Computational Geometry: Theory and Applications 2 (1992) 141–167
Elsevier

COMGEO 125

Parallel fractional cascading on hypercube multiprocessors

Frank Dehne*

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

Afonso Ferreira**

CNRS-Laboratoire de l'Informatique, du Parallelisme, Ecole Normale Superieure de Lyon, 69364, Lyon cedex 07, France

Andrew Rau-Chaplin***

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

Communicated by Bernard Chazelle
Submitted 11 February 1991
Accepted 19 February 1992

Abstract

Dehne, F., A. Ferreira and A. Rau-Chaplin, Parallel fractional cascading on hypercube multiprocessors, Computational Geometry: Theory and Applications 2 (1992) 141–167.

In this paper we present a new data-structuring technique for parallel computational geometry on a hypercube multiprocessor. This technique, called *hypercube cascading*, is an efficient parallel implementation of fractional cascading for the hypercube multiprocessor. That is, it allows complex data structures with catalogs to be traversed efficiently in parallel by a large number of simultaneous queries. We show that for *monotone* graphs with n nodes, m multiple look-up queries with path length at most p (including catalog look-ups) can be executed independently, in parallel, in time $O(p \log N + t_s(N))$ on a hypercube multiprocessor of size $N = \max\{n, m\}$. The term $t_s(N)$ denotes the time for sorting N elements on a hypercube of size N; currently $t_s(N) = O(\log N \log\log N)$. Note that, the best known sequential time complexity for one multiple look-up query, as presented by Chazelle and Guibas, is $O(p + \log N)$. Our solution allows an arbitrary number of search queries to access the same node and its catalog at

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

** Currently on leave from the University of Sao Paulo (Brazil), project BID/USP. Most of this work was done while the second author was visiting the Center for Parallel and Distributed Computing of the School of Computer Science at Carleton University. Support from the Centre Jacques Cartier is acknowledged.

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

the same time. We present two parallel computational geometry applications of this technique: multiple stabbing of a simple polygonal path and multiple slanted range search.

1. Introduction

Fractional cascading is a powerful general technique for deriving efficient sequential computational geometry algorithms [7, 8]. In the sequential domain, researchers have applied this data structuring technique to derive efficient solutions to a wide range of geometric problems. In this paper, we demonstrate how the fractional cascading technique can be supported on hypercube multiprocessors. Our method, which we will refer to as *hypercube cascading*, provides a new tool for parallel computational geometry on hypercubes. For many sequential algorithms based on fractional cascading, hypercube cascading provides a standardized method for obtaining an efficient parallel hypercube algorithm.

When designing parallel computational geometry algorithms, many researchers prefer the PRAM model over processor network models. A primary reason for this is that 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 has been successfully used in applying fractional cascading for the design of geometry algorithms on the PRAM (see, e.g., [1–3, 14]).

For processor networks, the parallel execution of independent queries on one joint data structure is not as straight forward. Computational geometry algorithms designed for processor networks have therefore tended to be not as elegant as their PRAM counterparts. In general, network algorithms use only very simple data structures, if any, and are mainly concerned with solving routing and collision avoidance problems.[1] This paper will demonstrate that advanced data structures based on fractional cascading [7, 8], which are commonly used in sequential computational geometry, can also be efficiently utilized on hypercube multiprocessors.

Consider a catalog graph of size n and bounded (fixed) degree, and a set of m multiple look-up queries along paths of length at most p [7]; see Sections 1.1 and 2 for definitions. We show that, if the graph is *monotone* (to be defined in Section 2), such m multiple look-up queries (including catalog look-ups) can be executed independently, in parallel, in time $O(p \log N + t_s(N))$ on a hypercube multiprocessor of size $N = \max\{n, m\}$. The term $t_s(N)$ denotes the time for sorting N elements on a hypercube of size N; currently $t_s(N) = O(\log N \log\log N)$ [10, 15].

[1] A more elegant approach to simulate PRAM algorithm on processor networks; the obtained results are however in most cases less efficient than algorithms designed directly for specific networks.

Note that, the best known sequential time complexity for one multiple look-up query, as presented by Chazelle and Guibas [7], is $O(p + \log N)$.[2] Our solution *allows an arbitrary number of search queries to access the same node and its catalog at the same time.*[3] 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). The parallel hypercube implementation of fractional cascading presented in this paper is completely different from the sequential algorithm presented in [7].

Our earlier work on implementing data structures on a hypercube was presented in [13]. The methods presented there supported the routing of n queries through a layered graph without catalogs. The algorithms described in [13] could not support fractional cascading, and were more restrictive in terms of the class of graphs considered and the manner in which queries could move. The hypercube cascading technique described in this paper solves these problems by extending our previous results as follows:

- The class of graphs considered is extended from *ordered h-level-graphs* (see [13]) to the more general class of *monotone graphs* (to be defined in Section 2).
- Hypercube cascading supports the addition of catalogs (of more than constant size) to each node of the graph. It allows each query to execute a catalog lookup at each visited node without increasing the overall time complexity (compared to the method in [13] without catalog look-ups).
- It allows more general movement of queries in the graph than the method in [13]. Queries are now allowed to move along edges in either direction (rather than only the direction of the edge, as in [13].

To demonstrate the use of our new technique, we present two new parallel computational geometry algorithms for the hypercube. Both are derived by applying hypercube cascading to Chazelle and Guibas' sequential algorithms for those problems. We present a hypercube solution for the *multiple stabbing problem,* i.e., the problem of computing all intersections of m lines with a simple polygonal path of length n. Our algorithm, described in Section 4, has time complexity

$$O\left(k \log \frac{N}{k} \log N + t_s(n)\right)$$

on a hypercube multiprocessor of size $N = \max\{n, m\}$, where k is the maximum number of intersections per line. Furthermore, we study the *multiple slanted range search problem* which consists of answering, in parallel, m slanted range search queries on one set S of n points [8]. We show that, with a preprocessing of $O(\log^3 N)$, m slanted range search queries (with a maximum of k results per query) can be answered on a hypercube of size $N = \max\{n, m\}$ in time $O(k \log N + t_s(N))$.

[2] This implies directly a $O(p + \log N)$ time PRAM algorithm for m multiple look-up queries.
[3] This cannot be achieved by, e.g., embedding graphs into hypercubes.

1.1. Problem overview

Fractional cascading, as described by Chazelle and Guibas [7], is a general data structuring technique in which catalog graphs are used to represent data structures. A catalog graph, G, is a directed acyclic graph where each node in the graph has associated with it a catalog of ordered items from some ordered domain U. Fig. 1 shows an example of a data structure represented as a catalog graph.

The *iterative search problem* consists of routing an *iterative search query* through G, where at each node v visited, the decision as to which node to visit next depends on a query look-up in v's catalog. An obvious sequential algorithm to perform this type of search is to route the query through the catalog graph while performing binary search at each step along the query's path. This naive approach has a $O(p \log N)$ time complexity, where p is the length of the path. Chazelle and Guibas' main result in [7] is that, in the sequential setting, this can be improved to time $O(p + \log N)$.

In the parallel setting, the analogous problem is considered in terms of one catalog graph (of size n) and a set of m search queries, each with a path length of, at most, p. This problem, referred to as the *multi iterative search problem,* can be solved on the PRAM with $N = \max\{n, m\}$ processors as follows. First, all processors work together in order to build the catalog graph. Then, each processor independently executes the sequential algorithm of fractional cascading for a single query. The PRAM's concurrent read capability permits one shared data structure to be traversed concurrently by all processors without interference. Such an approach provides $O(\log n)$ time parallel solutions for several computational geometry problems [1, 14].

