed are the details of procedure OrderQueriesByNextVertex. This procedure, which is described in Figure 8, 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.

Consider all edges (v,w) and (v',w') where Level(v)=Level(v')=x-1 and w and w' have the same successor rank. If Index(v) < Index(v') then Index(w) ≤ Index(w'). Therefore, the subsequence of queries for which the successor rank of the next vertex in their search path has the same value r is already sorted with respect to the index of the next vertex. Furthermore notice that, since each node has an outdegree of at most k, there are at most k=O(1) such subsequences. The idea for creating, in time O(k log N)=O(log N), the new ordering of the queries is therefore to extract these k ordered subsequences and merge them in k bitonic merge steps. The details are shown in Figure 8: for each of the k possible successor ranks, the respective subsequence of queries is extracted (Steps 4 and 5), inverted (Step 9), appended to the sequence of queries already ordered (Steps 10 and 11), and finally the so created bitonic sequence is converted into a sorted sequence (Step 12).

```
Procedure OrderQueriesByNextVertex:
(1)  Initialize all shift registers.
(2)  Every PE(i): Shift(i):=0
(3)  FOR r:=1 TO k DO
(4)        Every PE(i) with G(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 8. Details of Procedure OrderQueriesByNextVertex.

3.4 SUMMARY

We obtain the following

Theorem 1. *The m-way search problem for an ordered h-level graph with n nodes and s sources can be solved on a hypercube multiprocessor of size N, N=max{n,m}, in time O(min{s log N, log^{2}N} + h log N).*

The algorithm presented in Sections 3.1 to 3.3 has the additional property that it consists of h phases such that at the end of phase x ($1 \le x \le h$) all queries are sorted with respect to the index of the x^{th} node in their search path, and each processor PE(i) contains in its register v'(i) a copy of the x^{th} node in the search path of the query currently stored in its register q(i).

4 A HYPERCUBE IMPLEMENTATION OF A SEGMENT TREE FOR NEXT ELEMENT SEARCH

We will now apply the results obtained in Section 3 and 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.

In the following, we will first review the definition of the next element search problem as well as the definition and some basic properties of segment trees. We will then show how to implement a segement tree on a hypercube multiprocessor, using the parallel m-way search algorithm for ordered h-level graphs, and obtain an efficient solution for 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 $l_1,...,l_n$ 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 l_j first intersected by the ray starting at p_i in direction D_{next} (m=O(n)), as illustrated in Figure 9.

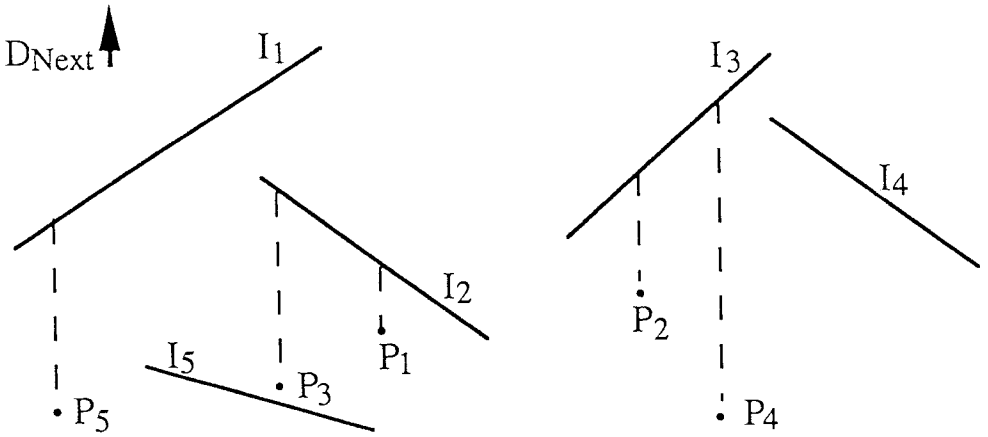

Figure 9. The Next Element Search Problem.

