Journal of Parallel and Distributed Computing **57**, 224–235 (1999)
Article ID jpdc.1998.1527, available online at http://www.idealibrary.com on IDEAL®

Coarse-Grained Parallel Geometric Search[1]

Albert Chan,* Frank Dehne,*,[2] and Andrew Rau-Chaplin†

*School of Computer Science, Carleton University, Ottawa, Canada K1S 5B6;
†Faculty of Computer Science, Dalhousie University, Halifax, Canada B3J 2X4
E-mail: {achan, dehne}@scs.carleton.ca, arc@cs.dal.ca

Received August 28, 1998; revised November 19, 1998; accepted December 18, 1998

We present a parallel algorithm for solving the *next element search problem* on a set of line segments, using a BSP-like model referred to as the *coarse grained multicomputer* (CGM). The algorithm requires $O(1)$ communication rounds (h-relations with $h = O(n/p)$), $O((n/p) \log n)$ local computation, and $O((n/p) \log p)$ memory per processor, assuming $n/p \geqslant p$. Our result implies solutions to the point location, trapezoidal decomposition, and polygon triangulation problems. A simplified version for axis-parallel segments requires only $O(n/p)$ memory per processor, and we discuss an implementation of this version. As in a previous paper by Develliers and Fabri (*Int. J. Comput. Geom. Appl.* **6** (1996), 487–506), our algorithm is based on a distributed implementation of segment trees which are of size $O(n \log n)$. This paper improves on *op. cit.* in several ways: (1) It studies the more general next element search problem which also solves, e.g., planar point location. (2) The algorithms require only $O((n/p) \log n)$ local computation instead of $O(\log p * (n/p) \log n)$. (3) The algorithms require only $O((n/p) \log p)$ local memory instead of $O((n/p) \log n)$. © 1999 Academic Press

Key Words: BSP; coarse-grained multicomputer; next element search; planar subdivision search; scalable parallel algorithms; segment tree; simple polygon triangulation; trapezoidal map.

1. INTRODUCTION

The next element search problem is a well-known problem in computational geometry and has many applications [1]. Given a set of n nonintersecting line segments $s_1, ..., s_n$ and a direction D_{next} (without loss of generality we can assume that D_{next} is the direction of the positive Y-axis), the next element search problem consists of finding for each query point q_i of a set of m query points $q_1, ..., q_m$ the line segment s_{j_i} first intersected by the ray starting at q_i in direction D_{next} ($m = O(n)$); see Fig. 1. A sequential solution requires $O(n \log n)$ time and $O(n)$ space [19].

[1] Research partially supported by the Natural Sciences and Engineering Research Council of Canada.
[2] Corresponding author.

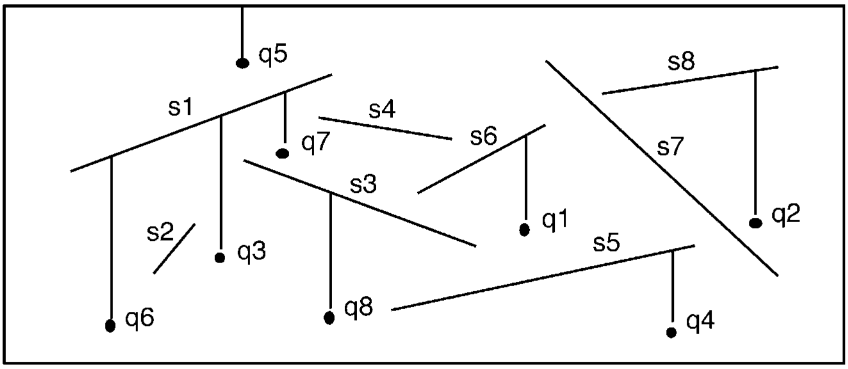

FIG. 1. Illustration of next element search.

In this paper, we present a parallel algorithm for solving the next element search problem on a coarse-grained multicomputer (CGM). Consult Section 2 for a discussion of the model. The algorithm requires $O(1)$ communication rounds, $O((n/p) \log n)$ local computation, and needs $O((n/p) \log p)$ memory per processor, assuming $n/p \geqslant p$. A simplified version for axis-parallel segments requires only $O(n/p)$ memory per processor.

The next element search algorithm presented here implies immediate solutions for the point location, trapezoidal decomposition and triangulation problems.

