Scalable Parallel Computational Geometry for Coarse Grained Multicomputers [*†]

Frank Dehne
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6

Andreas Fabri
INRIA, B.P.93
06902 Sophia-Antipolis Cedex
France

Andrew Rau-Chaplin
DIMACS, Rutgers University
P.O. Box 1179,
Piscataway, NJ 08855-1179
USA

November 22, 1994

Abstract

We study *scalable parallel computational geometry* algorithms for the *coarse grained multicomputer* model: p processors solving a problem on n data items, were each processor has $O(\frac{n}{p}) \gg O(1)$ local memory and all processors are connected via some arbitrary interconnection network (e.g. mesh, hypercube, fat tree). We present $O(\frac{T_{sequential}}{p} + T_s(n,p))$ time scalable parallel algorithms for several computational geometry problems. $T_s(n,p)$ refers to the time of a global sort operation.

Our results are independent of the multicomputer's interconnection network. Their time complexities become optimal when $\frac{T_{sequential}}{p}$ dominates $T_s(n,p)$ or when $T_s(n,p)$ is optimal. This is the case for several standard architectures, including meshes and hypercubes, and a wide range of ratios $\frac{n}{p}$ that include many of the currently available machine configurations.

Our methods also have some important *practical* advantages: For interprocessor communication, they use only a small fixed number of one global routing operation, global sort, and all other programming is in the sequential domain. Furthermore, our algorithms use only a small number of very large messages, which greatly reduces the overhead for the communication protocol between processors. (Note however, that our time complexities account for the lengths of messages.) Experiments show that our methods are easy to implement and give good timing results.

[*]A preliminary version of this paper has been published in the proceedings of the 1993 ACM Conference on Computational Geometry.

[†]This work was partially supported by the Natural Sciences and Engineering Research Council of Canada and the ESPRIT Basic Research Actions Nr. 3075 (ALCOM) and Nr. 7141 (ALCOM II).

1 Introduction

Parallel computational geometry is concerned with solving some given geometric problem of size n on a parallel computer with p processors (e.g., a PRAM, mesh, or hypercube multiprocessor) in time $T_{parallel}$. The parallel solution is *optimal* if $T_{parallel} = O(\frac{T_{sequential}}{p})$, where $T_{sequential}$ is the sequential time complexity of the problem. Theoretical work for parallel computational geometry has so far focussed on the case $\frac{n}{p} = O(1)$, also referred to as the *fine grained* case. However, for parallel geometric algorithms to be relevant in practice, such algorithms must be *scalable*, that is, they must be applicable and efficient for a wide range of ratios $\frac{n}{p}$. The design of such scalable algorithms is also listed as a major goal in the recent "Grand Challenges" report [10].

Yet, only little theoretical work has been done for designing scalable parallel algorithms for computational geometry problems. A related problem was studied in [2, 19]. The model considered there was a host machine with $O(n)$ memory attached to a systolic array of size p with $O(1)$ memory per processors. This model suffers however from the fact that data has to be frequently swapped between the host and the systolic array, and this "I/O bottleneck" is the main factor determining the computation time. A closely related "external memory" model was studied in [9]. At the end of Section 1 we will discuss more in detail the relationship of our work to previous results in the literature.

The architectures of most existing multicomputers (e.g. the Intel Paragon, Intel iPSC/860, and Thinking Machines Corp. CM-5) are quite different. They consist of a set of p *state-of-the-art* processors (e.g. SPARC proc.), each with considerable local memory, connected to some interconnection network (e.g. mesh, hypercube, fat tree). These machines are usually *coarse grained*, i.e. the size of each local memory is "considerably larger" than $O(1)$. In order to minimize the I/O bottleneck, the entire data set for a given problem is immediately loaded into the local memories and remains there until the problem is solved.

The *coarse grained multicomputer* model, or CGM(n,p) for short, considered in this paper is a set of p processors with $O(\frac{n}{p})$ local memory each, connected to some arbitrary interconnection network or a shared memory. The term "coarse grained" refers to the fact that (as in practice) the size $O(\frac{n}{p})$ of each local memory is defined to be "considerably larger" than $O(1)$. Throughout the paper, we will assume that $\frac{n}{p} \geq p$. This assumption is necessary for the correctness of our algorithms. On the other hand, for all currently available coarse grained parallel machines it is clearly true that $\frac{n}{p} \geq p$. It is an interesting open problem whether our methods can be generalized to apply also to the case $\frac{n}{p} < p$. Note that, for determining time complexities we will consider both, local computation time and interprocessor communication time, in the standard way.

The problem studied in this paper is the design of *scalable parallel geometric algorithms* for the coarse grained multicomputer model which are optimal or at least efficient for a wide range of ratios $\frac{n}{p}$.

Note that, if there exists an optimal fine grained algorithm with $T_{parallel} = O(\frac{T_{sequential}}{p})$ then, at least from a theoretical point of view, the problem is trivial. Standard simulation (also referred to as "virtual processor" simulation in many multiprocessor operating systems) gives an optimal algorithm for any ratio of n and p. However, for most interconnection networks used in practice, many problems do not as yet have such optimal fine grained algorithms, or optimal fine grained algorithms are impossible due to bandwidth or diameter limitations (e.g. for the mesh).

We present new techniques for designing efficient scalable parallel geometric algorithms.

Our results are independent of the communication network (e.g. mesh, hypercube, fat tree). A particular strength of our approach, which is very different from the one presented in [2, 9], is that all interprocessor communication is restricted to a constant number of usages of one single global routing operation: *global sort.*

In a nutshell, the basic idea for our methods is as follows: We try to combine optimal sequential algorithms for a given problem with an efficient global routing and partitioning mechanism. We devise a constant number of partitioning schemes of the global problem (on the entire data set of n data items) into p subproblems of size $O(\frac{n}{p})$. Each processor solves sequentially a constant number of such $O(\frac{n}{p})$ size subproblems, and we use a constant number of global routing operations to permute the subproblems between the processors. Eventually, by combining the $O(1)$ solutions of its $O(\frac{n}{p})$ size subproblems, each processor determines its $O(\frac{n}{p})$ size portion of the *global* solution.

The above is necessarily an oversimplification. The actual algorithms will do more than just those permutations. The main challenge lies in devising the above mentioned partitioning schemes. Note that, each processor will solve only a constant number of $O(\frac{n}{p})$ size subproblems, but eventually will have to determine its part of the entire $O(n)$ size problem (without having seen all of the n data items). The most complicated part of the algorithm is to ensure that at most $O(1)$ global communication rounds are required.