In this paper, we study the multi iterative search problem for the hypercube multiprocessor. For this model, the parallel execution of independent queries on one joint data structure is not as straight forward. There are two obvious problems: instead of having one large shared memory that resembles memory in the sequential model, the hypercube's memory is distributed over the network, and the hypercube does not provide a concurrent read capability. Our main result is that for data structures which are *monotone* catalog graphs (to be defined in

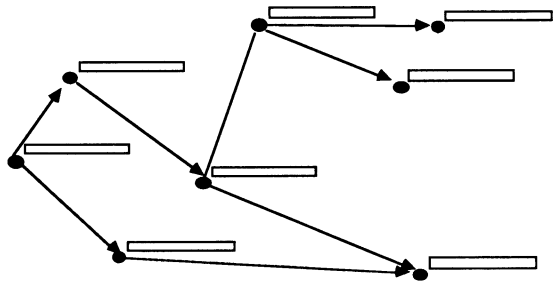


Fig. 1. An example catalog graph.

Section 2) of bounded (fixed) degree, m multi iterative search queries (including catalog lookups) can be executed independently, in parallel, in time $O(p \log N + t_s(N))$ on a hypercube multiprocesssor of size $N = \max\{n, m\}$. We will refer to this technique as *hypercube cascading*. It is important to note that our solution allows an arbitrary number of search queries to access the same node and its catalog at the same time.

Compared to the sequential methods presented in [7], we need to impose restrictions on the type of catalog graphs (monotonicity) as well as the type of search queries. We demonstrate, however, that, despite these restrictions, our method does provide hypercube implementations of many data structures (see Sections 2, 4, and 5).

The remainder of this paper is organized as follows. In Section 2 the original fractional cascading technique from Chazelle and Guibas is briefly reviewed. Section 3 is devoted to the introduction and analysis of the hypercube cascading technique. Applications of this new technique to computational geometry are shown in Sections 4 and 5. Section 6 contains concluding remarks.

2. Definitions

Consider a directed, connected, and planar graph $G = (V, E)$. The graph is also assumed to have bounded (fixed) degree; that is, there exists 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. As in [7], we associate with each vertex v of G a *catalog* C_v consisting of an 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, where N is the number of nodes plus the sum of the sizes of all catalogs.

A *path* π in G (of length p) is a sequence of vertices $v_0, v_1, \ldots, v_{p-1}$ such that for each $0 \leq i < p$, 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 $\sigma(q, C_{v_i})$.

It is shown in [7] that in $O(N)$ time and space it is possible to construct, for the standard sequential machine model, a data structure that solves the iterative search problem for a multiple look-up query with path length p in time $O(p + \log N.)$

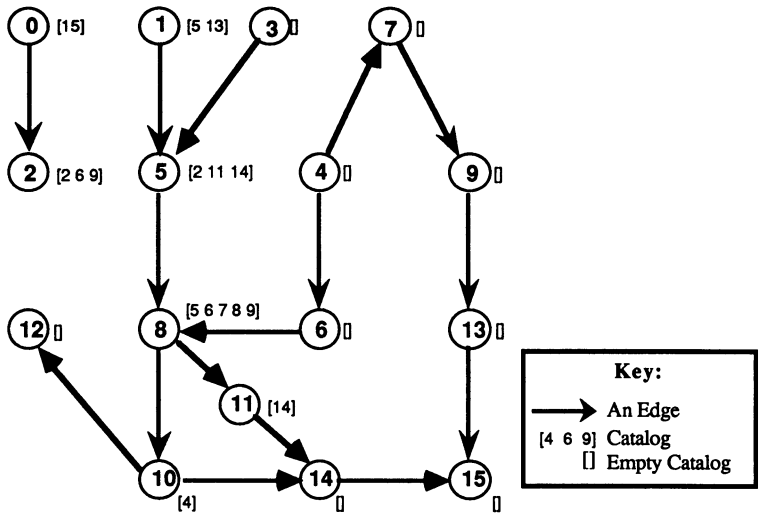


Fig. 2. An example of a monotone catalog graph.

We study how to obtain a parallel implementation of fractional cascading on a hypercube multiprocessor. Compared to the sequential methods presented in [7], 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: $V \Rightarrow \{1, \ldots, n\}$ with the following property: if (v, v') or (w, w') are two edges of G with $\text{Index}(v) < \text{Index}(w)$ then $\text{Index}(v') \leq \text{Index}(w')$. Fig. 2 shows an example of a monotone catalog graph of size 16.

Montone catalog graphs are not as general as the class of planar graphs used by Chazelle and Guibas [7, 8], but our experience suggests that most data structures can be modeled by monotone graphs. The only non monotone data structure we have found to date is Chazelle's hive graph [9]. Note that the class of monotone graphs includes all k-nary trees ($k = O(1)$). It is also less restrictive than the class of ordered h-level graphs considered in [13]. Fig. 3 demonstrates the relationship between these classes of graphs.

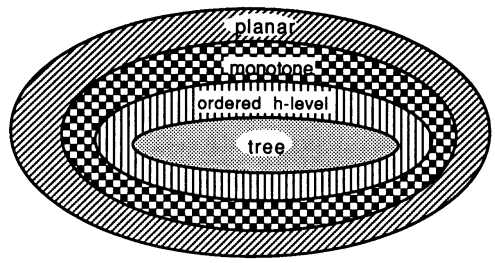


Fig. 3. Monotone graphs in relation to several other graph classes.

On the other hand, as we will see in the next section, the restriction of fractional cascading to monotone graphs allows for more efficient parallel hypercube implementations based on faster 'monotone' routing [16] rather than general routing.

3. Hypercube cascading

We show how, given a *monotone* catalog graph of size n and a set $Q = \{q_1, \ldots, q_m\}$ of m multiple look-up queries along paths of length at most p, these m multiple lock-up queries can be executed independently, in parallel, in time $O(p \log N + t_s(N))$ on a hypercube multiprocessor of size $N = \max\{n, m\}$[4]. We first describe the assumed initial configuration of the hypercube, and then present details of the different phases of the algorithm.

3.1. Initial configruation

The graph G is assumed to be stored in the hypercube such that each vertex v with $\text{Index}(v) = i$ is stored in register $v(i)$ of processor PE(i). For every vertex v, let $\text{pred}(v)$ and $\text{succ}(v)$ be the sets of possible predecessors 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 $v.\text{index}(i)$, $v.\text{successor}_1(i), \ldots, v.\text{successor}_{\mu_{\text{out}}}(i)$, and $v.\text{predecessor}_1(i), \ldots, v.\text{predecessor}_{\mu_{\text{in}}}(i)$, storing $\text{Index}(v)$, the indices of the vertices in $\text{succ}(v)$, and the indices of the vertices in $\text{pred}(v)$, respectively. Every register $v(i)$ also has a field $v.\text{EndCat}(i)$ storing the address (processor number) of the last record of the associated catalog.

We assume that each PE(i) also has a register $c(i)$ to store a catalog record. Each record $c(i)$ contains a field $c.\text{index}(i)$ storing the index of the associated vertex of G and a field $c.\text{key}(i)$. 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 (using $c.\text{key}(i)$).

For a given set $Q = \{(q_1, \pi_1), \ldots, (q_m, \pi_m)\}$ of m multiple look-up queries, every processor PE(i) stores in its register $q(i)$ one arbitrary query value q_j. The search paths π_j $(1 \leq j \leq m)$ are determined *on-line* by the following two functions:

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

For each q_j, $\text{start}(q_j)$ is the index of the first vertex v_1 in its search path π_j. Assume that the xth 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 $v.\text{predecessor}_{-y}(i)$ is the index of the $(x+1)$st vertex v_{x+1} of π_j; otherwise, $v.\text{successor}_y(i)$ is the index of the $(x+1)$st vertex. It

[4] We assume, w.l.o.g., that $N = 2^d$; otherwise we at most double the data set to the next power of 2.