As in a previous paper by Develliers and Fabri [13], our algorithm is based on a distributed implementation of segment trees which are of size $O(n \log n)$. This paper improves on [13] in several ways:

- It studies the more general next element search problem which also solves, e.g., planar point location.
- The algorithms require only $O((n/p) \log n)$ local computation instead of $O(\log p * (n/p) \log n)$.
- The algorithms require only $O((n/p) \log p)$ local memory instead of $O((n/p) \log n)$.

Note that [13] also assumes that $n/p \geqslant p$. It is also interesting to observe that the method presented in this paper needs less than $O(n \log n)$ total memory (over all processors) because it eliminates some of the segment tree catalogs (while maintaining the same next element search functionality).

The organization of this paper is as follows: Sections 2 and 3 define the coarse-grained multicomputer model and segment tree data structure, respectively. The algorithm is presented in Sections 4 and 5. In Section 6 we outline a simplified version for axis-parallel segments and discuss an implementation of this version. Section 7 concludes the paper and outlines some important applications.

2. THE COARSE-GRAINED MULTICOMPUTER MODEL

Speedup results for theoretical PRAM algorithms do not necessarily match the speedups observed on real machines [2, 20]. Given sufficient slackness in the number of processors, Valiant's BSP approach [22] simulates PRAM algorithms optimally on distributed memory parallel systems. Valiant points out, however, that one may want to design algorithms that utilize local computations and minimize global operations [21, 22]. The BSP approach requires that g ($=$local computation speed/router bandwidth) is low, or fixed, even for increasing number of processors. Gerbessiotis and Valiant [15] describe circumstances where PRAM simulations cannot be performed efficiently, among others if the factor g is high. Unfortunately, this is true for most currently available multiprocessors. The algorithms presented here consider this case for the next element search problem. Furthermore, as pointed out in [22], the cost of a message also contains a constant overhead cost s. The value of s can be fairly large and the total message overhead cost can have a considerable impact on the speedup observed (see, e.g., [8]). We are therefore using a variation of the BSP model, referred to as *coarse-grained multicomputer* (CGM).

A CGM is comprised of a set of p processors $P_1, ..., P_p$ with $O(N/p)$ local memory per processor and an arbitrary communication network (or shared memory). The term "coarse-grained" refers to the fact that we assume that the size $O(N/p)$ of each local memory is "considerably larger" than $O(1)$. Our definition of "considerably larger" will be that $N/p \geqslant p$. All algorithms consist of alternating local computation and global communication rounds. Each communication round consists of routing a single h-relation with $h = O(N/p)$; i.e., each processor sends $O(N/p)$ data and receives $O(N/p)$ data. We require that all information sent from a given processor to another processor in one communication round is packed into one message. In the BSP model, a computation/communication round is equivalent to a superstep with $L = (N/p)\, g$ (plus the above "packing" requirement).

Finding an optimal algorithm in the coarse-grained multicomputer model is equivalent to minimizing the number of communication rounds as well as the total local computation time. This considers all parameters discussed above that are affecting the final observed speedup, and it requires no assumption on g. Furthermore, it has been shown that minimizing the number of supersteps also leads to improved portability across different parallel architectures [11, 21, 22]. The above model has been used (explicitly or implicitly) in parallel algorithm design for various problems ([4, 7–10, 12, 14, 17]) and has shown very good practical timing results.

We now list some operations required by our algorithms. Each of these operations reduces to $O(1)$ communication rounds for $N/p \geqslant p$:

- *Global sort.* Sort $O(N)$ data items stored on a CGM, N/p data items per processor, with respect to the CGM's processor numbering. As shown in [16], for $N/p \geqslant p$ it is possible to sort in $O(1)$ communication rounds with $O(N/p)$ memory per processor and $O((N/p) \log N)$ local computation.

- *Global integer sort.* Sort $O(N)$ integers in the range $1, ..., N^c$ for fixed constant c stored on a CGM, N/p data items per processor, with respect to the CGM's processor numbering. The sort algorithm in [16] is based on Cole's merge sort [6]. The $O((N/p) \log N)$ local computation in [16] is due to a constant number of local sorts. Hence, by applying radix sort for the integer case, we obtain $O(N/p)$ local computation without increasing the number of communication rounds. For practical implementations, a much simpler CGM integer sorting algorithm with 9 communication rounds, $O(N/p)$ memory per processor and $O(N/p)$ local computation can be found in [5].

- *All-to-all broadcast.* Every processor sends one message to all other processors [8]. This operation requires $O((N/p))$ local computation.
- *Personalized all-to-all broadcast.* Every processor (in parallel) sends a different message to every other processor [8]. This operation requires $O((N/p))$ local computation.
- *Partial sum (scan).* Every processor stores one value, and all processors compute the partial sums of these values with respect to some associative operator [8] ($O((N/p))$ local computation).