We present scalable parallel algorithms for solving the following well known geometric problems on the coarse grained multicomputer model:

(1) area of the union of rectangles,

(2) $3D$-maxima,

(3) $2D$-nearest neighbors of a point set,

(4) lower envelope of non-intersecting line segments in the plane (and, with slightly more memory, for possibly intersecting line segments),

(5) $2D$-weighted dominance counting,

(6) multisearch on balanced search trees, segment tree construction, and multiple segment tree search.

We also study the following applications of (6): the problem of determining for a set of simple polygons all directions for which a uni-directional translation ordering exists, and determining for a set of simple polygons a multi-directional translation ordering.

Our scalable parallel algorithms for Problems 1-6 have a running time of

$$O(\frac{T_{sequential}}{p} + T_s(n,p))$$

on a p-processor coarse grained multicomputer, CGM(n,p), with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$ where $\frac{n}{p} \geq p$. $T_s(n,p)$ refers to the time to sort globally n data items stored on a CGM(n,p), $\frac{n}{p}$ data items on each processor.

Since $T_{sequential} = \Theta(n \log n)$ for Problems 1-6, our algorithms either run in optimal time $\Theta(\frac{n \log n}{p})$ or in sort time $T_s(n,p)$ for the respective architecture. Our results become optimal when $\frac{T_{sequential}}{p}$ dominates $T_s(n,p)$ or when $T_s(n,p)$ is optimal. Note that, sort time is a lower bound for the time complexities of all of the above problems.

Consider for example the *mesh* architecture. For the fine grained case, $\frac{n}{p} = O(1)$, a time complexity of $O(\sqrt{n})$ is the best we can achieve due to the diameter of the network. Standard simulation of the existing results on a coarse grained machine gives $O(\frac{n}{p}\sqrt{n})$ time coarse grained methods. Our methods for the above problems run in time $O(\frac{n}{p}(\log n + \sqrt{p}))$, a considerable improvement over the existing methods. For the *hypercube*, our algorithms are optimal for $n \geq p^{\log p}$, in which case they also yield a considerable improvement over previous methods.

Experiments with an implementation of our lower envelope algorithm (Problem 4) on a CM-5 and iPSC/860 have shown that, in addition to being scalable, our algorithm for Problem 4 quickly reaches the point of linear speed-up for reasonable data sizes. Even with modest programming efforts, our implementations showed good timing results. This is due largely to the following two facts:

(1) Our algorithms use only one well known and extensively studied global routing operation. *Global sort* is usually available as a system call (often implemented on machine level by the same group who wrote the operating system) or can be obtained as highly optimized public domain software. All other programming is within the sequential domain. Even with modest programming efforts, this produces highly optimized parallel programs.

(2) On most architectures, for each message exchanged between two processors, there is a considerable overhead involved (creating a communication channel, setting up the communication protocol, etc.) which is independent of the size of the message. Existing parallel computational geometry algorithms, applied to a coarse grained machine, tend to produce many short messages. Our methods involve only a small fixed number of global communication rounds, where large packets of size $O(\frac{n}{p})$ are exchanged between processors (i.e., the processors essentially swap their entire memory contents).

Our results are extensions of the methods in [2, 19] which study a machine model consisting of a host machine with $O(n)$ memory attached to a systolic array of size p with $O(1)$ memory per processor. The main architectural difference is that, in our model the data is already stored in the processors' memories, which allows improved computation times because the "I/O bottleneck" is not any more the determining factor. Nevertheless, several of the data partitioning schemes presented in [2] have been very useful for our methods. In fact, J.J. Tsay [19] pointed out that, for the special case of *hypercubic* networks, their methods can be generalized to a machine model with $O(\frac{n}{p})$ memory per processor, even for any ratio $\frac{n}{p} \geq p^{\epsilon}$, $\epsilon > 0$. One of the main contributions of our paper is that our methods can be applied to any interconnection network. Furthermore, the methods indicated in [19] are recursive and require more than a constant number of communication rounds. Our experiments show that few communication rounds (with large messages) are an important feature for good practical performance. This is another important advantage of the methods presented in this paper. It is an interesting open problem to study whether our methods can be generalized to work for ratios $\frac{n}{p} < p$ for arbitrary networks and with a constant number of communication rounds.

The remainder of this paper is organized as follows: In the next section we give more details about the coarse grained multicomputer model, CGM(n, p). Sections 3-8 present our algorithms for the problems listed above, and experimental results will be discussed in Section 9.

2 The "Coarse Grained Multicomputer" Model

The *coarse grained multicomputer*, CGM(n, p), considered in this paper is a set of p processors numbered from 1 to p with $O(\frac{n}{p})$ local memory each, connected via some arbitrary interconnection network or a shared memory. Commonly used interconnection networks for a CGM include the 2D-mesh (e.g. Intel Paragon), hypercube (e.g. Intel iPSC/860) and the fat tree (e.g. Thinking Machines CM-5). Each processor may exchange messages of $O(\log n)$ bits with any one of its immediate neighbors in constant time. For determining time complexities, we will consider both, local computation time and interprocessor communication time, in the standard way. The term "coarse grained" refers to the fact that the size $O(\frac{n}{p})$ of each local memory is assumed to be "considerably larger" than $O(1)$. Our definition of "considerably larger" will be that $\frac{n}{p} \geq p$.

2.1 The Basic Communication Operation: Global Sort

Global sort refers to the operation of sorting $O(n)$ data items stored on a CGM(n, p), $O(\frac{n}{p})$ data items per processor, with respect to the CGM's processor numbering. $T_s(n, p)$ refers to the time complexity of a global global sort.

Note that, for a mesh $T_s(n, p) = \Theta(\frac{n}{p}(\log n + \sqrt{p}))$ and for a hypercube $T_s(n, p) = O(\frac{n}{p}(\log n + \log^2 p))$. These time complexities are based on [14] and [3], respectively. Note that for the hypercube better deterministic algorithms exist [6], but they are not of practical use. One could also use randomized sorting [17], but in this paper we will only consider deterministic methods. We refer the reader to [3, 5, 11, 13, 14, 17] for a more detailed discussion of the different architectures and routing algorithms.

It is interesting to study, for which ratio of n and p the global sort becomes optimal, that is $T_s(n, p) = O(\frac{n \log n}{p})$. A simple calculation shows that the above sort methods are optimal for a mesh with $n \geq 2^{\sqrt{p}}$ and a hypercube with $n \geq p^{\log p}$.

