JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 8, 367–375 (1990)

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

In this paper, we study the problem of implementing standard data structures on a hypercube multiprocessor. We present a technique for efficiently executing multiple independent 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 obtaining $O(\log^2 n)$ time algorithms for solving the next element search problem, the trapezoidal composition problem, and the triangulation problem.

1. INTRODUCTION

One of the main differences, besides the difference in communication delay, between the parallel random access machine (PRAM) and the hypercube processor network is that the PRAM has one large (shared) memory similar to that of 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 implementation, on a PRAM, of well-established data structures like, e.g., segment trees [1, 2, 6] or subdivision hierarchies [5]. Once such a data structure has been built, each processor can search in it, independently of the others, in the standard manner.

For processor networks, the parallel execution of independent queries on one joint data structure is obviously not as straightforward. Most algorithms designed for processor networks are mainly concerned with solving the routing and collision avoidance problem and use only very simple data structures, if any. Another (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.

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

† Research partially supported by the Bell-Northern Research *Graduate Award Program.*

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

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 independently and in parallel. Our solution, which we call *m-way search,* allows an arbitrary number of search queries to access the same node at the same time (this cannot be achieved by, e.g., embedding graphs into hypercubes).

We propose m-way search as a general tool for designing hypercube algorithms. It allows elegant high-level algorithm design, using data structures in a way similar to that of PRAM algorithm design. As long as the underlying graphs are ordered h-level graphs, the obtained methods are only an $O(\log n)$ factor slower than the respective PRAM method.

To demonstrate an application of m-way search, we implement a segment tree [4, 7, 9] and solve the next element search problem on a hypercube. Since the total length of all lists attached to the nodes of a segment tree (for n segments) is $O(n \log n)$, a segment tree construction algorithm requires $O(n \log n)$ memory space. On the PRAM, Atallah, Cole, and Goodrich [1, 6] construct a segment tree (and solve the next element search problem) in $O(\log n)$ time using $O(n)$ processors and $O(n \log n)$ memory space (in an earlier paper, Aggarwal *et al.* [2] solve the same problems in $O(\log^2 n)$ time). On a hypercube multiprocessor, $O(n \log n)$ processors are necessary to obtain $O(n \log n)$ memory space; simulating the PRAM method by Atallah, Cole, and Goodrich on a hypercube would therefore require $O(n \log n)$ processors and $O(\log^3 n)$ time. Applying our m-way search method, we obtain an $O(\log^2 n)$ time algorithm for a hypercube of size $O(n \log n)$. This approach also provides $O(\log^2 n)$ time hypercube algorithms for the trapezoidal map construction problem and the triangulation problem (for simple polygons).

The paper is organized as follows: In Section 2, we define ordered h-level graphs and the associated m-way search

0743-7315/90 $3.00

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 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 note that this algorithm also implies $O(\log^2 n)$ time solutions for the trapezoidal decomposition and the triangulation problem.

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 $L_1, \ldots, L_h$ together with an ordering of the elements in each subset L_i $(1 \leq i \leq h)$.

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:

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

A directed acyclic graph $G = (V, E)$ is called an *ordered h-level graph* if it has the following properties (see Fig. 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 $L_1, \ldots, L_h$ of G such that

(a) every source of G is contained in L_1,

(b) if (v, w) is an edge of G then $\text{Level}(w) = \text{Level}(v) + 1$, and

(c) if $(v, w), (v', w')$ are two edges of G with $\text{Index}(v) < \text{Index}(v')$, then $\text{Index}(w) \leq \text{Index}(w')$.

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

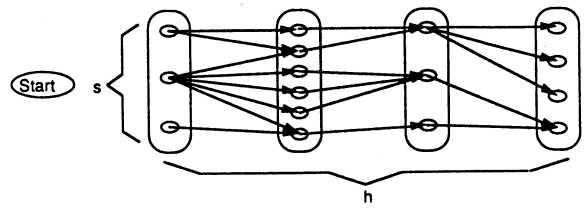

FIG. 1. An ordered h-level graph.