3. SEGMENT TREE DEFINITION

A well-known method for solving the next element search problem is to apply a segment tree [3, 18, 19]. Let $s_i^{(x)}[q_i^{(x)}]$ be the projection of the line segment s_i (query point q_i, respectively) onto the x-axis, and let $(x_1, x_2, ..., x_{2n})$ be the sorted sequence of the projections of the $2n$ endpoints of $s_1, ..., s_n$ onto the x-axis. The segment tree $T(S) = (V_s, E_s)$ is a complete binary tree with leaves $x_1, x_2, ..., x_{2n}$. For every node v of $T(S)$ an interval $xrange(v)$ is defined as

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

With every node v of the segment tree $T(S)$ there is associated a catalog $C(v) \subseteq S$ defined as

- $C(v) = \{s \in S \mid xrange(v) \subseteq s(x) \text{ and not } (xrange(\text{parent of } v) \subseteq s(x))\}$.

Note that each line segment can occur in $O(\log n)$ catalogs. The size of the segment tree $T(S)$, denoted $|T(S)|$, is equal to the number of nodes and edges of $T(S)$ plus the total size of all catalogs. Therefore $|T(S)| = O(n \log n)$. Hence, storing the segment tree with all of its catalogs requires $O(n \log n)$ space. Also note that the sum of the lengths of all catalogs of all nodes with the same level (height) is $O(n)$ [18]. For the remainder, define $xrange(T(S)) = xrange(r)$, where r is the root of $T(S)$. Also define $xrange(s)$ and $xrange(q)$ to be $s_i^{(x)}$ and $q_i^{(x)}$, respectively.

4. PARALLEL SEGMENT TREE CONSTRUCTION

In this section we will show how to construct a distributed representation of a segment tree $T(S)$, called a *parallel segment tree*, for a set of n line segments on a CGM such that the resulting data structure can be efficiently used to process next element search queries in parallel. The approach will be to partition the segment tree (without associated catalogs) into substructures of size $O(n/p)$ such that no processor stores more than $O(1)$ such substructures; see Fig. 2. The catalogs for

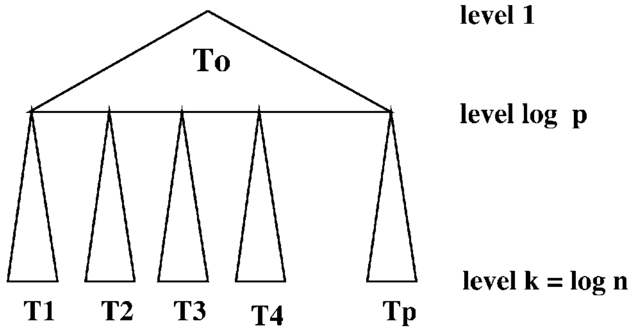

FIG. 2. Decomposition of the segment tree.

nodes in tree T_0 will be partitioned into lists of size $(n \log p)/p$ and distributed such that no processor stores more than $O(1)$ such lists; see Fig. 3. The catalogs for nodes in subtrees $T_1 \cdots T_p$ will not be built explicitly. Instead, the algorithm will use standard sequential plane sweep (using a standard binary search tree) [19]. We first describe our distributed segment tree representation and then give an algorithm that efficiently constructs such a segment tree on a CGM.

A parallel segment tree $T(S)$ (without catalogs) is a complete tree with n leaves which consists of $\log n$ levels, where the level of a node v is defined recursively as follows: $level(v) = 1$, if v is the root, and $level(v) = level(parent(v)) + 1$ otherwise. Let $rank(v)$ denote the rank of the vertex v in the left to right ordering of the vertices with the same level as v. For $v \in V$, let $subtree(v) \in 1 \cdots p$ be defined as follows: $subtree(v) = p$, if $1 \leqslant level(v) \leqslant \log p - 1$ and $subtree(v) = rank(v')$ otherwise, where v' is the ancestor of v such that $level(v') = \log p$. Let $V_i = \{v \in V \mid subtree(v) = i\}$. Let T_i denote the subtree of T with vertex set V_i and edge set $E_i = \{(v, v') \in E \mid subtree(v) = subtree(v') = i\}$. Let $S(T_i)$ be the set of segments of S with an endpoint in $xrange(T_i)$.