2.2 Other Communication Operations Based On Global Sort

For ease of description of our algorithms presented in the remainder, we will now outline four other operations for interprocessor communication. All of these operations can be implemented as a constant number of global sort operations and $O(\frac{n}{p})$ time local computation. Note that, for some interconnection networks it might be better in practice to implement these operations directly rather than using global sort. This can improve the time complexity constants of the algorithms described in the remainder.

Segmented broadcast: In a segmented broadcast operation, $q \leq p$ processors with numbers $j_1 < j_2 < \ldots < j_q$ are selected. Each such processor p_{j_i} broadcasts a list of $\frac{n}{p}$ data items from its local memory to the processors $p_{j_i+1} \ldots p_{j_{i+1}-1}$. The time for a segmented broadcast operation will be referred to as $T_{sb}(n, p)$.

Multinode broadcast: In a multinode broadcast operation, every processor sends one message to all other processors. The time complexity will be denoted as $T_b(p)$. For any interconnection network, $T_b(p) = O(p)$.

Total exchange: In a total exchange operation, every processor (in parallel) sends a different message to every other processor. The time complexity will be denoted as $T_x(p)$.

Partial sum (Scan): Every processor stores one value, and all processors compute the partial sums of these values with respect to some associative operator. The time complexity will be denoted as $T_p(p)$.

Lemma 1 *For any* CGM(n,p) *with* $\frac{n}{p} \geq p$,

(a) $T_{sb}(n,p) = O(\frac{n}{p} + T_s(n,p))$,

(b) $T_b(p) = O(\frac{n}{p} + T_s(n,p))$,

(c) $T_x(p) = O(\frac{n}{p} + T_s(n,p))$, *and*

(d) $T_p(p) = O(\frac{n}{p} + T_s(n,p))$.

Proof. For Part (a) we show that if $\frac{n}{p} \geq p$ then segmented broadcast can be simulated by $O(1)$ global sorts and $O(\frac{n}{p})$ time local computation.

Define an operation *segmented 1-broadcast* as follows: $r \leq p$ processors with numbers $k_1 < k_2 < \ldots < k_r$ are selected. Each processor p_{k_i} broadcasts one data item from its local memory to processors $p_{k_i} \ldots p_{k_{i+1}-1}$, and each processor creates $O(\frac{n}{p})$ copies of the received data item.

Segmented 1-broadcast can be simulated by $O(1)$ global sorts and $O(\frac{n}{p})$ local computation as follows: Using global sort, compress into processor p_0 all data items to be broadcast, and an *empty item* from each processor not broadcasting anything. Create the copies to be broadcast locally at processor p_0 by filling the empty items. Uncompress the data items using another global sort.

We now describe how a segmented broadcast can be reduced to a segmented 1-broadcast. Consider the $q \leq p$ processors $p_{j_1}, p_{j_2}, \ldots, p_{j_q}$ selected for the segmented broadcast and define $label(p_i) = k$ if and only if $j_k \leq i < j_{k+1}$. Create $\frac{n}{p}$ data items for each processor which are either the data items to be broadcast or $\frac{n}{p}$ *empty items*. Sort all n data items globally, using for each item x stored at processor p_i $(1 \leq i \leq p)$ $label(p_i)$ as first key, the rank of x in the local list of the $\frac{n}{p}$ items at p_i as second key, and i as the third key. After this sort, consider the total list of all items over all processors (ordered by processor number). Each item, y, to be broadcast is followed by all the empty items that need to be filled with y. For all those cases were y and its respective empty items reside within the same processor, the filling can be performed locally. The filling process for the remaining empty items reduces to a segmented 1-broadcast. After the filling is complete, the above sorting process is inverted, and the segmented broadcast is complete.

Parts (b), (c) and (d) are obvious. $\square$

3 Area of the Union of Isothetic Rectangles

Given a set R of n isothetic rectangles, the *measure problem* is to compute the area M covered by the union of R.

Assume that the vertical edges of all rectangles $r \in R$ are sorted by their x-coordinate and let $\mathcal{L} = \{l_1, \ldots, l_p\}$ be the set of vertical lines passing through every $\frac{n}{p}$-th vertical edge. Analogously let $\mathcal{H} = \{h_1, \ldots, h_p\}$ be the set of horizontal lines passing through every $\frac{n}{p}$-th horizontal edge. Let V_j be the vertical slab between l_j and l_{j+1}, let H_i be the horizontal slab between h_i and h_{i+1}, and let box b_{ij} be the intersection of H_i and V_j. See Figure 1.

For each box b_{ij} consider the rectangles $r \in R$ which have one or more vertices in b_{ij}. The horizontal lines through all these vertices inside b_{ij} cut b_{ij} into rectangles called stripes. Note that, the total number of stripes in a horizontal or vertical slab is $O(\frac{n}{p})$.

For each stripe s let $xcover(s)$ be the total length of the parts of its upper boundary covered by rectangles intersecting the stripe s with at least one vertical edge. Let $ycover(s)$

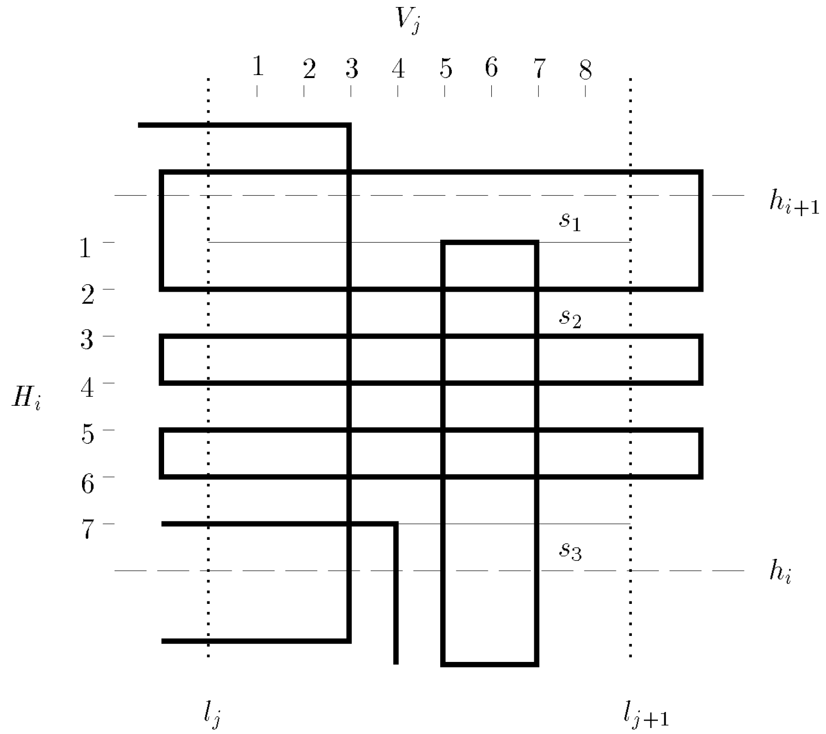

Figure 1: Illustration of Lemma 2. Box b_{ij} consists of stripes s_1, s_2 and s_3, with $xcover(s_1) = 3$, $xcover(s_2) = xcover(s_3) = 5$, $ycover(s_1) = 1$, $ycover(s_2) = 3$, and $ycover(s_3) = 0$.

be the total length of the parts of the right boundary of s covered by rectangles intersecting the box b_{ij} with at least one horizontal edge and having no corner in b_{ij}.