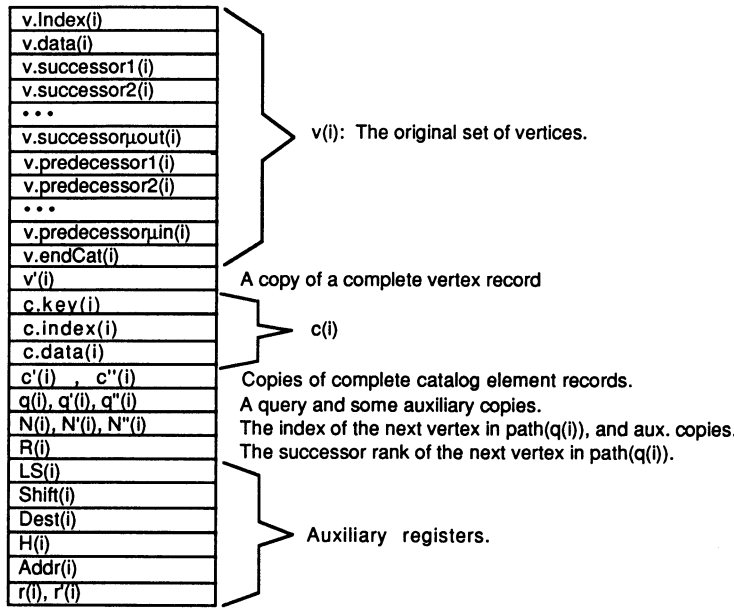


Fig. 4. The registers required at each processor PE(i).

is required that both functions, *start* and g, can be calculated in by one processor in time $O(\log N)$. Note that, in most cases, as for our example applications, O(1) time suffices.

Fig. 4 shows the set of registers necessary at every processor PE(i). 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 $Dest(i)$.

3.2. Algorithm overview

The global structure of the hypercube cascading algorithm is described in Fig. 5. The iterative search processes for all m queries $q_1, \ldots, q_m$ are executed in p phases; each phase moves all queries one step along their search paths.

The key idea is that, in Phase x $(1 \leq x \leq p)$, 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

Hypercube Cascading:
(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.

Fig. 5. Global structure of the hypercube cascading algorithm.

$q(i)$	q_5	q_7	q_1	q_4	q_{10}	q_{15}	q_2	q_{11}	q_6	q_{13}	q_0	q_{12}	q_8	q_{15}	q_3	q_9
$N(i)$	0	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11
$v'(i)$	v_0	v_0	v_1	v_1	v_1	v_1	v_3	v_3	v_5	v_5	v_7	v_7	v_8	v_8	v_{11}	v_{11}
$c''.key(i)$	15	15	5	5	13	-1	-1	-1	11	14	-1	-1	8	-1	14	14

Fig. 6. A typical situation at the end of a Phase.

complexity, the algorithm first permutes the queries (in registers $q(i)$) such that they are sorted with respect to the index of the xth node in their search path. It then creates, in registers $v'(i)$, copies of the respective nodes. The algorithm ensures 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 xth node in the search path of q_j (we will call this a *match* of q_j with the xth node in its search path). Finally, for each node v all queries that have v as the xth vertex in their search path are merged with C_v into one sorted list, determining for each query its successor in C_v.

The details of the individual phases are explained in the following sections. All procedures used in those phases have time complexity $O(\log N)$, and are composed of a constant number of calls either to the well known monotonic routing operations defined in [16] or to bitonic merge [4], see Appendix A. The only exception is the procedure Sort, to be used in Phase 1 of the hypercube cascading algorithm (Section 3.3). Its time complexity, $t_s(N)$, is currently $O(\log N \log\log N)$ [10, 15].

A typical situation at the end of a phase is depicted in Fig. 6; each vertical column represents the registers $q(i)$, $N(i)$, $v'(i)$ and $c''(i)$ of a processor PE(i).

In the following sections we will present details of Phase 1 and Phase x ($2 \leq x \leq p$), respectively. The first phase is different from the remaining phases because it starts with an arbitrary permutation of the queries.

3.3. Phase 1 of the hypercube cascading algorithm

An outline of Phase 1 is given in Fig. 7. The algorithm consists of five basic steps (see also Fig. 8 for an illustration).

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

Fig. 7. Outline of Phase 1.[5]

[5] Consult Appendix A for the definitions of the elementary operation used in this and following code fragments.

Algorithm Phase$_1$:
Input: The Vertices of the catalog graph $v(i)$, and the set of catalog elements $c(i)$.
Output: Every processor PE(i) stores a query $q(i)$, a copy of the first vertex $v'(i)$ on the $q(i)$'s path, and the result of the catalog search $c''(i)$. Note that the queries $q(i)$ are now sorted with respect to $N(i)$.
Method: (See Fig. 7) First, in Step 1, 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, 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, the

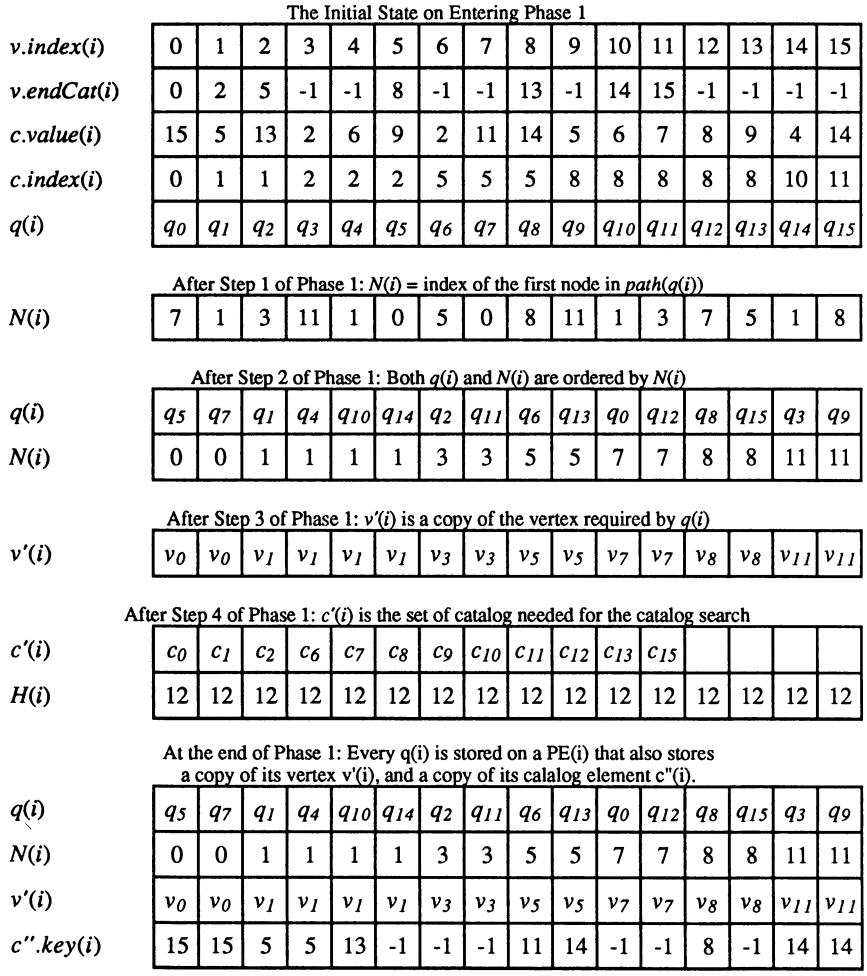


Fig. 8. An illustration of Phase 1.

```
MoveVerticesToQueries:
(1)  IdentifyBlockTail(N(i),N'(i))
(2)  Every PE(i): if N'(i) = N(i) then Dest(i) := -1
                  else Dest(i) := i
(3)  Every PE(i): addr(i) := -1
(4)  Route({[Dest(i),addr(i)]}, N(i), N'(i)≠N(i))
(5)  RouteAndCopy({[v(i),v'(i)]}, addr(i), addr(i)≠-1)
```

Fig. 9. Sketch of Procedure MoveVerticesToQueries.

catalogs associated with the current vertices are selected and, for each query, the successor record in the respective catalog is determined.

Running time: Step 1 is implemented in time O(1), whereas the time complexity of Step 2 is $t_s(N) = \Omega(\log N)$. As we will see in the following, Steps 3 to 5 take $O(\log N)$ time each. Therefore the overall running time is $O(t_s(N))$.