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

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

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

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

For example, if G is a binary search tree then, for every query q, $f(\text{start}, q)$ 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 $\text{Index}(v) = i$ is stored in processor $\text{PE}(i)$, then a *search process* for a query q with search path $(v_1, \ldots, v_h)$ is a process divided into h time steps $t_1 < t_2 < \cdots < t_h$ such that at time t_i, $1 \leq i \leq h$, 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 $Q = \{q_1, \ldots, q_m\} \subseteq U$ 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 implementation 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 $\text{PE}(i)$.

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 [8].

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 $O(\log N)$ [8].

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 [8].

RouteAndCopy([Reg$_1$(i), ..., Reg$_z$(i)], Dest(i), Cond(i)): Under the same assumptions as those 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 [8].

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 [8].

BitonicMerge([Reg$_1$(i), ..., Reg$_z$(i)], Key(i), Left, Peak, Right): This operation is the well known bitonic merge [3]. 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 [3] 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 $G = (V = L_1, \ldots, L_h, E)$ be an ordered h-level graph (with h-partitioning $V = L_1, \ldots, L_h$), 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: $(V \cup \{\text{start}\}) \times U \Rightarrow \mathbb{N}$ and g: $(V \cup \{\text{start}\}) \times U \Rightarrow \{1, \ldots, k\}$ 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 $x \in V \cup \{\text{start}\}$, $q \in U$.

In this section, we consider the problem of solving, on a hypercube multiprocessor, the m-way search problem for a set $Q = \{q_1, \ldots, q_m\} \subseteq U$ 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 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 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.successor$k$(i), respectively; see Fig. 2.

The set $Q = \{q_1, \ldots, q_m\}$ of m queries is stored in arbitrary order, 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.

The global structure of the m-way search algorithm is described in Fig. 3. The m search processes for all m queries

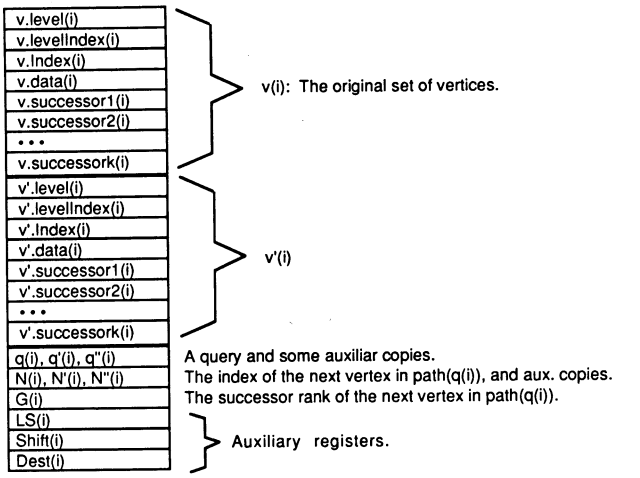

FIG. 2. The registers required at each processor PE(i,j).

$q_1, \ldots, q_m$ are executed in h phases; each phase moves all queries one step ahead in their search paths.

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 ($1 \leqslant x \leqslant h$):

- all queries are sorted with respect to the index of the xth 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 xth node in the search path of q (this is called a *match* of q and the xth node in its search path).

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

In Sections 3.2 and 3.3, we describe the details of Phase 1 and Phase x ($2 \leqslant x \leqslant h$), 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 Fig. 5a. The algorithm consists of four basic steps (see also Fig. 5b for an illustration). 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 $s \geqslant \log(N)$ then bitonic sorting [3] is used; if $s < \log(N)$, a procedure called SortBySourceIndex which sorts the queries in $O(s \log N)$ time is used and will be described below. Note that in many applications s is a constant (e.g., for search trees) and, thus, the sorting step is performed in time $O(\log N)$.

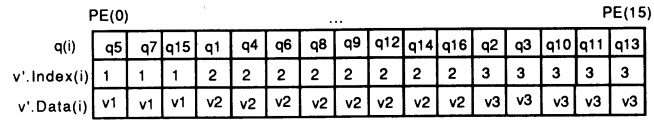

FIG. 4. A typical situation at the end of a phase.