Lemma 2 *Consider a box b_{ij} with stripes $s_1, \ldots, s_r$ and define as $m(b_{ij})$ the contribution of box b_{ij} to the area M covered by the union of R. Two possible cases may occur:*

(a) b_{ij} is contained in some rectangle $r \in R$, in which case $m(b_{ij})$ is the total area of b_{ij},

(b) $m(b_{ij}) = \sum_{1 \leq i \leq r} m(s_i)$ where each stripe s_i contributes an area
$m(s_i) = xcover(s_i)height(s_i) + ycover(s_i)length(s_i) - xcover(s_i)ycover(s_i)$.

Lemma 2 suggests the following algorithm: First we detect for each box b if it is contained in a rectangle r, and if this is the case we set $m(b)$ equal to the area of b. For each remaining box b we determine its contribution $m(b)$ by computing the values $m(s_i)$ for all its stripes s_i. The latter is obtained by computing for each stripe s the two values $xcover(s)$ within its vertical slab and $ycover(s)$ within its horizontal slab.

The following is an outline of our scalable parallel algorithm for solving the measure problem.

Algorithm 1
Architecture: A p-processor coarse grained multicomputer, CGM(n,p), with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $\frac{n}{p} \geq p$.
Input: Each processor stores $\frac{n}{p}$ rectangles $r \in R$.
Output: The area of the union of all $r \in R$.

(1) Globally sort the vertical (horizontal) edges of the rectangles by their x-coordinate (y-coordinate) and compute the set $\mathcal{L} = \{l_1, \ldots, l_p\}$ ($\mathcal{H} = \{h_1, \ldots, h_p\}$). Perform a multinode broadcast, where each processor p_i sends the edge l_i (h_i) with maximal x-coordinate (y-coordinate). After that, each processor holds a copy of $\mathcal{L}$ and $\mathcal{H}$. Rearrange all data using global sort such that each processor stores a vertical slab, that is all rectangles with a vertex in that vertical slab, and locally compute all stripes in that vertical slab.

(2) On each processor compute locally $xcover(s)$ for all stripes s in the respective vertical slab. Perform a plane sweep in upwards direction in time $O(\frac{n}{p} \log n)$, using the sequential measure of rectangles algorithm in [20] with minor adaptations.

(3) Determine all boxes b which are contained in a rectangle $r \in R$: Each processor locally builds a segment tree for $\mathcal{L}$ and $\mathcal{H}$, each. Using these segment trees, determine for each box b in the vertical slab the number $l(b)$ of boxes to its left that are covered by the rectangles with a corner in the vertical slab. Using a global sort rearrange the p^2 boxes such that each processor contains the boxes in a horizontal slab and their respective values $l(b)$. For each horizontal slab determine locally all covered boxes.

(4) Rearrange the data using global sort such that each processor stores a horizontal slab, all stripes in that horizontal slab, and all rectangles with a vertex in that horizontal slab. On each processor compute locally $ycover(s)$ for all stripes s in the respective horizontal slab which are not contained in a covered box. Perform a plane sweep in horizontal direction in time $O(\frac{n}{p} \log n)$, using the sequential measure of rectangles algorithm in [20] with minor adaptations.

(5) Compute locally the values $m(s)$ for all stripes not contained in a covered box, and the values $m(b)$ for boxes. Compute locally the sum of all $m(s)$ and $m(b)$ of all stripes and boxes stored at each processor. Add these values over all processors.

— End of Algorithm —

Theorem 1 *The measure problem for a set of n isothetic rectangles can be solved on a p-processor coarse grained multicomputer with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $\frac{n}{p} \geq p$, in time $O(\frac{n \log n}{p} + T_s(n,p))$.*

Proof. The correctness of Algorithm 1 follows from Lemma 2. Note that, each rectangle contributing to $xcover(s)$ of some stripe s has a vertex in the vertical slab containing s, and each rectangle contributing to $ycover(s)$ of some stripe s has a vertex in the horizontal slab containing s.

The space requirement is $O(\frac{n}{p} + p) = O(\frac{n}{p})$ per processor. In each step, the local computation time is at most $O(\frac{n}{p} \log n)$. The global communication in each step reduces to a constant number of global sorts and communication operations listed in Section 2.2. Hence, the time complexity follows from Lemma 1. $\square$

4 3D-Maxima

Consider a set S of n points in 3-space. For a point v let $x(v)$, $y(v)$, and $z(v)$ denote the x-coordinate, y-coordinate, and z-coordinate, respectively of v. Point v *dominates* a point w if and only if $x(v) > x(w)$, $y(v) > y(w)$, and $z(v) > z(w)$. A point is *maximal* in S if it is not dominated by any other point of S. The *3D-maxima problem* consists of determining the set $3Dmax(S)$ of all maximal points in S.

Consider a set of p horizontal planes $\mathcal{H}_i$ (parallel to the x,y-plane, $\mathcal{H}_i$ below $\mathcal{H}_{i+1}$) which partition S into p subsets H_i (the points between $\mathcal{H}_i$ and $\mathcal{H}_{i+1}$) of $\frac{n}{p}$ points each. Analogously, consider p vertical planes $\mathcal{V}_j$ (parallel to the y,z-plane, $\mathcal{V}_j$ to the left of $\mathcal{V}_{j+1}$) which partition S into p subsets V_j of $\frac{n}{p}$ points each. See Figure 2 for an illustration.

Let H_i' be the projection of all points of H_i onto the plane $\mathcal{H}_i$, and let $2Dmax(H_i')$ be the set of 2-D maximal points of H_i' within plane $\mathcal{H}_i$. Define $2Dmax_i$ to be the monotone chain within plane $\mathcal{H}_i$ induced by $2Dmax(H_i')$, and let q_{ij} be the intersection of $2Dmax(H_i')$ and $\mathcal{V}_{j+1}$; see Figure 2.