In our distributed representation of a segment tree $T(S)$, processor P_i $(1 \leqslant i \leqslant p)$ stores $S(T_i)$, the tree T_0 without catalogs but with the values $xrange(v)$ for each $v \in V_0$, and the list l_i which is a portion, or all, of the catalog of a node v of T_0.

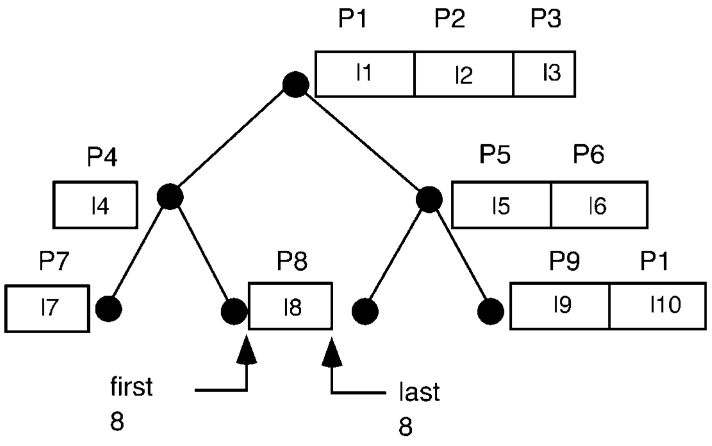

FIG. 3. The tree T_0 with catalogs partitioned into lists l_i, where list l_i is of size $|l_i| \leqslant (n \log p)/p$.

Observation 1. If a line segment contains $xrange(T_i)$ then it is not contained in any catalog of T_i (except for possibly the root).

Observation 2. $S(T_i)$ is of size $O(n/p)$.

A consequence of Observation is that each subset $S(T_i)$, $1 \leqslant i \leqslant p$, can be stored in the memory of a single processor. The tree T_0 consists of $O(p)$ nodes and catalogs whose combined size is $O(n \log p)$. Therefore T_0 is too big to be stored on a single processor. Instead, each processor will store a copy of T_0 (without catalogs) and a list l_i which is a portion, or all, of the catalog of a node v of T_0. Let L denote the list formed by concatenating the catalogs associated with nodes of T_0, where catalogs are ordered by level and then rank in level and all catalogs are padded to be of a length evenly divisible by $(n \log p)/p$. The list l_i consisting of elements $in/p \cdots ((i+1)\,n)/p$ from L will be stored on processor P_i; see Fig. 3.

Observation 3. Since T_0 has height $\log p$ and a line segment can appear in at most two catalogs of T_0 at the same level, the total size of list L is $O(n \log p)$.

Definition 1. Given two line segments s_1 and s_2. We define a relation "$\leqslant$" as follows. If $xrange(s_1) \cap xrange(s_2) = \varnothing$ then $s_1 \leqslant s_2$ if and only if the y-coordinate of the left endpoint of s_1 is smaller or equal than the y-coordinate of the left endpoint of s_2. If $xrange(s_1) \cap xrange(s_2) \neq \varnothing$ then $s_1 \leqslant s_2$ if and only if s_1 is below s_2.

Algorithm 1 (Parallel segment tree construction). *Input*: Processor P_i $(1 \leqslant i \leqslant p)$ stores a subset S_i of n/p elements of S. *Output*: Processor P_i $(1 \leqslant i \leqslant p)$ stores $S(T_i)$, the tree T_0 without catalogs but with the values $xrange(v)$ for each $v \in V_0$, and the list l_i which is a portion, or all, of the catalog of a node v of T_0.

(0) Sort S globally with respect to the relation "$\leqslant$" in Definition 1. For each $s \in S$ calculate its rank $y(s)$ in S with respect to this ordering and store it with s.

(1) Create for each $s \in S$ two copies, one for each endpoint, and sort the set by x-coordinate such that each processor P_i contains a subset S_i of size $O(n/p)$. Now, processor P_i stores $S(T_i)$ and $xrange(T_i)$. From Observation 2, $|S(T_i)| = O(n/p)$.

(2) Use an all-to-all broadcast to distribute all $xrange(T_i)$ $(1 \leqslant i \leqslant p)$ to all processors. Each processor computes T_0 without any catalogs but with $xrange(v)$ values for all $v \in V_0$.

(3) Processor P_i computes the catalogs of T_0 with respect to S_i only. We refer to this reduced version of T_0 as $T_{0,i}$. Note that $|T_{0,i}| = O((n/p) \log p)$.