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 performed in time $O(\log N)$ using the procedure Move VerticesToQueries described in Fig. 9.

See Fig. 10 for an illustration of the operation of this procedure.

The Initial State on MoveVerticesToQueries

$v(i)$	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	v_{15}
$q(i)$	q_5	q_7	q_1	q_4	q_{10}	q_{14}	q_2	q_{11}	q_6	q_{13}	q_0	q_{12}	q_8	q_{15}	q_3	q_9
$N(i)$	0	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11

After Step 1: If $N(i) \neq N'(i)$ then i is the tail of a block

$N'(i)$	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11	12

After Step 2: $Dest(i) = i$ for block tails otherwise -1

$Dest(i)$	-1	1	-1	-1	-1	5	-1	7	-1	9	-1	11	-1	13	-1	15

After Step 3: Initialize all $addr(i)$

$addr(i)$	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

After Step 4: $addr(i)$ = Address of last $q(i)$ requiring $v(i)$

$addr(i)$	1	5	-1	7	-1	9	-1	11	13	-1	-1	15	-1	-1	-1	-1

After Step 5: $v'(i)$ = a copy of the vertex required by $q(i)$

$q(i)$	q_5	q_7	q_1	q_4	q_{10}	q_{15}	q_2	q_{11}	q_6	q_{13}	q_0	q_{12}	q_8	q_{15}	q_3	q_9
$N(i)$	0	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11
$v'(i)$	v_0	v_0	v_1	v_1	v_1	v_1	v_3	v_3	v_5	v_5	v_7	v_7	v_8	v_8	v_{11}	v_{11}

Fig. 10. An example trace of MoveVerticesToQueries.

SelectCatalogs:
(1) IdentifyBlockTail($N(i),N'(i)$)
(2) RouteAndCopy([[$v'.index(i),Dest(i)$]], $v'.endCat(i)$, $N(i)\neq N'(i)$ and $v'.endCat(i)\neq -1$)
(3) Number($H(i),c.index(i) = dest(i)$)
(5) Concentrate([[$c(i),c'(i)$]], $c.index(i) = dest(i)$)

Fig. 11. Sketch of Procedure SelectCatalogs.

Algorithm MoveVerticesToQueries
Input: A set of queries $q(i)$, sorted by their associated $N(i)$ register which stores the index of the next vertex required by each query.
Output: Each query $q(i)$ is matched with (i.e., stored in the same processor as) a copy of the vertex v with index $N(i)$.
Method: (See Fig. 9) The idea is to identify for each node that needs to be matched (those with $\text{Dest}(i) \neq -1$), the largest address of the block of queries to be matched with (Steps 1 to 4), and then broadcast each node to the respective block of queries (Step 5).
Running time: $O(\log N)$.

Having matched each query $q(i)$ with a copy of the next vertex in its search path $v'(i)$, it is now necessary to perform the catalog look up. Before catalog look up search can be performed, the set of catalog elements associated with

The initial state on entering SelectCatalogs

$q(i)$	q_5	q_7	q_1	q_4	q_{10}	q_{14}	q_2	q_{11}	q_6	q_{13}	q_0	q_{12}	q_8	q_{15}	q_3	q_9
$N(i)$	0	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11
$v'(i)$	v_0	v_0	v_1	v_1	v_1	v_1	v_3	v_3	v_5	v_5	v_7	v_7	v_8	v_8	v_{11}	v_{11}
$v'.endCat(i)$	0	0	2	2	2	2	-1	-1	8	8	-1	-1	13	13	15	15
$c(i)$	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}
$c.index(i)$	0	1	1	2	2	2	5	5	5	8	8	8	8	8	10	11

After Step 1: If $N(i) \neq N'(i)$ then i is the tail of a block

$N'(i)$	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11	12

After Step 2: If $Dest(i) \neq -1$ then catalog element $c(i)$ has been selected

$Dest(i)$	0	1	1	5	5	5	5	5	5	8	8	8	8	8	11	11

After Step 3: $H(i)$ is the total number of selected catalog elements

$H(i)$	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12	12

At the end of SelectCatalogs: $c'(i)$ = a copy of all selected calalog elements

$c'(i)$	c_0	c_1	c_2	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{15}				

Fig. 12. An example trace of Procedure SelectCatalogs.

vertices currently being visited by queries must be identified. This operation is performed by the algorithm SelectCatalogs described in Fig. 11.

See Fig. 12 for an illustration of the operation of this procedure.

Algorithm SelectCatalogs:
Input: The set of catalog elements, $c(i)$.
Output: The set of catalog elements $c'(i)$ associated with vertices currently being visited by queries.
Method: (see Fig. 11) First, in Step 1, the tail of each block formed by register $N(i)$ is identified. Then, in Step 2, every catalog element associated with a vertex currently being visited by a query is selected. Lastly, the selected catalog elements are concentrated into register $c'(i)$ and counted.
Running time: O(log N).

Having selected the set of required catalog elements, we can now perform the catalog search of each query $q(i)$ in its catalog. At the end of the search procedure, each PE(i) storing a query $q(i)$ and a vertex $v'(i)$ will also store a catalog element $c''(i)$ which is the result of searching with query $q(i)$ in the catalog of vertex $v'(i)$. The catalog search is performed by algorithm Search-CatalogsForQueries described in Fig. 13.

Procedure SearchCatalogsForQueries is the last step in Phase 1. Fig. 14 illustrates the operation of this procedure.

Algorithm SearchCatalogsForQueries:
Input: The set of queries $q(i)$ sorted by the index of the vertex they are currently visiting and the set of catalog elements of those vertices $c'(i)$ (also sorted).
Output: Register $c''(i)$ containing a copy of the catalog element found in the catalog search, if any. The value -1 will be used to indicate no catalog element found.
Method: (see Fig. 13) First, in Step 1, the current location of each query is recorded in q . currentIndex(i). In Steps 2 and 3, for each query $q(i)$ its successor in the associated catalog is computed. This is obtained by a biotonic merge of the queries and catalog elements into a simulated double length register $r(i)$. The principal keys for the merge are the indices of the vertices being visited by the queries' $v'(i)$ and the catalogs' indices c' . index(i). The secondary keys are the queries' and catalogs' values q . key(i) and c' . key(i) respectively. When the

SearchCatalogsForQueries:
(1) *q.currentIndex(i) := i*
(2) Merge({[*q(i),q(i)*]},*N,c'(i),H(0),r(i)*)
(3) RouteAndCopy({[*r(i),r'(i)*]},*i,r(i)* = a catalog element)
(4) Route({[*r'(i),c''(i)*]},*r.currentIndex(i),r(i)* = a query)
(5) Every PE(i): If *c''.index(i)* ≠ *N(i)* then *c''(i)* := -1

Fig. 13. Sketch of Procedure SearchCatalogsForQueries.

The initial state on entering SearchCatalogsForQueries

$q(i)$	q_5	q_7	q_1	q_4	q_{10}	q_{14}	q_2	q_{11}	q_6	q_{13}	q_0	q_{12}	q_8	q_{15}	q_3	q_9
$N(i)$	0	0	1	1	1	1	3	3	5	5	7	7	8	8	11	11
$c'(i)$	c_0	c_1	c_2	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{15}				
$c'.key(i)$	15	5	13	2	6	9	5	6	7	8	9	14				
$c'.index(i)$	0	1	1	5	5	5	8	8	8	8	8	11				

After Step 2: $r(i)$ (a simulated double length register) = the result of merging registers $c'(i)$ and $q(i)$

$r(i) =$																
	q_5	q_7	c_0	q_1	q_4	c_1	q_{10}	c_2	q_{14}	q_2	q_{11}	c_6	q_6	c_7	c_8	q_{13}
	q_0	q_{12}	c_9	c_{10}	c_{11}	q_8	c_{12}	c_{13}	q_{15}	q_3	q_9	c_{15}				

After Step 3: if r(i) = a query then $r'(i)$ (a simulated double length register) = the result of the search with $q(i)$ in the catalog of $v'(i)$.