Observation 1 *Any point $v \in V_j \setminus H_i$ which is dominated by a point $w \in H_i \setminus V_j$ is also dominated by q_{ij}.*

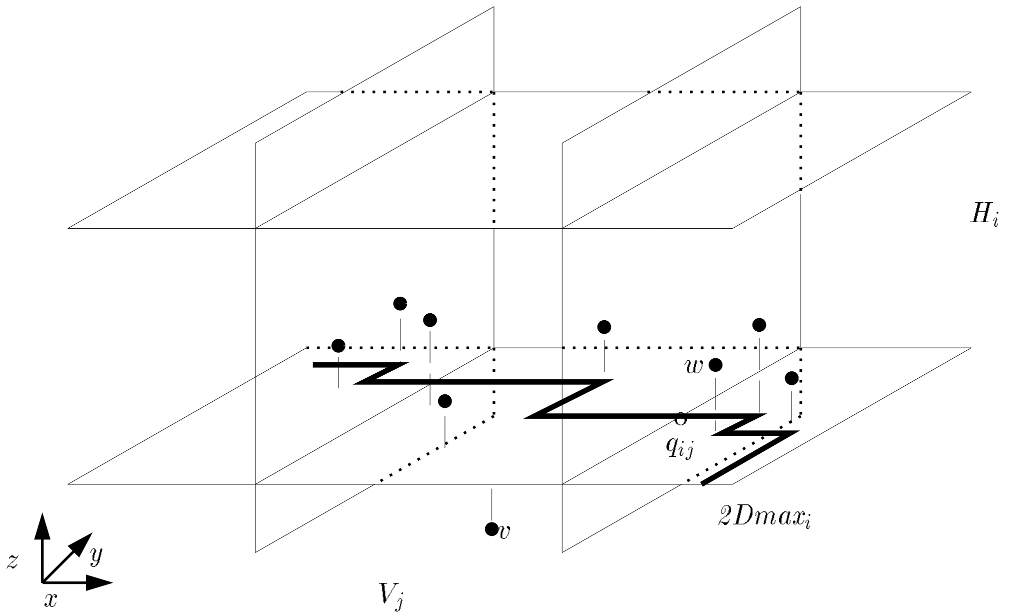

Figure 2: Illustration of Observation 1. The point $v \in V_j \setminus H_i$ is dominated by $w \in H_i \setminus V_j$ and thus dominated by the intersection point q_{ij}.

Define $Q_i = \{q_{ij} | 1 \leq j \leq p\}$. Note that $|Q_i| = p$ because $2Dmax_i$ is monotone. Observation 1 suggests the following algorithm for solving the 3D-maxima problem.

Algorithm 2
Architecture: A p-processor coarse grained multicomputer, CGM(n,p), with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $\frac{n}{p} \geq p$.
Input: Each processor stores $\frac{n}{p}$ points of S.
Output: Each processor stores at most $\frac{n}{p}$ maximal points of S.

(1) Globally sort S by x-coordinate. Processor p_j stores subset V_j and bounding plane $\mathcal{V}_j$. Perform a multinode broadcast where each processor p_j sends bounding plane $\mathcal{V}_j$ to all other processors. As a result, each processor stores all bounding planes $\mathcal{V}_1, \ldots \mathcal{V}_p$.

(2) Globally sort S by z-coordinate. Processor p_i stores subset H_i and bounding plane $\mathcal{H}_i$. Each processor p_i computes locally $3Dmax(H_i)$ using the standard sequential algorithm as described e.g. in [16], and removes all points dominated in H_i. Each processor p_i computes locally the 2D-projection H_i', $2Dmax(H_i')$, and the monotone chain $2Dmax_i$. Using the bounding planes $\mathcal{V}_1, \ldots \mathcal{V}_p$, each processor p_i computes the set Q_i.

(3) Globally sort $\bigcup_{i=1}^{p}(3Dmax(H_i) \cup Q_i)$ by x-coordinate. Processor p_j stores the set V_j^* consisting of the points of $\cup_{i=1}^{p} 3Dmax(H_i)$ between the bounding planes $\mathcal{V}_j$ and $\mathcal{V}_{j+1}$ as well as $\{q_{ij} | 1 \leq i \leq p\}$. Note that $|V_j^*| \leq \frac{n}{p}$. Each processor p_j computes locally $3Dmax(V_j^*)$ using the standard sequential algorithm. The reported result, $3Dmax(S)$, is $\cup_{i=1}^{p} 3Dmax(V_j^*)$.

— End of Algorithm —

Theorem 2 *The 3D-maxima problem for a set of n points in 3-space can be solved on a p-processor coarse grained multicomputer with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $\frac{n}{p} \geq p$, in time $O(\frac{n \log n}{p} + T_s(n,p))$.*

Proof. The correctness of Algorithm 2 follows from Observation 1. The space requirement is $O(\frac{n}{p} + p) = O(\frac{n}{p})$ per processor. In each step, the local computation time is at most $O(\frac{n}{p} \log n)$. The global communication in each step reduces to a constant number of global sorts and communication operations listed in Section 2.2. Hence, the time complexity follows from Lemma 1. $\Box$

5 2D-Nearest Neighbors of a Point Set

Given a set S of n points in the Euclidean plane, the *all-nearest neighbor problem* for S is to determine for each point $v \in S$ its nearest neighbor $NN_S(v)$ in S, where $NN_S(v)$ is formally defined as a point $w \in S \setminus \{v\}$ such that $dist(v,w) \leq dist(v,u)$ for all $u \in S \setminus \{v\}$.

Consider a set of p horizontal lines which partition S into p subsets H_i of $\frac{n}{p}$ points each. Analogously, consider p vertical lines which partition S into p subsets V_j of $\frac{n}{p}$ points each. See Figure 3 for an illustration. Let I_{ij} denote the four point where the boundary lines of H_i and V_j cross. Define C_{ij} as the set of all $w \in V_j \setminus H_i$ such that w is closer to a point of I_{ij} than to its nearest neighbor $NN_{V_j}(w)$ in V_j.

We recall the following lemma from [2].

Lemma 3 *$|C_{ij}| \leq 8$, and every $w \in V_j \setminus H_i$ such that $NN_S(w) \in H_i \setminus V_j$ is an element of C_{ij}.*

Algorithm 3
Architecture: A p-processor coarse grained multicomputer, CGM(n,p), with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $\frac{n}{p} \geq p$.
Input: Each processor stores $\frac{n}{p}$ points of S.
Output: Each processor stores $NN_S(v)$ for each of its points.