(4) Assume that all nodes of T_0 have a unique index. Consider a line segment s in the catalog of the node v in T_0 with index $j(s)$. For each such line segment s, define a key $\lambda(s)$ obtained by concatenating the bits of $j(s)$ and $y(s)$, in that order. Using a global integer sort, all line segments in the catalogs of all $T_{0,i}$, $1 \leqslant i \leqslant p$, are sorted with respect to key $\lambda(s)$ in such a way that no processor stores no two line segments with different j values. The latter can be achieved by using $2p$ virtual processors.

End of algorithm.

THEOREM 2. *Algorithm* 1 *constructs a parallel segment tree on a CGM using* $O((n/p)\log p)$ *memory per processor*, $O(1)$ *communication rounds, and* $O((n/p)\log n)$ *local computation.*

Proof. The memory bound follows from Observations 2 and 3. The algorithm uses a constant number of the basic communication operations of Section 2 and, hence, $O(1)$ communication rounds. We now prove that the algorithm requires $O((n/p)\log n)$ local computation. We observe that the local computation is dominated by the sorting steps. There are two types of sorting operations used: (1) global sort on $O(n/p)$ data per processor in Steps 0 and 1 and (2) global *integer* sort on $O((n/p)\log p)$ data per processor in Step 4. Since $O((n/p)\log n) + O((n/p)\log p) = O((n/p)\log n)$, the claim follows. ▮

5. PARALLEL QUERY PROCESSING

Given a segment tree $T(S)$ and a query point $q \in Q$, the next element of q in S can be determined by a simple search in $T(S)$ from the root of $T(S)$ to the leaf v whose *xrange* contains q (see, e.g., [19] for details).

Recall that, at the end of Algorithm 1, processor P_i ($1 \leqslant i \leqslant p$) stores $S(T_i)$, the tree T_0 without catalogs but with the values *xrange*(v) for each $v \in V_0$, and a list l_i which is a portion, or all, of the catalog of a node v of T_0. Let *first*$_i$ and *last*$_i$ refer to the first and last element of l_i, respectively (see Fig. 4).

The following algorithm uses the parallel segment tree to answer all queries in parallel. Each individual query is first "routed" through T_0 and then through the respective T_i. In T_0, the tree structure is used to schedule the computation. However, the catalog lookups are reduced to sequential next element search problems. For the subtrees T_i, a load balancing scheme is used to ensure equal distribution of work. In each T_i, all search processes are reduced to a single sequential next element search problem.

ALGORITHM 2 (Parallel query processing). *Input*: A parallel segment tree $T(S)$ as produced by Algorithm 1 and a set Q of n queries, where each processor P_i stores a subset Q_i of size n/p. *Output*: Each Processor P_i ($1 \leqslant i \leqslant p$) stores for each $q \in Q_i$ its next element $s \in S$.

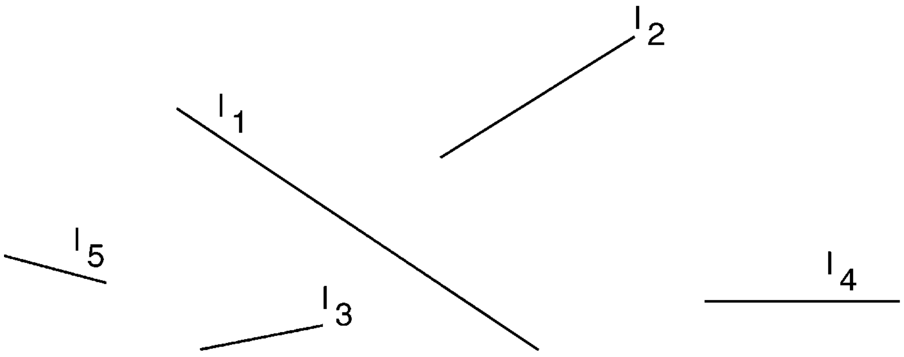

FIG. 4. A set of line segments with order $l_3 \leqslant l_4 \leqslant l_5 \leqslant l_1 \leqslant l_2$.

(1) Using an all-to-all broadcast, send all $first_i$ and $last_i$ to all processors P_i. Recall that every processor P_i ($1 \leqslant i \leqslant p$) stores T_0 without catalogs but with values $xrange(\cdot)$.

(2) Using the $O(p)$ line segments received in Step 1, processor P_i computes for each $q \in Q_i$ the sublists l_j ($1 \leqslant j \leqslant p$) that have to be searched in order to process q on T_0. The problem reduces to solving for each node v of T_0 a next element search problem for a certain number X_v of line segments and a certain number Y_v of queries. Note that, each such problem requires $O((X_v + Y_v) \log(X_v + Y_v))$ computation. Since $\sum_v X_v = p$ and $\sum_v Y_v = n/p$, it follows that the total local time is $O((n \log n)/p)$.