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.

We first discuss the details of procedure SortBySourceIndex; see Fig. 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 $O(s \log N)$.

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 per-

```
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.}
```

FIG. 3. Global structure of the m-way search algorithm.

```
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)
```

FIG. 5a. Outline of Phase 1.

The Initial State on entering Phase 1

v.index(i)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
v.level(i)	1	1	1	2	2	2	2	2	2	3	3	3	4	4	4	4
q(i)	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10	q11	q12	q13	q14	q15	q16

After Step 1 of Phase 1: N(i) = index of the first node in path(q(i)).

N(i)	2	3	3	2	1	2	1	2	2	3	3	2	3	2	1	2

After Step 2 of Phase 1: S(i) = number of sources.

S(i)	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3

After Step 3 of Phase 1: q(i) and N(i) ordered by N(i).

v.index(i)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
v.level(i)	1	1	1	2	2	2	2	2	2	3	3	3	4	4	4	4
q(i)	q5	q7	q15	q1	q4	q6	q8	q9	q12	q14	q16	q2	q3	q10	q11	q13
N(i)	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3

After Step 4, the final step of Phase 1: v'(i) = copy of the first node in path(q(i))

q(i)	q5	q7	q15	q1	q4	q6	q8	q9	q12	q14	q16	q2	q3	q10	q11	q13
N(i)	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3
V'(i)	v1	v1	v1	v2	v2	v2	v2	v2	v2	v2	v2	v3	v3	v3	v3	v3

FIG. 5b. An illustration of Phase 1.

formed in time $O(\log N)$ using the procedure MoveVerticesToQueries described in Fig. 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 ($2 \leq x \leq h$) 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 xth node in their search path, and have each processor PE(i) contain, in its register $v'(i)$, a copy of the xth node in the search path of the query $q(i)$.

An outline of the algorithm for Phase x is given in Fig. 7a; see also Fig. 7b for an illustration. 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 that described in Section 3.2.

What remains to be discussed are the details of procedure OrderQueriesByNextVertex. This procedure, which is described in Fig. 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 $\text{Level}(v) = \text{Level}(v') = x-1$ and w and w' have the same successor rank. If $\text{Index}(v) < \text{Index}(v')$ then $\text{Index}(w) \leq \text{Index}(w')$. 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 out-degree 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 Fig. 8: for each of the k possible successor ranks, the respective subsequence of queries is extracted (Steps 4 and 5), inverted (Step 9), and appended to the sequence of queries already

```
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)
```

FIG. 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)
```

FIG. 6b. Detailed description of procedure MoveVerticesToQueries.

```
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)
```

FIG. 7a. Overview of Phase x, $2 \leq x \leq h$.

Initial input to Phase 2.

q(i)	q5	q7	q15	q1	q4	q6	q8	q9	q12	q14	q16	q2	q3	q10	q11	q13
N(i)	1	1	1	2	2	2	2	2	2	2	2	3	3	3	3	3
V'(i)	v1	v1	v1	v2	v2	v2	v2	v2	v2	v2	v2	v3	v3	v3	v3	v3

After Step 1 of Phase 2: N(i) =index of the next node in path(q(i)); G(i) = its successor rank.

q(i)	q5	q7	q15	q1	q4	q6	q8	q9	q12	q14	q16	q2	q3	q10	q11	q13
N(i)	5	4	5	6	8	7	7	9	6	6	5	9	9	9	9	9
G(i)	2	1	2	2	4	3	3	5	2	2	1	1	1	1	1	1

After Step 2 of Phase 2: q(i) and N(i) have been ordered by N(i).

q(i)	q7	q5	q15	q2	q1	q14	q16	q8	q4	q9	q16	q2	q3	q10	q11	q13
N(i)	4	5	5	5	6	6	6	7	8	9	9	9	9	9	9	9

After Step 3 of Phase 2: v'(i) = copy of the next node in path(q(i)).

q(i)	q7	q5	q15	q2	q1	q14	q16	q8	q4	q9	q16	q2	q3	q10	q11	q13
N(i)	4	5	5	5	6	6	6	7	8	9	9	9	9	9	9	9
v'(i)	v4	v5	v5	v5	v6	v6	v6	v7	v8	v9	v9	v9	v9	v9	v9	v9

FIG. 7b. An illustration of Phase 2.

ordered (Steps 10 and 11), and finally the so created bitonic sequence is converted into a sorted sequence (Step 12).

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 ($1 \leq x \leq h$) all queries are sorted with respect to the index of the xth node in their search path, and each processor PE(i) contains in its register $v'(i)$ a copy of the xth node in the search path of the query currently stored in its register $q(i)$.