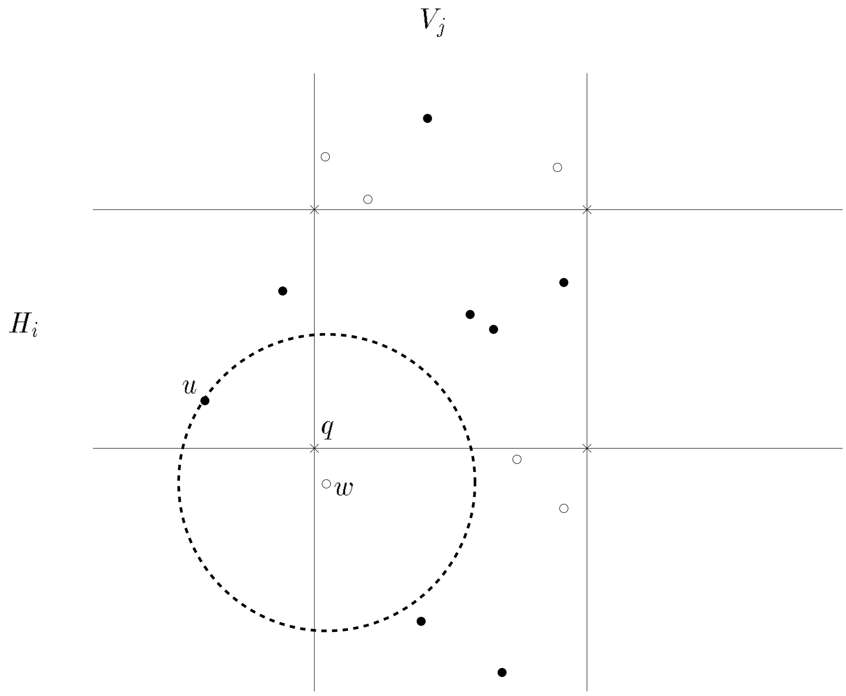

Figure 3: Illustration of Lemma 3. The points $\circ$ belong to the set C_{ij}, the points $\times$ denote the set I_{ij}.

(1) Globally sort S by y-coordinate, such that processor p_i stores subset H_i and the two horizontal lines bounding H_i. Perform a multinode broadcast, such that every processor stores all p horizontal bounding lines. Every processor p_i computes sequentially $NN_{H_i}(v)$ for each of its points $v \in H_i$, using the standard sequential algorithm [16].

(2) Globally sort S by x-coordinate, such that processor p_j stores subset V_j and the two vertical lines bounding V_j. Every processor p_j computes sequentially $NN_{V_j}(v)$ for each of its points $v \in V_j$, using the standard sequential algorithm. Using all p horizontal bounding lines broadcast in Step 1, every processor p_j computes the p sets I_{ij} and C_{ij}, $1 \leq i \leq p$.

(3) Globally sort S by y-coordinate, such that processor p_i stores subsets H_i and $C_i := \bigcup_{j=1}^{p} C_{ij}$. Every processor p_i computes sequentially $NN_{H_i \cup C_i}(v)$ for each of its points $v \in H_i \cup C_i$, using the standard sequential algorithm.

(4) Using global sort, the three "nearest neighbors" for each $v \in S$, as determined in Steps 1-3, are routed back to the processor who initially stored v, and the closest one is reported as $NN_S(v)$.

— End of Algorithm —

Theorem 3 *The all-nearest neighbor problem for a set of n points in the Euclidean plane can be solved on a p-processor coarse grained multicomputer with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $\frac{n}{p} \geq p$, in time $O(\frac{n \log n}{p} + T_s(n,p))$.*

Proof. The correctness of Algorithm 2 follows from Lemma 3. The space requirement is $O(\frac{n}{p} + p) = O(\frac{n}{p})$ per processor. In each step, the local computation time is at most $O(\frac{n}{p} \log n)$. The global communication in each step reduces to a constant number of global sorts and communication operations listed in Section 2.2. Hence, the time complexity follows from Lemma 1. $\square$

6 Lower Envelope of Non-Intersecting Line Segments in the Plane

Given a set S of n non-intersecting line segments in the plane, the *lower envelope* problem consists of computing the set $LE(S)$ of segment portions visible from the point $(0, -\infty)$.

Observation 2 *The lower envelope of n non-intersecting line segments is x-monotone and has size $O(n)$.*

Algorithm 4
Architecture: A p-processor coarse grained multicomputer, CGM(n, p), with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $\frac{n}{p} \geq p$.
Input: Each processor p_i stores a set S_i of $\frac{n}{p}$ line segments of S.
Output: Each processor stores $O(\frac{n}{p})$ segment portions of $LE(S)$.

(1) Each processor p_i computes sequentially $LE(S_i)$ for its subset S_i of line segments (ignoring all other segments) [12].

(2) Globally sort the segments in $\bigcup_{i=1}^{p} LE(S_i)$ by the x-coordinate of their right endpoints, which moves to each processor p_i a new set V_i of $O(\frac{n}{p})$ segments. Note that, each processor p_i also keeps the set $LE(S_i)$.

(3) Each processor p_i determines the vertical line l_i through the rightmost vertex of a segment of V_i. Perform a multinode broadcast where processor p_i sends l_i to all other processors. Hence, each processor stores all p vertical lines $l_1, \ldots, l_p$.

(4) Perform a total exchange, with processor p_i sending segment $s \in LE(S_i)$ to processor p_j iff s intersects the vertical line l_j. Let R_j be the set of segments received by processor p_j.

(5) Each processor p_i computes sequentially $LE(V_i \cup R_i)$.

— End of Algorithm —

Theorem 4 *The lower envelope problem for a set of n non-intersecting line segments in the plane can be solved on a p-processor coarse grained multicomputer with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $\frac{n}{p} \geq p$, in time $O(\frac{n \log n}{p} + T_s(n, p))$.*

Proof. Since each $LE(S_i)$ is x-monotone (Observation 2), it can intersect each vertical line l_j at most once. Hence, each R_j has a cardinality of at most p. Furthermore, each segment r in V_i can only be obstructed by a segment r' from another set V_j if $j > i$ and r' intersects l_i. The space requirement is $O(\frac{n}{p} + p) = O(\frac{n}{p})$ per processor. In each step, the local computation time is at most $O(\frac{n}{p} \log n)$. The global communication in each step reduces to a constant number of global sorts and communication operations listed in Section 2.2. Hence, the time complexity follows from Lemma 1. $\square$

The above algorithm also computes the lower envelope for possibly intersecting line segments. The only difference is that the size of the lower envelope may become $O(n\alpha(n))$, where $\alpha()$ is the extremely slow growing inverse Ackermann function. As we cannot produce completely balanced output, this may require $O(n\alpha^2(n))$ memory space. Although each processor has at most $O(\frac{n}{p}\alpha(\frac{n}{p}))$ line segments after Step 4, it may happen that in Step 5 a processor computes $O((\frac{n}{p}\alpha(\frac{n}{p}))\alpha(\frac{n}{p}\alpha(\frac{n}{p})))$ line segments. This is no contradiction to the upper bound for the size of the lower envelope of n line segments, if, for example, all the other processors produce lower envelopes of size $O(\frac{n}{p})$ and if $p > \alpha(n)$. We thus obtain the following.