$r'(i) =$																
	c_0	c_0	c_0	c_1	c_1	c_1	c_2	c_2	c_6	c_6	c_6	c_6	c_7	c_7	c_8	c_9
	c_9	c_9	c_9	c_{10}	c_{11}	c_{12}	c_{12}	c_{13}	c_{15}	c_{15}	c_{15}	c_{15}				

After Step 4: if $c''(i) = N(i)$ then $C''(i)$ the result of the search with $q(i)$ in the catalog of $v'(i)$.

$c''(i)$																
	c_0	c_0	c_1	c_1	c_2	c_6	c_6	c_6	c_7	c_9	c_9	c_9	c_{12}	c_{15}	c_{15}	c_{15}

After Step 5: c''(i) = the result of the search with q(i) in the catalog of v'(i).

$c''(i)$																
	c_0	c_0	c_1	c_1	c_2	-1	-1	-1	c_7	c_9	-1	-1	c_{12}	-1	c_{15}	c_{15}

Fig. 14. An example trace of procedure SearchCatalogsForQueries.

merge step is finished, for every query its successor catalog element is the first catalog element to its right. As several queries may have the same successor, Step 3 is required to make copies of such catalog elements. In Step 4 the located catalog elements are routed back to the processor holding the queries that requested them. Some catalog searches may result in no catalog element being found for a particular query $q(i)$. In these cases Step 5 sets the register $c''(i)$ to -1.

3.4. Phase x ($2 \leq x \leq p$) of the hypercube cascading algorithm

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 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 query $q(i)$ and (in register $c'(i)$) a copy of the successor of $q(i)$ in $C_{v'(i)}$.

Algorithm Phase (x):
Input: The vertices $v(i)$ of the catalog graph, the set of catalog elements $c(i)$, and a set of queries $q(i)$ sorted with respect to the index of the $(x-1)$th node in their search path.
Output: Every query $q(i)$ is stored on a processor PE(i) with a copy of the xth vertex on $q(i)$'s path, and the result of the catalog search is stored in $c''(i)$. Note that, the queries $q(i)$ are sorted with respect to $N(i)$.
Method: (see Fig. 15) 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, storing these two numbers in the auxiliary registers $R(i)$ and $N(i)$ respectively. Note that, if the query has to be routed in opposite direction of an edge in graph G (backwards), 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 backwards 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 $O(\log N)$: Steps 3 to 5 are the same as in Phase 1 described in Section 3.3; Step 2 is explained in the remainder.
Running time: $O(\log N)$.

See Fig. 16 for an illustration of the operation of this procedure.

What remains to be discussed is procedure OrderQueriesByNextVertex. This procedure, which is sketched in Fig. 17, 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. It allows queries to move along any edge to any connected vertex.

```
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
```

Fig. 15. Sketch of Phase x, $2 \leq x \leq p$.

The Initial State on Entering Phase x

$v.index(i)$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$v.endCat(i)$	0	2	5	-1	-1	8	-1	-1	13	-1	14	15	-1	-1	-1	-1
$c.value(i)$	15	5	13	2	6	9	2	11	14	5	6	7	8	9	4	14
$c.index(i)$	0	1	1	2	2	2	5	5	5	8	8	8	8	8	10	11
$q(i)$	q_5	q_7	q_1	q_4	q_{10}	q_{14}	q_2	q_{11}	q_6	q_{13}	q_0	q_{12}	q_8	q_{15}	q_3	q_9

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

$N(i)$	2	2	5	5	5	5	5	5	8	3	4	9	11	10	14	8

After Step 2 of Phase 1: Both $q(i)$ and $N(i)$ are ordered by $N(i)$

$q(i)$	q_5	q_7	q_1	q_{13}	q_1	q_4	q_{10}	q_{14}	q_2	q_{11}	q_6	q_9	q_{12}	q_{15}	q_8	q_3
$N(i)$	2	2	3	4	5	5	5	5	5	5	8	8	9	10	11	14

After Step 3 of Phase 1: $v'(i)$ is a copy of the vertex required by $q(i)$

$v'(i)$	v_2	v_2	v_3	v_4	v_5	v_5	v_5	v_5	v_5	v_5	v_8	v_8	v_9	v_{10}	v_{11}	v_{14}

After Step 4 of Phase 1: $c'(i)$ is the set of catalog needed for the catalog search

$c'(i)$	c_2	c_3	c_4	c_5	c_8	c_9	c_{10}	c_{11}	c_{14}							
$H(i)$	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13	13

At the end of Phase 1: Every $q(i)$ is stored on a PE(i) that also stores a copy of its vertex $v'(i)$, and a copy of its calalog element $c''(i)$.

$q(i)$	q_5	q_7	q_1	q_{13}	q_1	q_4	q_{10}	q_{14}	q_2	q_{11}	q_6	q_9	q_{12}	q_{15}	q_8	q_3
$N(i)$	2	2	3	4	5	5	5	5	5	5	8	8	9	10	11	14
$v'(i)$	v_2	v_2	v_3	v_4	v_5	v_5	v_5	v_5	v_5	v_5	v_8	v_8	v_9	v_{10}	v_{11}	v_{14}
$c''.key(i)$	2	2	-1	-1	2	2	11	14	14	-1	7	9	-1	4	14	-1

Fig. 16. Example of Phase $x = 2$.

```
OrderQueriesByNextVertex:
(1)  Every PE(i): Shift(i):=0
(2)  FOR r:= -μin,...,-1,1,...,μout DO
(3)        Concentrate([[q(i),q'(i)], [N(i),N'(i)]],G(i)=r)
(4)        Number(LS(i), G(i)=r)
           ls := LS(0)
           shift := Shift(0)
(5)        Reverse([[q'(i),q'(i)],[N'(i),N'(i)]],0,ls)
(6)        Route([[q'(i),q''(i)],[N'(i),N''(i)]], i+shift, i<ls)
(7)        Merge([[q''(i),q''(i)],[N''(i),N''(i)]],N''(i),0,shift,shift+ls)
           Every PE(i): Shift(i):=Shift(i)+LS(i)
(8)  Every PE(i): q(i):=q''(i), N(i):=N''(i)
```

Fig. 17. Sketch of Procedure OrderQueriesByNextVertex.

Algorithm OrderQueriesByNextVertex:
Input: A set of n unsorted queries $q(i)$ each with an associated $N(i)$ specifying the index of the next vertex in each queries search path.
Output: The n queires $q(i)$ along with their $N(i)$ sorted with respect to $N(i)$.
Method: (see Fig. 17) 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 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 $\text{Index}(v) < \text{Index}(v')$ then $\text{Index}(w) \leq \text{Index}(w')$. Therefore, the sub-sequence 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 $\mu_{\text{out}} = O(1)$ such subsequences. The new ordering of the queries can therefore be created in time $O(\mu_{\text{out}} \log N) = O(\log N)$ by successively extracting these μ_{out} ordered sub-sequences and merging them in μ_{out} bitonic merge steps.

For the backward edges, the same idea applies because a monotone graph has the same monotonicity properties forwards or backwards. Thus, the same steps described above can be used in order to sort the backward queries with respect to the indices of the next nodes in the search path. As in the previous case, the time for sorting them is $O(\mu_{\text{in}} \log N) = O(\log N)$, since μ_{in} is the constant maximum in-degree in the graph G.

The algorithm is sketched in Fig. 17: for each of the $\mu_{\text{in}} + \mu_{\text{out}}$ possible values of $R(i)$, the respective sub-sequence of queries is extracted (Step 3), inverted (Step 5), appended to the sequence of queries already ordered (Step 6), and finally the newly created bitonic sequence is converted into a sorted sequence by a bitonic merge (Step 7).
Running time: $O(\log N)$.

Note that the monotonicity of the graph G is crucial for the efficient implementation of the preceeding procedure. For general graphs, this step might require an arbitrary permutation to be performed on the hypercube.

3.5. Summary of results

The following lemma and theorem summarize our results.