(3) Using global sort and partial sum operations, determine for each sublist l_i the number of queries, $g(l_i)$, that have to be located in l_i. Let $k(l_i) = \lceil (g(l_i)\, p)/n \rceil$.

(4) Create $k(l_i)$ copies of l_i ($1 \leqslant i \leqslant p$). Note that, this requires $2p$ virtual processors. Broadcast the new addresses of the sublists l_i.

(5) Each processor P_i makes $\log p$ copies of its query set Q_i and routes the queries to the respective sublists using integer sort.

(6) The queries are processed on the sublists to which they were sent in step 5, and the $\log p$ results for each query are collected in a single processor by using global integer sort.

(7) Determine for each T_i the number, $a(T_i)$, of queries whose search path includes the root of T_i ($1 \leqslant i \leqslant p$). This can be computed by using global integer sort and partial sum operations. Let $b(T_i) = \lceil a(T_i)/(n/p) \rceil$.

(8) Create $b(T_i)$ copies of $S(T_i)$. Note that, this requires $2p$ virtual processors. Using integer sort, route n/p queries to each processor such that a processor storing $S(T_i)$ receives n/p queries whose search path contains the root of T_i.

(9) Each processor processes the queries for its subtree T_i ($1 \leqslant i \leqslant p$) by applying plane sweep [19].

(10) Combine the results of Step 9 with those obtained in Step 6, using integer sort (by query ID).

End of algorithm.

THEOREM 2. *Algorithm 2 solves the next element search problem for n line segments on a CGM with $O((n/p) \log p)$ memory per processor using $O(1)$ communication rounds and $O((n/p) \log n)$ local computation.*

Proof. The memory bound follows from Theorem 1. The algorithm uses a constant number of the basic communication operations of Section 2 and, hence, $O(1)$ communication rounds. We now prove that the algorithm requires $O((n/p) \log n)$ local computation. We observe that the local computation is dominated by sorting and sequential plane sweep. There are two types of sorting operations used: (1) global sort on $O(n/p)$ data per processor in Step 3 and (2) global *integer* sort on $O((n/p) \log p)$ data per processor in steps 5–8 and 10. Since sequential plane sweep requires $O((n/p)\log n))$ steps [19], and $O((n/p)\log n) + O((n/p)\log p) = O((n/p)\log n)$, the claim follows. ▮

6. A SIMPLIFIED ALGORITHM FOR AXIS-PARALLEL LINE SEGMENTS

If we limit the segments to be axis-parallel (i.e., they are all horizontal), we can reduce the space requirement to $O(n/p)$ per processor by applying the lower envelope algorithm presented in [8]. During the algorithm, queries will be handled like line segments of zero length.

ALGORITHM 3 (Next element search for axis-parallel line segments). *Input*: A set S of n axis-parallel line segments and a set Q of n queries, where each processor P_i stores n/p line segments and queries, respectively. *Output*: The next element in S for each query point $q \in Q$, where each processor stores n/p next element results.

(1) Sort $S \cup Q$ by increasing y-coordinate. Each processor P_i solves both the next element search problem and the lower envelope problem sequentially for the set $S_i \cup Q_i$ it received in the sort. Let Q' be the set of all queries whose next element has not yet been found, and let S' be the union of all lower envelopes.

(2) Sort $S' \cup Q'$ by the x-coordinate of the right endpoints and the x-coordinates of the queries, respectively.

(3) Let $l_1, ..., l_{p-1}$ be the vertical lines that separate the sorted segments in the p different processors. Perform an all-to-all broadcast, where processor P_i sends l_i to all other processors.

(4) Perform a personalized all-to-all broadcast, where processor P_i sends segment $s \in S'$ to processor P_j if and only if s intersects the vertical line l_j.

(5) Each processor P_i solves locally the next element search problem for its subset of Q' and the line segments of S' received in steps 2 and 4.

End of algorithm.