Corollary 1 *The lower envelope problem for a set of n possibly intersecting line segments in the plane can be solved on a p-processor coarse grained multicomputer with arbitrary interconnection network and local memories of size $O(\frac{n\alpha^2(n)}{p})$, $\frac{n\alpha^2(n)}{p} \geq p$, in time $O(\frac{n\alpha^2(n)\log n}{p} + T_s(n\alpha^2(n), p))$.*

7 2D-Weighted Dominance Counting

Let S be a set of n points in the plane with some *weight* $w(v)$ assigned to each $v \in S$. The *2D-weighted dominance counting* problem consists of determining for each $v \in S$ the total weight, $wdom(v, S)$, of all points of S which are dominated by v.

Consider a set of p horizontal lines h_i which partition S into p subsets H_i of $\frac{n}{p}$ points each (with h_i below H_i, and h_{i+1} above H_i). Analogously, consider p vertical lines l_j which partition S into p subsets V_j of $\frac{n}{p}$ points each (with l_j to the left of V_j, and l_{j+1} to the right of V_j).

For a subset $A \subset S$ let $w(A) = \sum_{a \in A} w(a)$. Denote with S_{ij} the set of points in S which are below h_i and to the left of l_j, and let V_{ij} be the set of points of V_j that are below h_i.

Observation 3

(a) For each point $v \in H_i \cap V_j$:
$$wdom(v, S) = wdom(v, H_i) + wdom(v, V_j) - wdom(v, H_i \cap V_j) + w(S_{ij}).$$

(b) $w(S_{ij}) = \sum_{k=1}^{j-1} w(V_{ik})$.

Algorithm 5
Architecture: A p-processor coarse grained multicomputer, CGM(n, p), with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $\frac{n}{p} \geq p$.
Input: Each processor stores $\frac{n}{p}$ points of S.
Output: Each processor stores $wdom(v, S)$ for each of its $\frac{n}{p}$ points $v \in S$.

(1) Globally sort the points by their y-coordinates such that processor p_i stores H_i and h_i. Perform a multinode broadcast, where processor p_i sends h_i to all other processors; i.e. every processor stores now all horizontal lines $h_1, \ldots, h_p$.

(2) Each processor p_i sequentially computes $wdom(v, H_i)$ for each $v \in H_i$.

(3) Globally sort the points by their x-coordinates such that processor p_j stores V_j and l_j.

(4) Each processor p_j sequentially computes $wdom(v, V_j)$ for each $v \in V_j$.

(5) Each processor p_j determines the sets $V_j \cap H_1, \ldots, V_j \cap H_p$ using the lines $h_1, \ldots, h_p$, respectively, received in Step 1, and computes sequentially $wdom(v, H_i \cap V_j)$ for each $v \in H_i \cap V_j$.

(6) Each processor p_j determines the sets V_{1j}, ..., V_{pj} using the lines h_1, ..., h_p, respectively, received in Step 1, and computes sequentially $w(V_{1j})$, ..., $w(V_{pj})$.

(7) Perform a total exchange, where processor p_j sends $w(V_{ij})$ to processor p_{i+1}, $1 \leq i < p$.

(8) Globally sort the points by their y-coordinates such that processor p_i stores H_i.

(9) Each processor p_i sequentially computes $w(S_{ij}) = \sum_{k=1}^{j-1} w(V_{ik})$, and $wdom(v, S) = wdom(v, H_i) + wdom(v, V_j) - wdom(v, H_i \cap V_j) + w(S_{ij})$ for each $v \in H_i \cap V_j$, $1 \leq j \leq p$.

— End of Algorithm —

Theorem 5 *The 2D-weighted dominance counting problem for a set of n weighted points in the plane can be solved on a p-processor coarse grained multicomputer with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $\frac{n}{p} \geq p$, in time $O(\frac{n \log n}{p} + T_s(n,p))$.*

Proof. The correctness of Algorithm 5 follows from Observation 3. The space requirement is $O(\frac{n}{p} + p) = O(\frac{n}{p})$ per processor. In each step, the local computation time is at most $O(\frac{n}{p} \log n)$. The global communication in each step reduces to a constant number of global sorts and communication operations listed in Section 2.2. Hence, the time complexity follows from Lemma 1. $\square$

8 Parallel Tree Search and Applications

Let $T = (V, E)$ be a balanced k-ary tree of size n and height $h = O(\log_k n)$, where k is a fixed constant. We recall from [7] the definition of the multisearch problem for T and a set $Q = \{q_1, \ldots, q_m\}$ of $m = O(n)$ search queries on T.

Each query $q \in Q$ has a *search path*, $path(q) = (v_1(q), \ldots, v_h(q))$, of h vertices of T (from the root to a leaf of T) which is a sequence defined by a successor function $f : (V \cup start) \times Q \to V$ with the following properties: $f(start, q) = v_1$, $f(v_i, q) = v_{i+1}$ where $(v_i, v_{i+1}) \in E$ and $f(v_i, q)$ can be computed by a single processor in time $O(1)$. We say that query q *visits* node $v_t(q)$ at time t. The *multisearch problem* for Q on T consists of executing (in parallel) all m search processes induced by the m search queries. It is important to note that the m search processes may overlap arbitrarily. That is, at any time t, any node of T may be visited by an arbitrary number of queries. See [1, 7] for more details.

Define as T_0 the subtree of T induced by the root and all nodes of T which have a distance from the root of at most $\log_k p$. Subtree T_0 has $p' \leq p$ leaves. To simplify exposition, assume w.l.o.g. that $p' = p$. Let T_i be the subtree of T rooted at the i-th leaf of T_0, $1 \leq i \leq p$.

Algorithm 6
Architecture: A p-processor coarse grained multicomputer, $\mathrm{CGM}(n,p)$, with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $\frac{n}{p} \geq p$.
Input: Each processor stores $\frac{n}{p}$ nodes of T and $\frac{m}{p} = O(\frac{n}{p})$ queries $q \in Q$.
Result: Each $q \in Q$ visits its entire search path $path(q)$.