Lemma 1. *The multi iterative search algorithm described above consists of p phases such that at the end of Phase x ($1 \leq x \leq p$) all queries are sorted (in registers $q(i)$) with respect to the index of the xth node in their search path, and each processor* PE(i) *contains in its registers $v'(i)$ and $c'(i)$ a copy of the xth node v_x in the search path of $q(i)$ and a compy of the successor record of $q(i)$ in C_{v_x}.*

Note that all the sub algorithms used in Phase(x) require $O(\log N)$ time. Hence, the time complexity of algorithm Phase(x) as a whole is $O(\log N)$. Thus, we obtain the following.

Theorem 1. *For a monotone catalog graph of size n (and fixed degree), m iterative search queries along paths of length at most p can be executed independently, in parallel, in time* $O(p \log N + t_s(N))$ *on a hypercube multiprocessor of size* N; $N = \max\{n, m\}$.

4. The multiple stabbing problem

In this section we present our first example of how hypercube cascading can be applied to solve goemetric problems. Consider the problem of determining all intersections of m lines with a simple polygonal path of length n. We will refer to this problem as the *multiple stabbing problem.* Using hypercube cascading we solve the multiple stabbing problem in

$$O\left(k \log \frac{N}{k} \log N + t_s(N)\right)$$

time on $O(N)$ processors ($O(1)$ memory per processor), where k is the maximum number of intersections between one of the m query lines and the polygonal path of length n, and $N = \max\{n, m\}$.

Our solution is built exclusively on an advanced data structure without the need for extensive new routing methods as would be required by a direct hypercube solution. It is also efficient: its time complexity differs by less than a factor of log N from the best known sequential solution for the single query line version of the problem [8].

Solving this stabbing problem (see Fig. 18) in $O(n)$ time for a given P and one line l is trivial. The problem becomes much more interesting if we have many query lines $l_1, l_2, \ldots, l_x$ and wish to successively test each of them for intersection with P as efficiently as possible. Chazelle and Guibas [8] present a sequential fractional cascading algorithm that allows to answer, with $O(n)$ space and $O(n \log n)$ preprocessing, one stabbing query in time $O(k \log(n/k))$, where k is the size of the output. We will first describe their data structure and algorithm and then show how, using our hypercube cascading methodology, their sequential algorithm can be converted into a hypercube algorithm that efficiently solves the multiple stabbing problem.

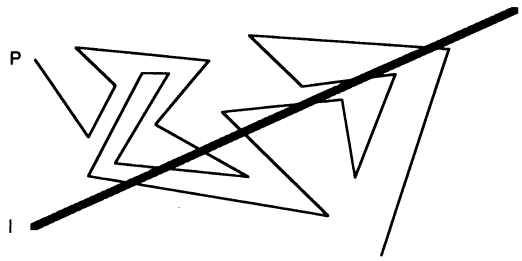


Fig. 18. Intersections of a line with a simple polygonal path of length n.

4.1. Review of Chazelle and Guibas' sequential stabbing algorithm

The central idea behind the algorithm in [8] is that a straight-line l intersects a polygonal line path P if and only if l intersects the convex hull, CH(P), of P. Chazelle and Guibas [8] build a data structure based on convex hulls represented by their slope sequences. The *slope-sequence* of a convex polygon C is the ordering of the edges of C in nondecreasing order of edge slopes. Let s and s' be the two slopes of a line l obtained by giving l its two possible orientations; if we know the position of s and s' within the slope-sequence of a convex polygon C, we can clearly determine whether C and l intersect in constant time.

Chazelle and Guibas [8] define a convex hull decomposition tree for a simple polygonal path P as follows. Let $F(P)$ and $S(P)$ denote the first and second halves of the path P of length n, respectively; that is, the subpaths of P consisting of the first $n/2$ and second $n/2$ edges. A convex hull decomposition tree, T, for a simple polygonal path P, is a binary tree with CH(P) assigned to its root, and its two children being the roots of convex hull decomposition trees representing $F(P)$ and $S(P)$, respectively. Convex hull edges that occur in several convex hulls (for several nodes) are stored only once at the highest node where they occur (see Fig. 19, bold edges). The edges stored at a node v form a subsequence of adjacent edges of the respective convex hull; we will refer to them as the *subhull* stored at v. Chazelle and Guibas show that, if P is simple, then CH($F(P)$) and CH($S(P)$) have at most two common tangents. Therefore, the resulting tree uses linear space.

The subhulls at the nodes in the tree are each stored as slope sequences; each slope sequence represents a catalog for fractional cascading. Having constructed a convex hull decomposition tree T for a simple polygonal path P, all intersections between P and a query line l can be reported by searching the tree T with query line l through a branch and bound type searching procedure. At each node, if the query line intersects the respective convex hull, then both subtrees have to be searched recursively; otherwise the search is bounded at this point. In [8] it is

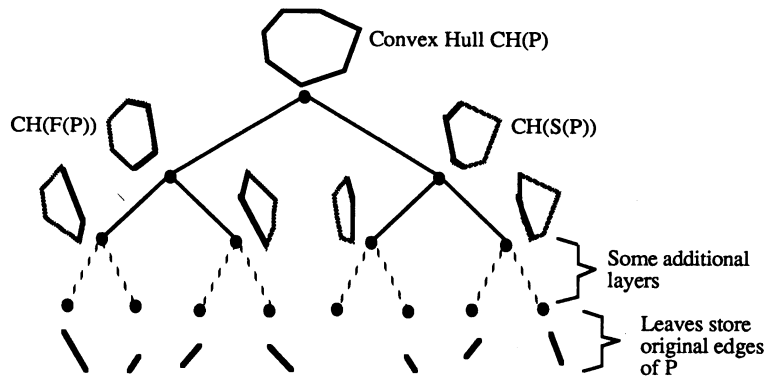


Fig. 19. A convex hull decomposition tree.

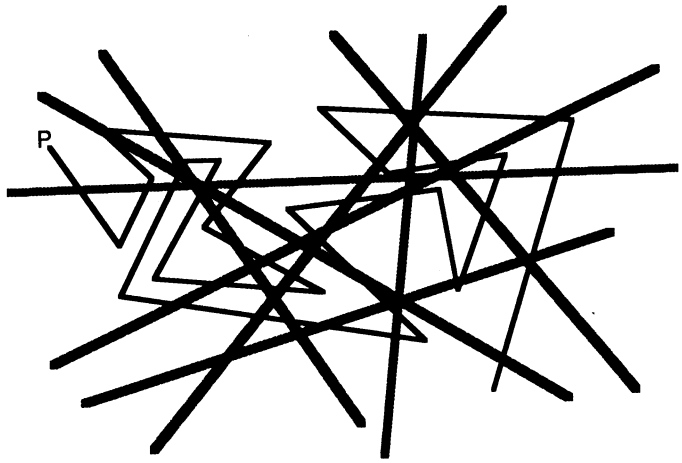


Fig. 20. The multiple stabbing problem.

shown how the intersection text can be performed on the subhull stored at the node, rather than the entire hull. This test is based on locating the slope of the query line l in the slope sequence of the edges of the subhull.

4.2. Hypercube cascading for the multiple stabbing problem

This section presents a hypercube algorithm for determining all intersections of m lines with a simple polygonal path of length n; see Fig. 20.

The parallel hypercube algorithm described in this section will solve the multiple stabbing problem by applying our hypercube cascading technique presented in Section 3 to Chazelle and Guibas' algorithm. In order to be able to apply hypercube cascading, we have to solve the following additional problems:

(1) Construct the convex hull decomposition tree in parallel, on a hypercube.

(2) Define a query routing scheme for each query line for reporting its intersections with the polygonal path P. Note that, Chazelle and Guibas' branch and bound scheme is not suitable for hypercube cascading.

We observe that Chazelle and Guibas' convex hull decomposition tree is both monotone and of fixed degree, and therefore meets the requirements of hypercube cascading.