```
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)
```

FIG. 8. Details of procedure OrderQueriesByNextVertex.

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 [4]. The segment tree is a widely used structure which has, for example, been utilized to obtain efficient implementations of plane sweep algorithms in computational geometry [4, 7, 9]. 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 segment 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 nonintersecting line segments $I_1, \ldots, I_n$ 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 $p_1, \ldots, p_m$ the line segment I_j first intersected by the ray starting at p_i in direction D_{next} ($m = O(n)$), as illustrated in Fig. 9.

An obvious method for solving the next element search problem is to apply a plane sweep in direction D_{next} using a *segment tree* [4, 7, 9].

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 $(x_1, x_2, \ldots, x_{2n})$ be the sorted sequence of the projections of the $2n$ endpoints of $I_1, \ldots, I_n$ onto the x-axis. The segment tree $T(S) = (V_s, E_s)$ for S is the complete binary tree with leaves $x_1, \ldots, x_{2n}$. For every node v of $T(S)$, an interval $x\text{range}(v)$ is defined as follows:

—if v is a leaf x_i, then $x\text{range}(v) = [x_i, x_{i+1})$. ($[x_{2n}, x_{2n+1}) = [x_{2n}, x_{2n}]$.)

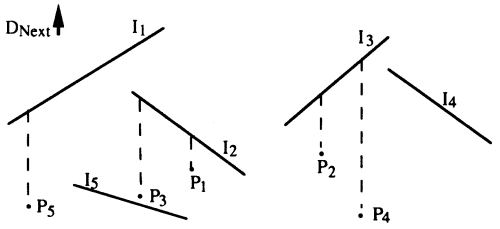

FIG. 9. The next element search problem.

—if v is an internal node, then $x\text{range}(v)$ is the union of all intervals $x\text{range}(v')$ 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 $\text{NL}(v) \subseteq S$ which is defined as follows:

$\text{NL}(v) = \{I \in S \mid x\text{range}(v) \subseteq I^{(x)}$ and not $(x\text{range}(\text{father of } v) \subseteq I^{(x)})\}$.

A segment tree $T(S)$ is an ordered h-level graph where h is the height of $T(S)$; see Fig. 10. For any node v, $\text{Level}(v)$ is the height of v in $T(S)$, $\text{Levelindex}(v)$ is the rank of v in $\{v' \mid \text{Level}(v') = \text{Level}(v)\}$ with respect to the ordering of these nodes by increasing x-coordinate of $x\text{range}(v')$, and $\text{Index}(v)$ is defined by the above as described in Section 2.1.

For every query point p, we define $\text{path}(p)$ to be the path in $T(S)$ from the root to the leaf v such that $p^{(x)} \in x\text{range}(v)$. In order to solve the next element search problem, we first construct the segment tree $T(S)$ and then route every query point p along $\text{path}(p)$. At every node v on the path, the next element of p in $\text{NL}(v)$ is determined (this process will be referred to as *locating* p in $\text{NL}(v)$). For each query point, the final result to be reported is the closest of those next elements.

We show next how to build $T(S)$, in particular how to build the node lists $\text{NL}(v)$. 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 $O(n \log n)$ [7]. 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 *I-path*(l) to be the path from the root of $T(S)$ to the leaf v of $T(S)$ with $a \in x\text{range}(v)$. Likewise we define *r-path*(l) to be the path from the root of $T(S)$ to the leaf v of $T(S)$ with $b \in x\text{range}(v)$. We observe that, if a line segment I is contained in a node list $\text{NL}(v)$, then exactly one of the following four cases applies:

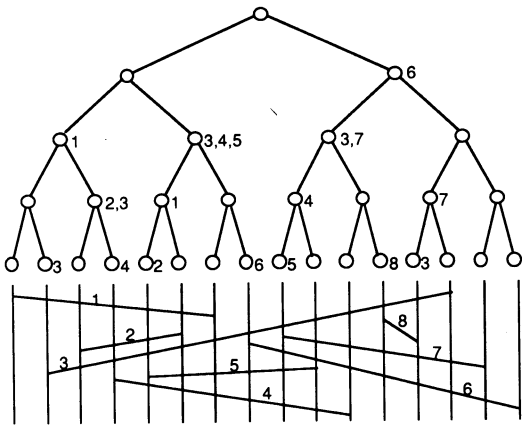

FIG. 10. A segment tree (the numbers associated with the nodes represent the node lists).

