# Coarse Grained Parallel Next Element Search

Albert Chan
School of Computer Science
Carleton University
Ottawa, Canada, K1S 5B6
*achan@scs.carleton.ca*

Frank Dehne
School of Computer Science
Carleton University
Ottawa, Canada, K1S 5B6
*dehne@scs.carleton.ca*

Andrew Rau-Chaplin
School of Computer Science
Technical University of Nova Scotia
Halifax, Canada, B3J 2X4
*arc@tuns.ca*

## Abstract

*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 n) storage per processor. 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) storage per processor, and we discuss an implementation of this version.*
*As in a previous paper by Develliers and Fabri[11], our algorithm is based on a distributed implementation of segment trees which are of size O(n log n). This paper improves on [11] which presented a CGM algorithm for the special case of trapzoidal decomposition only and requires O((n/p) * log p * log n) local computation.*

## 1. Introduction

The next element search problem is a well known problem in computational geometry with many applications[1]. Given a set of $n$ non-intersecting line segments $s_1, \ldots, s_n$ 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 $q_1, \ldots, q_m$ the line segment $s_j$ first intersected by the ray starting at $q_i$ in direction $D_{next}$ ($m=O(n)$); see Figure 1. A sequential solution requires $O(n \log n)$ time and $O(n)$ space[17].

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

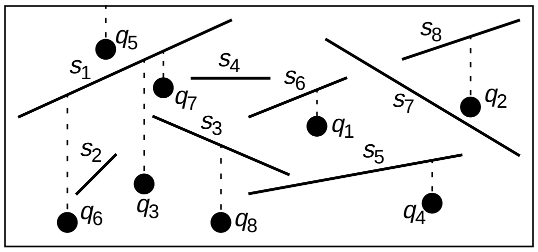

Figure 1: Illustration of Next Element Search

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

As in a previous paper by Develliers and Fabri [11], our algorithm is based on a distributed implementation of segment trees which are of size $O(n \log n)$. This paper improves on [11] which presented a CGM algorithm for the special case of trapzoidal decomposition only and requires $O((n/p) * \log p * \log n)$ local computation.

The organization of this paper is as follows: Section 2 and Section 3 define the coarse grained multicomputer model and segment tree, respectively. The algorithm is presented in Section 4 and Section 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][8]. Given sufficient slackness in the number of processors, Valiant's BSP approach [20] 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 [19][20]. The BSP approach requires that $g$ (= local computation speed / router bandwidth) is low, or fixed, even for increasing number of processors. Gerbessiotis and Valiant [13] 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 [20], 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. [6]).

We are therefore using a variation of the BSP model, referred to as *Coarse Grained Multicomputer,* CGM. It is comprised of a set of $p$ processors $P_1, \ldots, P_p$ with $O(m/p)$ 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 $O(m/p)$ of each local memory is "considerably larger" than $O(1)$. Our definition of "considerably larger" will be that $m/p \geq p$.

All algorithms consist of alternating local computation and global communication rounds. Each communication round consists of routing a single $h$-relation with $h{=}O(m/p)$, i.e. each processor sends $O(m/p)$ data and receives $O(m/p)$ 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 $L = (m/p)g$ (plus the above "packing" and "coarse grained" 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 [9][19][20]. The above model has been used (explicitly or implicitly) in parallel algorithm design for various problems ([4][5][6][7][8][10][12][15]) and shown very good practical timing results.

We now list the basic operations required by our algorithms. Each of these operations requires $O(1)$ communication rounds, $O((n/p) \log n)$ local computation, and requires $n/p \geq p$.

**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[14].

**All-to-all broadcast**: Every processor sends one message to all other processors[6].

**Personalized all-to-all broadcast**: Every processor (in parallel) sends a different message to every other processor[6].

**Partial sum (Scan)**: Every processor stores one value, and all processors compute the partial sums of these values with respect to some associative operator[6].