THEOREM 3. *Algorithm 3 solves the next element search problem for n axis-parallel line segments on a GCM with $O(n/p)$ memory per processor using $O(1)$ communication rounds and $O((n/p)\log n)$ local computation, assuming $n/p \geqslant p$.*

Proof. The correctness follows from the fact that for each $q \in Q_i$, its next element is either in S_i or in the lower envelope of an S_j with $j > i$. The algorithm uses a constant number of the basic communication operations of Section and duplicates no data. ▮

Algorithm 3 was implemented on an Intel iPSC/860 hypercube and tested for $p = 2$, 4, and 8 processors. For each value of p, we ran tests for $n = 100$, 200, 500, 1000, 2000, 5000, and 10,000. We used 10 sets of data on each combination of p and n. In five of the sets, the line segments were evenly distributed in a unit square and, in the other five, the line segments were evenly distributed in a unit circle. The

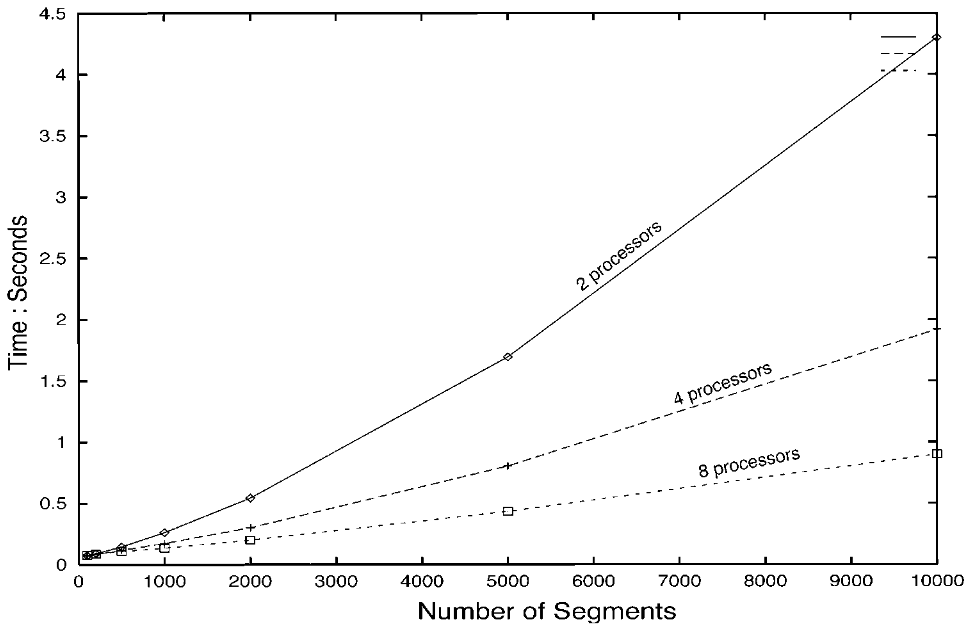

FIG. 5. Running times for algorithm 3 on an Intel iPSC/860.

average length of the segments was 1/10 unit length. The result is summarized in the Fig. 5. Observe the close to linear speedup obtained.

7. APPLICATIONS AND CONCLUSIONS

In this paper, we presented a BSP-like coarse-grained parallel algorithm for the next element search problem which requires $O(1)$ h-relations ($h = O(n/p)$), $O(n/p \log p)$ memory per processor and $O((n/p) \log n)$ local computation, assuming $n/p \geqslant p$. An important advantage of our model is that it gives a very good indication of the running time observed in an actual implementation. Our implementation on an Intel iPSC/860 hypercube obtained a very close to linear speedup for the axis-parallel case. As demonstrated in [7–9], coarse-grained parallel algorithms with $O(1)$ communication rounds are also portable across very different parallel platforms. Therefore, we expect that our algorithm presented here will also run well on other parallel machines.

Next element search can be used to solve many other geometric problems. Some of the more important examples include

1. *Planar subdivision search problem.* Given a plane graph $G = (V, E)$ with vertex coordinates, and a set of n query points q_i ($1 \leqslant i \leqslant n$), find for each query point q_i, the face of G containing q_i.

2. *Trapezoidal map problem.* Given a set of segments in the plane, decompose the plane into a set of trapezoids based on the arrangement of the segments.

3. *Triangulation problem for a simple polygon.* Partition the interior of a simple polygon into a set of triangles.

The above three problems can be reduced to $O(1)$ next element search problems (obvious for 1 and 2; see [23] for 3). Hence, Theorem 1 applies to these problems as well and we obtain

COROLLARY 1. *The planar subdivision search problem, trapezoidal map problem, and triangulation problem for a simple polygon can be solved on a CGM with* $O((n/p)\log p)$ *memory per processor in* $O(1)$ *communication rounds and* $O((n/p)\log n)$ *local computation, assuming* $n/p \geqslant p$.

ACKNOWLEDGMENTS

We thank Franco Preparata and Roberto Tamassia for their helpful discussion on the next element search problem for axis-parallel line segments. We thank the referees for their constructive comments.