An obvious 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 $l_i^{(x)}$ [$p_i^{(x)}$] be the projection of line segment l_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 $l_1,...,l_n$ 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) = { $l \in S$ | xrange(v) $\subseteq l^{(x)}$ and not (xrange(father of v) $\subseteq l^{(x)}$)}.

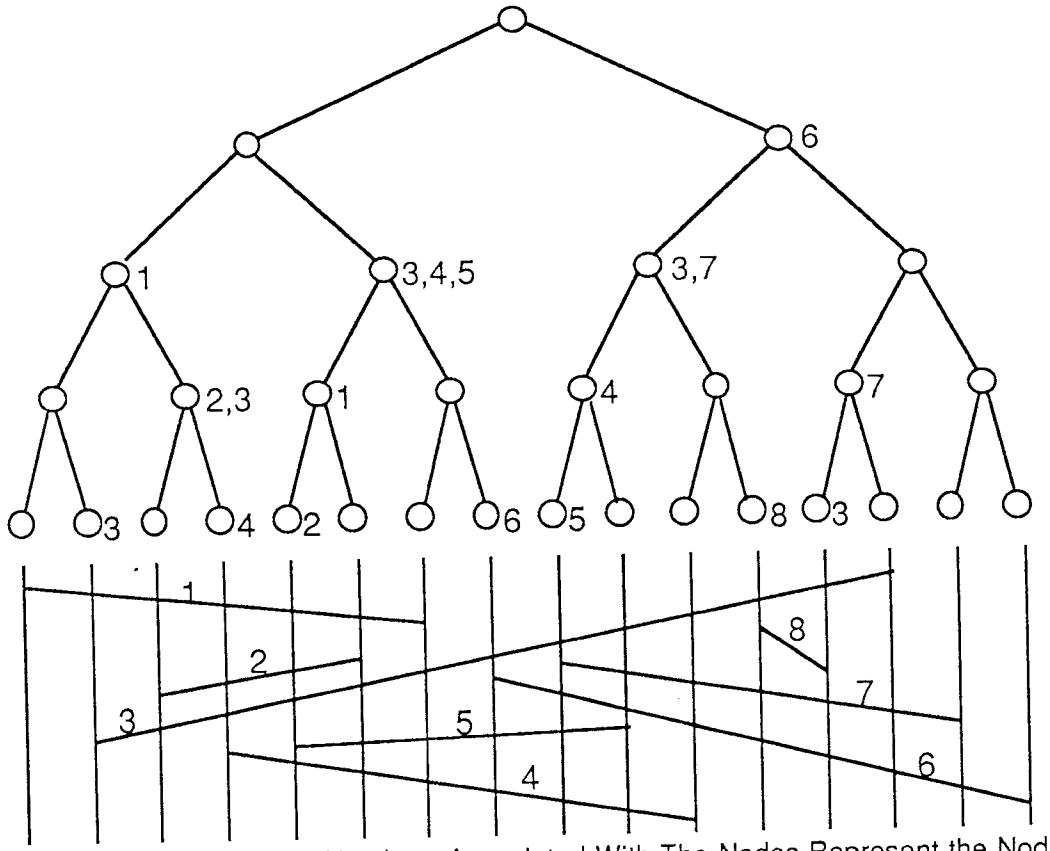

Figure 10. A Segment Tree (The Numbers Associated With The Nodes Represent the Node Lists).

A segement tree T(S) is an ordered h-level graph where h is the height of T(S). For any node v, Level(v) is the height of v in T(S), Levelindex(v) is the rank of v in {v'|Level(v')=Level(v)} with respect to the ordering of these nodes by increasing x-coordinate of xrange(v'), and Index(v) is defined by the above as described in Section 2.1.

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 next element of p in NL(v) is determined (this process will be referred to as *locating* p in NL(v)). For each query point, the final result to be reported is obtained by maintaining, throughout the process, the closest next element found so far. The routing of all query points along their paths can be implemented in time $O(\log^2 N)$ using the m-way search algorithm of Section 3. What remains to be described is how to build the node lists NL(v) and how to locate the query points in the node lists on their paths.

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 would require O(n log n) processors.