## 3. Segment tree definition

A well known method for solving the next element search problem is to apply a plane sweep in direction $D_{next}$ using a segment tree [3][16][17]. 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 $(x_1, x_2, \ldots, x_{2n})$ be the sorted sequence of the projections of the $2n$ endpoints of $s_1, \ldots, s_n$ onto the $x$-axis. The segment tree $T(S) = (V_s, E_s)$ is a complete binary tree with leaves $x_1, x_2, \ldots, x_{2n}$. For every node $v$ of $T(S)$ an interval $xrange(v)$ is defined as follows:

- if $v$ is a leaf $x_i$, then $xrange(v) = [x_i, x_{i+1})$, assuming that $[x_{2n}, x_{2n+1}) = [x_{2n}, x_{2n}]$.
- 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 $C(v) \in S$ defined as follows:

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

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 $N = O(n \log n)$ space. Also note that the sum of the lengths of all catalogs of all nodes with the same level (height) is $O(n)$[16]. 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 $s_i^{(x)}[qi^{(x)}]$ 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 Figure 2. The catalogs for nodes in

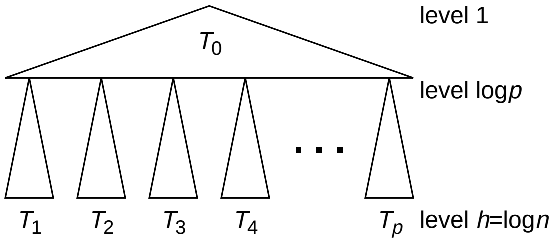

Figure 2: Decomposition of the Segment Tree

subtrees $T_1$, …, $T_p$ will be stored with their associated subtrees, while the catalogs for nodes in tree $T_0$ will be partitioned, when necessary, into lists of size $((n \log p) / p)$ and distributed such that no processor stores more than $O(1)$ such lists; see Figure 3. We first describe our

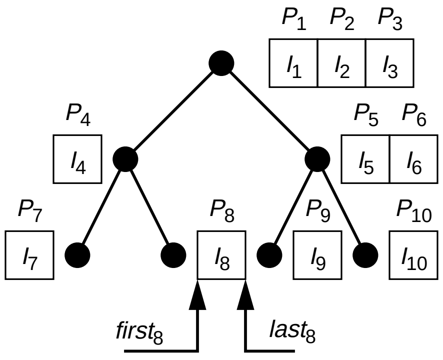

Figure 3: The tree $T_0$ with catalogs partitioned into lists $l_i$, where list $l_i$ is of size $|l_i| \le n/p$

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 $k = \log n$ 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 level for $v$. For $v \in V$, let $subtree(v) \in 1...p$ be defined as follows: $subtree(v) = p$, if $1 \le level(v) \le \log p-1$ 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 $S$ whose endpoint is in $xrange(T_i)$.

In our distributed representation of a segment tree $T(S)$, every subtree $T_i$ $(1 \le i \le p)$ will be stored on processor $P_i$. Each processor $P_i$ will also store the catalogs associated with nodes in $V_i$.

**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 2 is that each subset $S(T_i)$ $(1 \le i \le p)$ with its catalogs 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 $in/p$, …, $(i+1)n/p$ from $L$ will be stored on processor $P_i$; see Figure 3.

**Observation 3** *Since $T_0$ has height log p and a line segment can appear in at most 2 catalogs of $T_0$ at the same level, the total size of list L is $O(n \log p)$.*

**Algorithm 1: "Parallel Segment Tree Construction".** *Input*: Processor $P_i$ $(1 \le i \le p)$ stores a subset $S_i$ of $n/p$ elements of $S$. *Output*: Processor $P_i$ $(1 \le i \le p)$ 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 $T_0$.