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

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

The algorithm for constructing the segment tree $T(S)$ consists of four parts. In Part r, $1 \leq r \leq 4$, all line segments are routed through $T(S)$. When they arrive at the nodes of height i, $1 \leq i \leq h$, the node list $\text{NL}_r(v)$ of all those nodes are created. In order to efficiently determine, for a query point, the next line segment in a node list $\text{NL}(v)$, the segments have to be sorted with respect to the above–below relation within the vertical slab defined by $x\text{range}(v)$. We will create every sublist $\text{NL}_i(v)$ in sorted order; at the end of Part 4, the node lists $\text{NL}(v)$ in sorted order are obtained from the sublists $\text{NL}_i(v)$ by applying bitonic sort [3].

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 segment tree construction algorithm, i.e., how to create the node lists $\text{NL}_1(v)$ for all nodes v. This problem is solved by using m-way search to route every segment $I \in S$ along $I\text{-path}(l)$. When applying the m-way search algorithm of Section 3 to the tree $T(S)$ for this set of queries, at the end of Phase i $(1 \leq i \leq h)$ for every node v with $\text{Level}(v) = i$ there exists a block of consecutively numbered processors containing all line segments s such that $v \in I\text{-path}(s)$. From these, we can immediately extract all line segments $s \in \text{NL}_1(v)$.

What remains to be discussed is how to obtain a sorted ordering of the node lists $\text{NL}_1(v)$. We observe that the m-way search algorithm of Section 3 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 $\text{PE}(j_1)$ and $\text{PE}(j_2)$ with $j_1 < j_2$, and the ith node in $I\text{-path}(q_1)$ is the same as the ith node in $I\text{-path}(q_2)$, then at the end of Phase i the queries q_1 and q_2 are stored in two processors $\text{PE}(j_1')$ and $\text{PE}(j_2')$ 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., $\text{NL}_1(v)$ is in the desired order.

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 $\text{NL}_2(v)$. In contrast to the construction of the node lists $\text{NL}_1(v)$, the ordering of the line segments in $\text{NL}_2(v)$, with respect to the above–below relation in the vertical slab defined by $x\text{range}(v)$, cannot be obtained by using

the sorted order of the left (or right) endpoints of the segments.

Let $v[i,j]$ be the vertex v of $T(S)$ with Level$(v) = i$ and Levelindex$(v) = j$. We introduce $h-1$ new vertices $v[i, 2^{i-1}]$, $1 \leq i \leq h-1$, and define a new tree $T'(S)$ as follows (see Fig. 11): $T'(S) = (V'_s, E'_s)$, where

- $V'_s = (V_s - \{v[i,0] \mid 1 \leq i \leq h-1\}) \cup \{v[i, 2^{i-1}] \mid 1 \leq i \leq h-1\}$, and
- $(v[i,j], v[i',j']) \in E'$ if and only if $((i = h, j \bmod 2 = 1, i' = i-1$, and $j' = (j+1)/2)$ or $(1 < i < h, i' = i-1$, and $j' = \lfloor (j+1)/2 \rfloor^2))$.

For each line segment $I \in S$ let I-path$'(l)$ be the path in $T'(S)$ either from the last node v of I-path(l), 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 $\mathrm{NL}_2(v)$ then v is a node in I-path$'(l)$. We also observe that if I-path$'(l) = (w_1, \ldots, w_h)$ and $I \notin \mathrm{NL}_2(w_i)$, then $I \notin \mathrm{NL}_2(w_j)$ 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 $\mathrm{NL}_2(w_0) \subseteq \mathrm{NL}_2(w_1) \cup \mathrm{NL}_2(w_2)$. Let $x\mathrm{range}(w_0) = [a, b)$ and consider the ordering of $\mathrm{NL}_2(w_0)$ 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 $\mathrm{NL}_2(w_1)$ and $\mathrm{NL}_2(w_2)$ by eliminating from these sequences the elements not contained in $\mathrm{NL}_2(w_0)$ and merging the so obtained subsequences.