Fortunately, the sum of the lengths of the node lists of all nodes with the same level (height) is O(n). Hence, the strategy for solving the next element search problem will be to first construct the segment tree without its node lists (which is trivial) and, while the query points are routed through the tree only the node lists at the level currently reached by the query points are constructed.

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)

during this search, it is possible to delete some line segments (i.e., eliminate them from further consideration) in any phase of the m-way search algorithm without changing the time complexity. In this particular case, we delete a line segment $l \in S$ if it has been routed to some node v with $l \notin NL_2(v)$.

At the end of Phase i, $1 \le i \le h$, for each node w_0 in G with $Level(w_0)=h-i+1$ there exists a consecutive sequence of processors containing all query points p such that w_0 is the i^{th} node in path'(p) and all line segments $l \in NL_2(w_0)$. 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 (could be ordered in constant time) 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.

Similarly, every query point p is routed along its path'(p), and such that at every level of T(S), the queries routed to each node of that level are ordered with respect to their y-coordinate.

Hence, at the end of each phase i, $1 \le i \le h$, we are now in the same situation as in Part 1 of the algorithm; i.e., the line segments and queries which were routed to each node v at level i are ordered such that the next element search problem for each query (with respect to these line segments) can be solved by merging the two ordered sequences (of line segements and query points, respectively) using a bitonic merge procedure.

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

As already indicated above, Parts 3 and 4 of the algorithm are symmetric to Parts 1 and 2, respectively.

Summarizing, we obtain

Theorem 2. *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 in time $O(\log^2 N)$; $N=\max\{m,n\}$.*

5 APPLICATIONS

Consider a subdivision of the plane consisting of n edges, and a set of O(n) query points. The (multiple) *planar subdivision search* problem consists of identifying for each query point p the face of the subdivision containing p [DK]. Goodrich [G] presented an $O(\log n)$ time solution of this problem for a PRAM with n processors (and O(n log n) memory space); an $O(\sqrt{n})$ time algorithm for the $\sqrt{n} \times \sqrt{n}$ mesh-of-processors is described in [JL].

The (multiple) planar subdivision search problem can obviously be reduced to the next element search problem. Hence, as a consequence of Theorem 2, we obtain

Corollary 1. *The (multiple) planar subdivision search problem for a subdivision with n edges, and O(n) query points, can be solved on a hypercube of size n in time $O(\log^2 n)$.*

Theorem 2 also 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 2, we obtain

Corollary 2. *For a set of n disjoint line segments, the trapezoidal map can be computed on a hypercube with 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 2, we obtain

Corollary 3. *An n-vertex simple polygon can be triangulated on a hypercube multiprocessor of size n in time $O(\log^2 n)$.*

6 CONCLUSION

In this paper, we have presented a general technique for implementing standard data structures on a hypercube multiprocessor.

A paradigm frequently used for the design of efficient PRAM algorithms is to use well established standard data structures in a parallel environment by executing (in parallel) a linear number of independant search processes on these structures. We have shown that this paradigm can also be efficiently applied to hypercube multiprocessors for the class of data structures that can be represented by ordered h-level graphs. This follows from an algorithm presented here that solves the m-way search problem for ordered h-level graphs with n nodes and s sources on a hypercube multiprocessor of size N, N=max{n,m}, in time $O(\min\{s \log N, \log^2 N\} + h \log N)$.

We applied this method to the implemention of a segment tree for next element search on a hypercube, and showed that our approach provides an $O(\log^2 n)$ time hypercube algorithm for the next element search problem. As a consequence of this result, we also obtained $O(\log^2 n)$ time solutions to the planar (multiple) point location problem, the trapezoidal map construction problem, and the triangulation problem.

REFERENCES

[ACG] M.J. Atallah, R. Cole, and M.T. Goodrich, "Cascading divide-and-conquer: a technique for designing parallel algorithms", Technical Report CSD-TR-665, Deparetment of Computer Science, Purdue University, 1987.

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

[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.

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

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

[JL] C-S. Jeong and D.T. Lee, "Parallel geometric algorithms on mesh-connected computers", in Proc. Fall Joint Computer Conf., 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.