(1) Create for each $s \in S$ two copies, one for each endpoint. Refer to the new set as $S'$. Sort $S'$ 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 $xrange(T_i)$ $(1 \le i \le p)$ to all processors. Each processor computes $T_0$ without any catalogues 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 $j$. Let $l$ be the left vertical boundary of the vertical slab defined by $xrange(v)$; see Figure 4. Let $y(s)$ denote the $y$-coordinate of the intersection of $s$ and $l$. We define $j$ and $y(s)$ as the primary and secondary key for $s$, respectively. Using a global sort, all line segments in the catalogs of all $T_{0,i}$, $(1 \le i \le p)$, are sorted with respect to their primary and secondary key in such a way that no processor stores two line segments with different primary keys. The latter can be achieved by using $2p$ virtual processors.

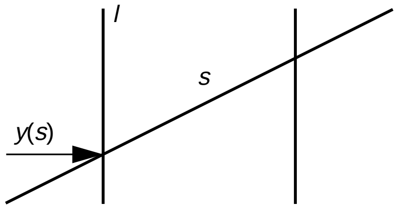

Figure 4: The vertical slab defined by *xrange*(*v*)

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

**Proof.** The memory bound follows from Observation 2 and Observation 3. The algorithm uses a constant number of the basic communication operations of Section 2. The local computation time is bounded by the local time for sorting (Steps 1 and 4). ❑

## 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. [17] for details).

Recall that, at the end of Algorithm 1, processor $P_i$ ($1 \le i \le p$) 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 Figure 3).

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 $P_i$ ($1 \le i \le p$) stores for each $q \in Q_i$ its next element $s$.

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

(2) Using the $first_j$ and $last_j$ values of all processors, processor $P_i$ computes for each $q \in Q_i$ the sublists $l_j$ ($1 \le i \le p$) 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 $\sum_v X_v = n/p$ and $\sum_v Y_v = p$, 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 searched in $l_i$. Let $k(l_i) = \lceil g(l_i)\, p / n \rceil$.

(4) Create $k(l_i)$ copies of $l_i$ ($1 \le i \le p$). 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 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 a global sort operation. (Note that, $\log p \le n/p$)

(7) Determine for each $T_i$ the number, $a(T_i)$, of queries whose search path includes the root of $T_i$ ($1 \le i \le p$). This can be computed by using global 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. 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 $T_i$ ($1 \le i \le p$) by applying the standard sequential next element search algorithm [17] for $S(T_i)$ and its query set.

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

**Theorem 2** *Algorithm 2 solves the next element search problem for n line segments on a CGM with O((n/p) log n) 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. The local computation time is bounded by the local time for sorting and sequential next element search. ❑

## 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 [6].

**Algorithm 3: "Next Element Search for Axis**

**Parallel Line Segments".** *Input*: A set $S$ of $n$ axis parallel line segments and a query set $Q$, 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. Note: During the algorithm, queries will be handled like line segments of zero length.

(1) Sort $S \cup Q$ by increasing $y$-coordinate. Each processor $P_i$ solves, both, the next element search problem and lower envelope problem sequentially for the data $S_i \cup Q_i$ it has received. Let $Q'$ be 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 $x$-coordinate of the right endpoints and the $x$-coordinates of the query points, respectively.

(3) Let $l_1, \dots l_{p-1}$ 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.

**Theorem 3** *Algorithm 3 solves the next element search problem for n axis parallel line segments on a CGM with O(n/p) memory per processor using O(1) communication rounds and O((n/p) log n) local computation.*

**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 2 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 10000. We used 10 sets of data on each combination of $p$ and $n$. In 5 of them the line segments were evenly distributed in a unit square and in the other 5 the line segments were evenly distributed in a unit circle. The average length of the segments are 1/10 unit length.