Algorithm Construct Parallel Convex Hull Decomposition Tree:
Input: The n edges of a simple polygonal path sorted by slope.
Output: A convex hull decomposition tree represented as a catalog graph meeting the requirements of hypercube cascading as specified in Section 3.
Method: First, construct a complete binary tree with n leaves and empty catalogs attached to all its nodes. Then, assign to each leaf a query consisting of a line segment, such that the line segments are sorted by slope. These queries will be

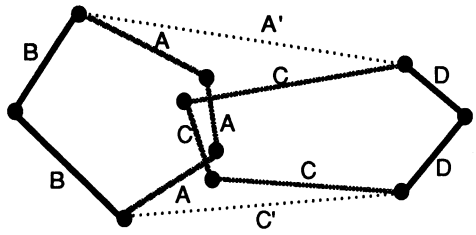


Fig. 21. Identifying the merge points of two convex hulls.

referred to as *line queries*. In $O(\log n)$ phases, the line queries are then moved towards the root of the tree using hypercube cascading. In each phase a node may be visited by line queries from its left and right children. The set of line queries from any child form a convex hull (queries at leaves form degenerate convex hulls) sorted by slope. Upon arriving at a node, the groups of line queries from the left and right children are merged to form the convex hull of the union of both sets. The $O(\log n)$ merging of two convex hulls can be implemented by a bitonic merge operation [4]. This process creates six types A, B, C, D, A', C' of line queries as shown in Fig. 21.

Type A line queries are returned to the node's left child to form its catalog elements (sorted by slope). Type C line queries are returned to the node's right child to form its catalog elements (sorted by slope). The remaining queries are compressed into a single sequence of lines sorted by slope and sent to the node's parent. If the node is the root node, the merged sequence is stored as its catalog. Since the maximum path length of any query in this algorithm is $O(\log n)$ it follows from Theorem 1 that the entire algorithm can be completed in time $O(\log^2 n)$ using $O(n)$ processors.

Theorem 2. *A convex hull decomposition tree for a simple polygonal path of length n can be constructed on a hypercube multiprocessor of size n in time* $O(\log^2 n)$.

What remains to be shown is that Chazelle and Guibas' branch and bound algorithm for reporting the intersections of one segment with P can be converted into a scheme for hypercube cascading. We could execute each line query by starting a query at the root of the tree and simulating Chazelle and Guibas' branch and bound approach by duplicating the query at any branch. Such an approach may however create, in the worst case, a total of $O(nm)$ queries. Hence rather than duplicating queries, our approach will be to have each line segment represented by a single query which will report all its intersections with the simple polygonal path.

Let $T'(l)$ be the subtree of the convex hull decomposition tree T that must be searched by Chazelle and Guibas' algorithm to report all intersections of one query line l. In [8] it is shown that the size $|T'(l)|$ of $T'(l)$ is $O(k \log (n/k))$,

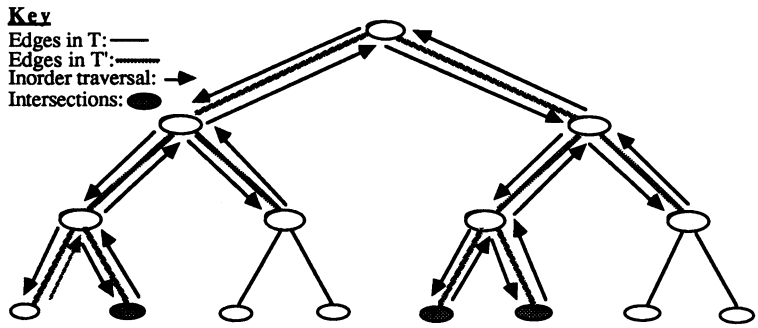


Fig. 22. An order traversal of the subtree T' of T.

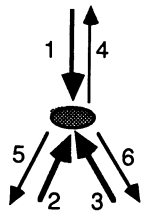
where k is the number of intersections of l with P. Our approach will be to create for each query line l one single query which traverses $T'(l)$ in inorder (see Fig. 22) and reports all intersections. The length of the path is $O(k \log(n/k))$. Hence, by applying Theorem 1, we obtain a

$$O\left(k \log \frac{N}{k} \log N + t_s(N)\right)$$

time algorithm for answering m such queries. The details of this routing scheme are described below.

Algorithm Report Intersections:
Input: A convex hull decomposition tree T for a simple polygonal path P, and m query lines.
Output: For each query line l, each edge of P intersected by l is reported.
Method: For each query line l, generate one query $q(l)$ that is routed through T (using hypercube cascading). All queries initially visit the root of T, and then each $q(l)$ traverses in inorder the subtree $T'(l)$ and reports all intersections. When a query $q(l)$ visits a node v, the slope of l is located in the attached catalog, and the intersection test with the convex polygon associated with v is done in exactly the same way as in the sequential algorithm (see Section 4.1). What remains to be shown is how a query $q(l)$ can decide locally which node to visit next, such that the resulting path is the inorder traversal of $T'(l)$. It is easy to see that this decision can be made by one processor in constant time based on the node previously visited (incoming direction) and the result of the intersection test. All possible combinations are listed in Fig. 23.

Theorem 3. *Given a simple polygonal path P of length n and a set of m 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*



Incoming Direction	No intersection	Intersection
1	4	5
2	N/A	6
3	N/A	4

Fig. 23. Local computation of the next node in the inorder traversal of $T'(l)$.

time

$$O\left(k \log \frac{N}{k} \log N + t_s(N)\right);$$

$N = \max\{n, m\}$.

Note that we assume that at the end of each phase of hypercube cascading, every processor can report a result without having to store it. Otherwise, after each phase, the reported results would need to be concentrated in order to obtain an even data distribution, and the number of processors and time complxity would increase to

$$O(N+M) \quad \text{and} \quad O\left(k \log \frac{N}{k} \log(N+M) + t_s(N)\right), \text{ respectively,}$$

where M denotes the total output size for all queries.

5. Multiple slanted range search

In this section we give another example of how hypercube cascading can be used to generate efficient parallel computational geometry algorithms for hypercube multiprocessors.

Consider a set S of n points in the Euclidean plane. An *aligned trapezoid* is a trapezoid with one side b on the x-axis and the two adjacent sides parallel to the y-axis. The *slanted range search problem* consist of reporting all points contained in the aligned trapezoid [8]; see Fig. 24. The multiple slanted range search problem consists of answering, in parallel, m slanted range search queries on one set S of n points.

Chazelle and Guibas [8] present a sequential fractional cascading algorithm that allows to answer, with $O(n)$ space and $O(n \log n)$ preprocessing, one slanted range query in time $O(k + \log n)$, where k is the size of the output. Their algorithm is based on a *tree of convex hulls*, TC(S), representing S as follows: With the root of TC(S), we associate the lower hull LH(S) of S; i.e., the lower portion of the convex hull of S. Consider the left half L and right half R,

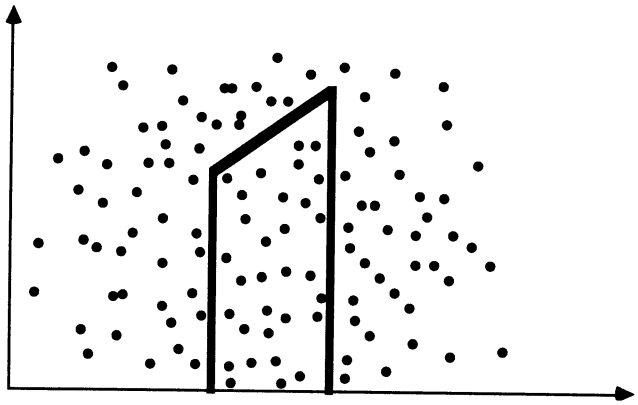


Fig. 24. Slanted range search problem.

respectively, of the set S minus the points of $L(S)$. The left and right subtrees recursively represent L and R, respectively. See Fig. 25 for an illustration.

In [8], one slanted range search query is again executed by a branch-and-bound type search on TC(S), 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 associated with that node.