REFERENCES

1. S. G. Akl and K. A. Lyons, "Parallel Computational Geometry," Prentice–Hall, Englewood Cliffs, NJ, 1996.
2. R. J. Anderson and L. Snyder, A comparison of shared and nonshared memory models of computation, *Proc. IEEE* **79**, 4 (1979), 480–487.
3. J. L. Bentley and D. Wood, An optimal worst case algorithm for reporting intersections of rectangles, *IEEE Trans. Comput.* **29**, 7 (1980), 571–576.
4. G. E. Blelloch, C. E. Leiserson, B. M. Maggs, and C. G. Plaxton, A comparison of sorting algorithms for the Connection Machine CM-2, *in* "Proc. ACM Symp. on Parallel Algorithms and Architectures, 1991," pp. 3–16.
5. A. Chan and F. Dehne, "A Note on Coarse Grained Parallel Integer Sorting," Technical Report TR-98-06, School of Computer Science, Carleton University, 1998. [http://www.scs.carleton.ca/].
6. R. Cole, Parallel merge sort, *SIAM J. Comput.* **17**, 4 (1988), 770–785.
7. F. Dehne, A. Fabri, and C. Kenyon, Scalable and architecture independent parallel geometric algorithms with high probability optimal time, *in* "Proc. 6th IEEE Symposium on Parallel and Distributed Processing, 1994," pp. 586–593.
8. F. Dehne, A. Fabri, and A. Rau-Chaplin, Scalable parallel computational geometry for coarse grained multicomputers, *in* "Proc. ACM Symp. Computational Geometry, 1993," pp. 298–307.
9. F. Dehne, X. Deng, P. Dymond, A. Fabri, and A. A. Kokhar, A randomized parallel 3D convex hull algorithm for coarse-grained parallel multicomputers, *in* "Proc. ACM Symp. on Parallel Algorithms and Architectures, 1995," pp. 27–33.
10. X. Deng, A convex hull algorithm for coarse grained multiprocessors, *in* "Proc. 5th International Symposium on Algorithms and Computation, 1994," pp. 634–640.
11. X. Deng and P. Dymond, Efficient routing and message bounds for optimal parallel algorithms, *in* "Proc. Int. Parallel Proc. Symp. (IPPS), 1995," pp. 556–563.
12. X. Deng and N. Gu, Good programming style on multiprocessors, *in* "Proc. IEEE Symposium on Parallel and Distributed Processing, 1994," pp. 538–543.
13. O. Devillers and A. Fabri, Scalable algorithms for bichromatic line segment intersection problems on coarse grained multicomputers, *Int. J. Comput. Geom. Appl.* **6** (1996), 487–506.
14. A. Ferreira, A. Rau-Chaplin, and S. Ubeda, Scalable 2D convex hull and triangulation for coarse grained multicomputers, *in* "Proc. 6th IEEE Symp. on Parallel and Distributed Processing, San Antonio, 1996," pp. 561–569.

15. A. V. Gerbessiotis and L. G. Valiant, Direct bulk-synchronous parallel algorithms, *in* “Proc. 3rd Scandinavian Workshop on Algorithm Theory,” Lecture Notes in Computer Science, Vol. 621 (1992), pp. 1–18.

16. M. T. Goodrich, Communication efficient parallel sorting, *in* “Proc. 28th Annual ACM Symp. on Theory of Computing (STOC’96), 1996,” pp. 247–256.

17. H. Li and K. C. Sevcik, Parallel sorting by overpartitioning, *in* “Proc. ACM Symp. on Parallel Algorithms and Architectures, 1994,” pp. 46–56.

18. K. Mehlhorn, “Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry,” Springer-Verlag, New York/Berlin, 1984.

19. F. P. Preparata and M. I. Shamos, “Computational Geometry—An Introduction,” Springer-Verlag, 1985.

20. L. Snyder, Type architectures, shared memory and the corollary of modest potential, *Annu. Rev. Comput. Sci.* **1** (1986), 289–317.

21. L. G. Valiant, A bridging model for parallel computation, *Commun. ACM* **33** (1990), 103–111.

22. L. G. Valiant, General purpose parallel architectures, “Handbook of Theoretical Computer Science” (J. van Leeuwen, Ed.), MIT Press/Elsevier, 1990, pp. 943–972.

23. C. K. Yap, Parallel triangulation of a polygon in two calls to the trapezoidal map, *Algorithmica* **3** (1988), 279–288.