The result is summarized in the Figure 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 ($h = O(n/p)$), $O((n \log n) / p)$ memory per processor and $O((n/p) \log n)$ local computation. An important advantage of our model is that

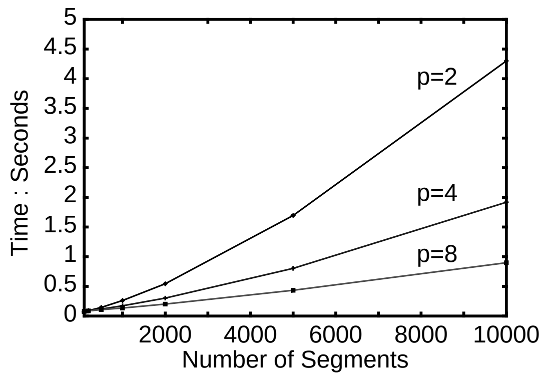

Figure 5: Runing time of Algorithm 3 on an Intel iPSC/860

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. As demonstrated in [5][6][7], 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 the following:

1. **Planar subdivision search problem**. Given a plane graph $G=(V,E)$ with vertex coordinates, and a set of $n$ query points $q_i$ ($1 \le i \le n$), 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 [21] 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 n) memory per processor in O(1) communication rounds and O((n/p) log n) local computation.*

## Acknowledgment

We would like to thank Franco Preparata and Roberto Tamassia for their helpful discussion on the next element search problem for axis parallel line segments.

## Bibliography

1. S.G. Akl and K.A. Lyons, “Parallel Computational Geometry”, Prentice Hall, 1993.
2. R.J. Anderson and L. Snyder, “A Comparison of Shared and Nonshared Memory Models of Computation”, in Proc. of the IEEE, 79(4), 1979, pp. 480-487.
3. J.L. Bentley and D. Wood, “An optimal worst case algorithm for reporting intersections of rectangles”, IEEE Transactions on Computers 29:7, 1980, pp. 571-576.
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. 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.
6. 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.
7. 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.
8. X. Deng, “A Convex Hull Algorithm for Coarse Grained Multiprocessors”, in Proc. 5th International Symposium on Algorithms and Computation, 1994.
9. X. Deng and P. Dymond, “Efficient Routing and Message Bounds for Optimal Parallel Algorithms”, in Proc. Int. Parallel Proc. Symp., 1995.
10. X. Deng and N. Gu, “Good Programming Style on Multiprocessors”, in Proc. IEEE Symposium on Parallel and Distributed Processing, 1994, pp. 538-543.
11. O. Develliers and A. Fabri, “Scalable Algorithms for Bichromatic Line Segment Intersection Problems on Coarse Grained Multicomputers”, in Proc. Workshop on Algorithms and Data Structures (WADS’93), Springer Lecture Notes in Computer Science, 1993, pp. 277-288 (also to appear in the International Journal of Computational Geometry and Applications).
12. A. Ferreira, A. Rau-Chaplin, S. Ueda, “Scalable 2D Convex Hull and Triangulation for Coarse Grained Multicomputers”, in Proc. 7th IEEE Symp. on Parallel and Distributed Processing, San Antonio, 1995, pp. 561-568.
13. 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.
14. M.T. Goodrich, “Communication Efficient Parallel Sorting”, in Proc. 28th Annual ACM Symp. on Theory of Computing (STOC’96), 1996.
15. H. Li and K.C. Sevcik, “Parallel Sorting by Overpartitioning”, in Proc. ACM Symp. on Parallel Algorithms and Architectures, 1994, pp. 46-56.
16. K. Mehlhorn, “Data Structures and algorithms 3: multi-dimensional searching and computational geometry”, Spring Verlag, 1984.
17. F.P. Preparata and M.I. Shamos, “Computational geometry - an introduction”, Springer Verlag, 1985.
18. L. Snyder, “Type Architectures, Shared Memory and the Corollary of Modest Potential”, Annu. Rev. Comput. Sci. 1, 1986, pp. 289-317.
19. L.G. Valiant, “A Bridging Model for Parallel Computation”, Communications of the ACM, 33, 1990, pp. 103-111.
20. L.G. Valiant et. al., “General Purpose Parallel Architectures”, Handbook of Theoretical Computer Science, Edited by J. van Leeuwen, MIT Press/Elsevier, 1990, pp. 943-972.
21. C.K.Yap, “Parallel Triangulation of a Polygon in Two Calls to the Trapezoidal Map,” Algorithmica, Vol.3, 1988, pp 279-288.