We solve the slanted range search problem by applying our hypercube cascading technique presented in Section 3 to Chazelle and Guibas' algorithm. We have to solve the following additional problems: Constructing the tree of convex hulls in parallel, on a hypercube, and defining a query routing scheme for each query (that is consistent with the requirements of Section 3).

We observe that the convex hull of n points can be 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 [4]. Hence, the tree TC(S) together with catalogs representing the lower hull edges associated with each node (sorted by their slopes) can be constructed in time $O(\log^3 n)$. Note that, since TC(s) is a binary

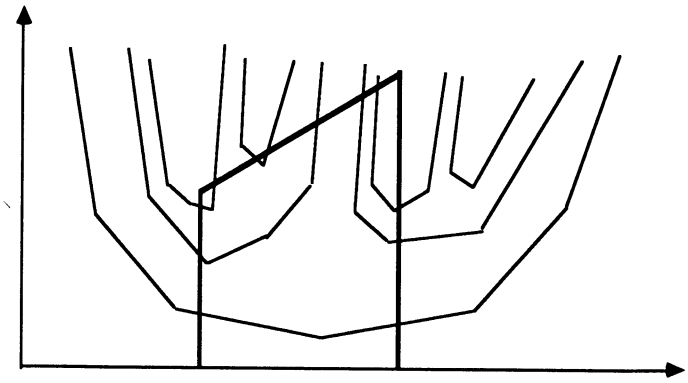


Fig. 25. Tree of convex hulls.

tree, it is a monotone graph of fixed degree and therefore meets the requirement of Theorem 1.

In order to answer, in parallel, a set of m slanted range queries on S, we again replace for each query the branch and bound scheme in [8] by an inorder traversal of length $O(k \log (n/k))$. The approach is as in Section 4. The decision each query has to make in order to find the next node in the inorder traversal of the subtree searched by the sequential branch and bound scheme, can be made by one processor in O(1) time based on what node was previously visited and the result of the intersection test. Applying Theorem 1, we obtain the following.

Theorem 4. *Given a set of n points in the Euclidean plane then, with a preprocessing of $O(\log^3 n)$, m slanted range search queries (with a maximum of k results per query) can be solved on a hypercube of size $N = \max\{mn, n\}$ in time $O(k \log N + t_s(N))$.*

6. Conclusion

Fractional cascading is a powerful and widely used technique for designing efficient sequential computational geometry algorithms. In this paper we presented an efficient algorithm for implementing parallel fractional cascading on a hypercube multiprocessor, thereby providing a new tool for parallel computational geometry on hypercubes. As example applications, we presented hypercube algorithms for the multiple stabbing and the multiple slanted range search problems.

Appendix A: Standard hypercube routing procedures

The algorithms in Section 3 of this paper used slightly generalized versions of 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 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* [16].

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

Concentrate({[$S_1(i)$,$D_1(i)$], . . . , [$S_z(i)$,$D_z(i)$]},Cond(i)): This operation includes an initial Rank(R(i), Cond(i)) operation. Then for each PE(i) with Cond(i) = *true*, the source registers S_1(i), . . . S_z(i) are copied to PER(R(i)) where the values

are stored in registers $D_1(i), \ldots, D_z(i)$ respectively: $z = O(1)$. The time complexity of this operation is also O(log N) [16].

Route($\{[S_1(i)D_1(i)], \ldots, [S_z(i),D_z(i)]\}$,Dest($i$),Cond($i$)): Every processor PE(i) has $2z = O(1)$ data registers $S_1(i), \ldots, S_z(i)$ and $D_1(i), \ldots, D_z(i)$, a destination register Dest(i), and a boolean condition register Cond(i). It is assumed that the destinations Dest(i) are monotone; i.e., if $i < j$ then Dest(i) < Dest(j). The operation routes, for every processor PE(i) with Cond(i) = true, all source registers $S_1(i), \ldots, S_z(i)$ to processor PE(Dest(i)) here the values are stored in registers $D_1(i), \ldots, D_z(i)$ respectively. It can be implemented with an O(log N) time complexity by using a Concentrate operation followed by a Distribute operation described in [16].

RouteAndCopy($\{S_1(u)D_1(i)], \ldots, [S_z(i),D_z(i)]\}$,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 source registers $S_1(i), \ldots, S_z(i)$ to destination registers $D_1(i), \ldots, D_z(i)$ of processors PE(Dest(i − 1) + 1), . . . , PED(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 [16].

Reverse($\{[S_1(i),D_1(i)], \ldots, [S_z(i),D_z(i)]\}$,Start,End): This operation routes for every PE(i) with Start $\leq i \leq$ End, its source registers $S_1(i), \ldots, S_z(i)$, $z = O(1)$, to destination registers $D_1(i), \ldots, D_z(i)$ on 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 [16].

Merge($\{[S_1(i),D_1(i)], \ldots, [S_z(i),D_z(i)]\}$,Key($i$),Left,Peak,Right): This operation is the well known bitonic merge [4]. It converts in time O(log N) a bitonic sequence (with respect to register Key(i)) into a sorted sequence; it simultaneously permutes the source registers $S_1(i), \ldots, S_z(i)$ ($z = O(1)$) storing the results in the destinations registers $D_1(i), \ldots, D_z(i)$. 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($\{[S_1(i),D_1(i)], \ldots, [S_z(i),D_z(i)]\}$,Key($i$)): This operation refers to sorting with respect to Key(i); it simultaneously permutes the source registers $S_1(i), \ldots, S_z(i)$ ($z = O(1)$) storing the result in registers $D_1(i), \ldots, D_z(i)$ respectively. The time complexity, $t_s(N)$, of this operation is currently $O(\log N \log\log N)$ [10, 15].

IdentifyBlockTail(block(i),tail(i)): A block is a set of contiguous processors which share the same value in some register block(i). This operation identifies the last processor of each block defined by register block(i). An O(log N) time implementation is easily obtained by applying the above Route operation.

References

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

[2] A. Aggarwal, B. Chazell, L. Guibas, C. O'Dunlaing and C. Yap, Parallel computational geometry, Algorithmica 3 (1988) 293–327.

[3] M.J. Atallah and M.T. Goodrich, Efficient plane sweeping in parallel, in: Proc. ACM Symp. Computational Geometry (1986) 216–225.

[4] K.E. Batcher, Sorting networks and their applications, in: Proc. AFIPS Spring Joint Computer Conference (1968) 307–314.

[5] A. Bonopera, V. Ho, A. Rau-Chaplin and D. Yeong, Connection Machine Implementations of Hypercube operations and Data Structuring Techniques, Technical Note 90–1, Dept. of Computer Science, Carleton University.

[6] J.L. Bentley and D. Wood, An optimal worst case algorithm for reporting intersections of rectangles, IEEE Transactions on Computers 29 (1980) 571–576.

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

[8] B. Chazelle and L.J. Guibas, Fractional cascading: II, Applications, Algorithmica 1 (1986) 163–192.

[9] B. Chazelle, Filtering search: a new approach to query answering, SIAM J. Comp. 15 (1986) 703–724.

[10] R. Cypher and C.G. Plaxton, Deterministic sorting in nearly logarithmic time on the hypercube and related computers, 1990 ACM Symposium on Theory of Computing.

[11] 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) 1084–1093.

[12] N. Dadoun and D.G. Kirkpatrick, Parallel processing for efficient subdivision search, in: Proc. ACM Symp. on Computational Geometry (1987) 205–214.

[13] F. Dehne and A. Rau-Chaplin, Implementing data structures on a hypercube multiprocessor and applications in parallel computational geometry, J. Parallel Distributed Comput. 8 (1990) 367–375.

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

[15] F.T. Leighton, Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes (Morgan Kaufmann, Los Altos, CA, 1991) Section 3.5.3.

[16] D. Nassimi and Sahni, Data broadcasting in SIMD computers, IEEE Trans. Comput. 30 (1981) 101–106.

[17] F.P. Preparata and M.I. Shamos, Computational Geometry—An Introduction (Springer, Berlin, 1985).