The idea for Part 2 of the segment tree construction algorithm is to route all line segments along I-path$'(l)$. Since $T'(S)$ is an ordered h-level 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 m-way search 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 $I \notin \mathrm{NL}_2(v)$. 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 query points p such that w_0 is the ith node in path$'(p)$ and all line segments $I \in \mathrm{NL}_2(w_0)$. In Phase $i-1$, these line segments have been routed to at most two different nodes w_1 and w_2. If $\mathrm{NL}_2(w_1)$ and $\mathrm{NL}_2(w_2)$ were previously ordered as described above, then the same ordering for $\mathrm{NL}_2(w_0)$ can be obtained by extracting the two subsequences of segments previously routed to $\mathrm{NL}_2(w_1)$ and $\mathrm{NL}_2(w_2)$, respectively, and merging these subsequences using a bitonic merge [7]. Since only two line segments were initially routed to every source of $T'(S)$, the orderings of all lists $\mathrm{NL}_2(v)$ can be maintained through all phases with an overhead of $O(\log N)$ steps per phase.

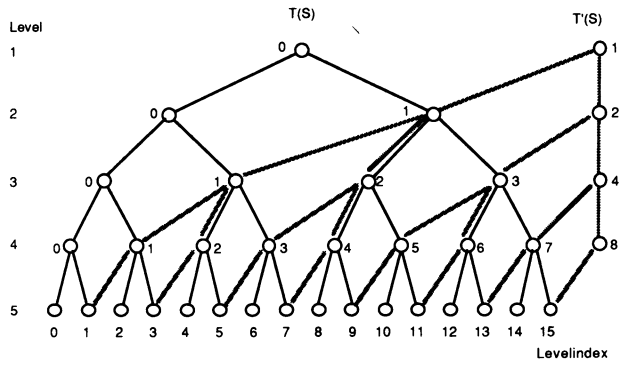

FIG. 11. The trees $T(S)$ and $T'(S)$.

We obtain:

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

In order to solve the next element search problem, we first construct the segment tree $T(S)$ as described above. Then, we route the query points down $T(S)$, constructing a query list $\mathrm{QL}(v)$ for each node v, containing a copy of all those queries that visit v. At the end of every phase of the m-way search algorithms, the created query lists are concentrated and appended to the query lists created in previous phases (using the "Concentrate" and "Route" operations of Section 2.3).

Then, we convert in $O(\log^2 N)$ time the node lists $\mathrm{NL}(v)$ into a forest of binary trees, one for each list. As a result of the segment tree construction method described above, every node list $\mathrm{NL}(v)$ is sorted with respect to the above–below relation in the vertical slab defined by $x\mathrm{range}(v)$. Each node list is converted into a balanced binary search tree $t(v)$ with respect to this ordering. Each element calculates its position in the respective tree, and then a bitonic sort is used to construct the so defined forest.

Now, the next element search problem, for the m query points, can be solved by using another m-way search as follows: The query set is the union of all query lists $\mathrm{QL}(v)$, and the ordered h-level graph is the union of all trees $t(v)$. First, every query point p_i in a query list $\mathrm{QL}(v)$ is routed to the root of $t(v)$. Then, in order to locate p_i in $\mathrm{NL}(v)$, it is routed through $t(v)$ (using the above–below relation between p_i and the edges in $\mathrm{NL}(v)$). Finally, for each point the best of the $O(\log n)$ partial results is obtained by sorting all results (line segments) with the primary key being the query point and the secondary key being the vertical distance to the query point.

Summarizing, we obtain:

THEOREM 3. *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* $O(\log^2 N)$; $N = \max\{m, n\}$.

5. APPLICATIONS

Theorem 3 immediately implies an efficient hypercube solution for another fundamental geometric problem: the construction of the *trapezoidal map* [10].

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 the 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., [6, 10, 11]. Atallah, Cole, and Goodrich [1, 6] 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 3, we obtain:

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

Yap [11] has shown that on a PRAM with $O(n)$ processors and $O(n \log n)$ space, the *triangulation* of a simple polygon (see [10]) can be computed in time $O(\log n)$ by essentially applying two calls of the trapezoidal map algorithm (of [1, 6]). By combining the result in [11] 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* $O(\log^2 n)$.

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, several independent 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 implementation of a segment tree for next element search on a hypercube and showed that 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.

Received February 15, 1989; revised September 21, 1989

REFERENCES

1. Attallah, M. J., Cole, R., and Goodrich, M. T. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM J. Comput.* **18,** 3 (1989), 499–532.
2. Aggarwal, A., Chazelle, B., Guibas, L., O'Dunlaing, C., and Yap, C. Parallel computational geometry. *Algorithmica* **3,** 3 (1988), 293–327.
3. Batcher, K. E. Sorting networks and their applications. *Proc. AFIPS Spring Joint Computer Conference,* 1968, pp. 307–314.
4. Bentley, J. L., and Wood, D. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Trans. Comput.* **29,** 7 (1980), 571–576.
5. Dadoun, N., and Kirkpatrick, D. G. Parallel processing for efficient subdivision search. *Proc. ACM Symposium on Computational Geometry,* 1987, pp. 205–214.
6. Goodrich, M. T. Efficient parallel techniques for computational geometry. Ph.D. thesis, Department of Computer Science, Purdue University, 1987.
7. Mehlhorn, K. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry.* Springer-Verlag, New York/Berlin, 1984.
8. Nassimi, D., and Sahni, S. Data broadcasting in SIMD computers. *IEEE Trans. Comput.* **30,** 2 (1981), 101–106.
9. Preparata, F. P., and Shamos, M. I. *Computational Geometry—An Introduction.* Springer-Verlag, New York/Berlin, 1985.
10. Tarjan, R. E., and Van Wyk, C. J. An $O(n \log \log n)$ time algorithm for triangulating a simple polygon. *SIAM J. Comput.* **17,** (1988), 143–178.
11. Yap, C.-K. Parallel triangulation of a polygon in two calls to the trapezoidal map. *Algorithmica* **3,** 2 (1988), 279–288.

FRANK DEHNE is currently an assistant professor at the School of Computer Science (Center for Parallel and Distributed Computing), Carleton University, Ottawa. He received a M.C.S. degree (Dipl. Inform.) from the Technical University of Aachen (West Germany) in 1983 and a Ph.D. (Dr. rer. nat.) from the University of Würzburg (West Germany) in 1986. His research interests include computational geometry, data structures, and parallel algorithms and VLSI.

ANDREW RAU-CHAPLIN is currently a graduate student at the School of Computer Science (Center for Parallel and Distributed Computing), Carleton University, Ottawa, on educational leave of absence from Bell-Northern Research. He received a B.A. degree from York University, Toronto, in 1986. His research interests include data structures, parallel algorithms, and parallel AI applications.