(1) Using a total exchange operation, create p copies of T_0 and distribute them such that each processor has one copy of T_0.

(2) Using its copy of T_0, each processor performs the first $\log_k p$ multisearch steps for its $O(\frac{n}{p})$ search queries.

(3) For each tree T_i compute $c(T_i) = \left\lceil \frac{|\{q \in Q : v_{\log_k p}(q) \in T_i\}|}{\frac{m}{p}} \right\rceil$, $1 \leq i \leq p$.

(4) Create $c(T_i)$ copies of each subtree T_i and and distribute them such that each processor stores at most two subtrees.

(5) Redistribute Q such that every query $q \in Q$ is stored at a processor that also stores a copy of the subtree T_i $(1 \leq i \leq p)$ containing $v_{\log_k p}(q)$.

(6) Each processor performs the remaining $h - \log_k p$ multisearch steps for its $O(\frac{n}{p})$ search queries.

— End of Algorithm —

Theorem 6 *The multisearch problem for a balanced search tree of size $O(n)$ and fixed degree k, and a set of $m = O(n)$ search queries, can be solved on a p-processor coarse grained multicomputer with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $\frac{n}{p} \geq p$, in time $O(\frac{n \log n}{p} + T_s(n,p))$.*

Proof. The correctness of Algorithm 6 follows from the following three observations. First, all subtrees $T_0, T_1, \ldots, T_p$ have a size of $O(\frac{n}{p})$. Then, the total number $\sum_{i=1}^{p} c(T_i)$ of all tree copies created in Step 4 is $O(p)$. Finally, in Step 5, the number of queries moved to each processor is $O(\frac{n}{p})$. The space requirement is $O(\frac{n}{p} + p) = O(\frac{n}{p})$ per processor. In each step, the local computation time is at most $O(\frac{n}{p} \log n)$. The global communication in each step reduces to a constant number of global sorts and communication operations listed in Section 2.2. Hence, the time complexity follows from Lemma 1. $\square$

Observe that the above version of multisearch is "read only", that is queries only read the contents of the nodes they are visiting without making any changes. The more general *multisearch problem with changing node values* refers to the case where queries can also change the contents of visited nodes. If several queries attempt to write different values into the same node, we use an associative operator $\times$ (e.g. sum, min, max, or, and, not, ...) to determine the node's value.

Algorithm 6 is easily generalized to solve the multisearch problem with changing node values. We insert after Steps 2 and 6 a procedure which combines the results written into the different copies of the same node of T (residing on different processors). This is easily performed in time $O(\frac{n \log n}{p} + T_s(n,p))$ by sorting all tree nodes such that the $O(p)$ copies of the same node of T reside in the same processor, and executing the associative operator $\times$ locally on those copies.

Corollary 2 *The multisearch problem with changing node values for a balanced searchtree of size $O(n)$ and fixed degree k, and a set of $m = O(n)$ search queries, can be solved on a p-processor coarse grained multicomputer with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $\frac{n}{p} \geq p$, in time $O(\frac{n \log n}{p} + T_s(n,p))$.*

In the remainder of this section, we study some applications of our multisearch algorithm.

A *segment tree*, originally introduced by Bently [4], is a data structure designed for storing line segments. The segment tree T for a set of n line segments is a complete binary tree with $2n$ leaves (for ease of description let $2n$ be a power of 2) representing the x-coordinates of the segments' endpoints in sorted order. For each internal node v, its "node list" is the set of all line segments s with the property that the projection of s onto the x-axis contains all x-coordinates of leaves of the subtree rooted at v but but it does not contain all leaves of the subtree rooted at the direct ancestor of v in T. See Preparata and Shamos's book [16] for details and a catalog of applications. It is easy to see that the total size of all "node lists" is $O(nlogn)$. The *linear size segment tree* is obtained by storing for each vertex v only the *size* of its "node list". As outlined e.g. in [16], a variety of problems (e.g. computing the union of rectangles) can be solved through $O(n)$ queries (insert, delete, search) executed on a linear size segment tree.

Obviously, a parallel segment tree search, with $O(n)$ queries executed in parallel, reduces to a multisearch procedure. As shown in [7], the construction of a linear size segment tree can also be reduced to a constant number of multisearch procedures on a complete binary tree.

Corollary 3 *The linear size segment tree construction problem for n line segments as well as the parallel linear size segment tree search problem for $O(n)$ search queries can be solved on a p-processor coarse grained multicomputer with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $\frac{n}{p} \geq p$, in time $O(\frac{n \log n}{p} + T_s(n,p))$.*

Let S be a set of r pairwise disjoint m-vertex polygons. The *uni-directional separability* problem [8] consists of determining all directions d such that S is separable by a sequence of r translations in direction d (one for each polygon). The *multi-directional separability* problem [8] asks if S is separable by a sequence of r translations in different directions. We refer the reader to [8] for more details on these problems. The solutions presented in [8] are based on multiple searches on a modified segment tree and another tree data structure called *wedge tree*. It is not a complicated exercise to follow the steps of the algorithms in [8] and observe that each step can be parallelized for a CGM(n, p) with $n = O(r^2 + rm)$, $\frac{n}{p} \geq p$, using the tools presented in this section.

Corollary 4 *The uni-directional and multi-directional separability problems for r pairwise disjoint m-vertex polygons can be solved on a p-processor coarse grained multicomputer with arbitrary interconnection network and local memories of size $O(\frac{n}{p})$, $n = O(r^2 + rm)$ and $\frac{n}{p} \geq p$, in time $O(\frac{r^2(m+\log r)}{p} + T_s(r^2, p))$.*

9 Experimental Results

To demonstrate the practical relevance of our scalable CGM algorithms, we implemented the lower envelope algorithm for (possibly intersecting) line segments (Section 6) on a CM-5 with 32 processors and on an Intel iPSC/860 with 8 processors.

We first discuss our CM-5 implementation. Our code is less than 400 lines long and is highly optimized. The sequential local computation of the lower envelope consists of $\log n$ phases which merge pairs of envelopes, starting with envelopes consisting of a single segment each. For parallel sorting we used a merge sort available as public domain code from [18]. The *total exchange* operation was implemented by using sort (see Section 2.2). *Multinode broadcast* was available as a CM-5 system call, but *partial sum* had to be re-implemented because the available system call did not handle n/p data per processor. Each line segment was implemented as a structure of 4 double precision floats. The implementation did not make use of the CM-5's vector units. The timings were made under time sharing and the installation of the machine is experimental. Figure 4 describes therefore only the asymptotic behavior of our algorithm.

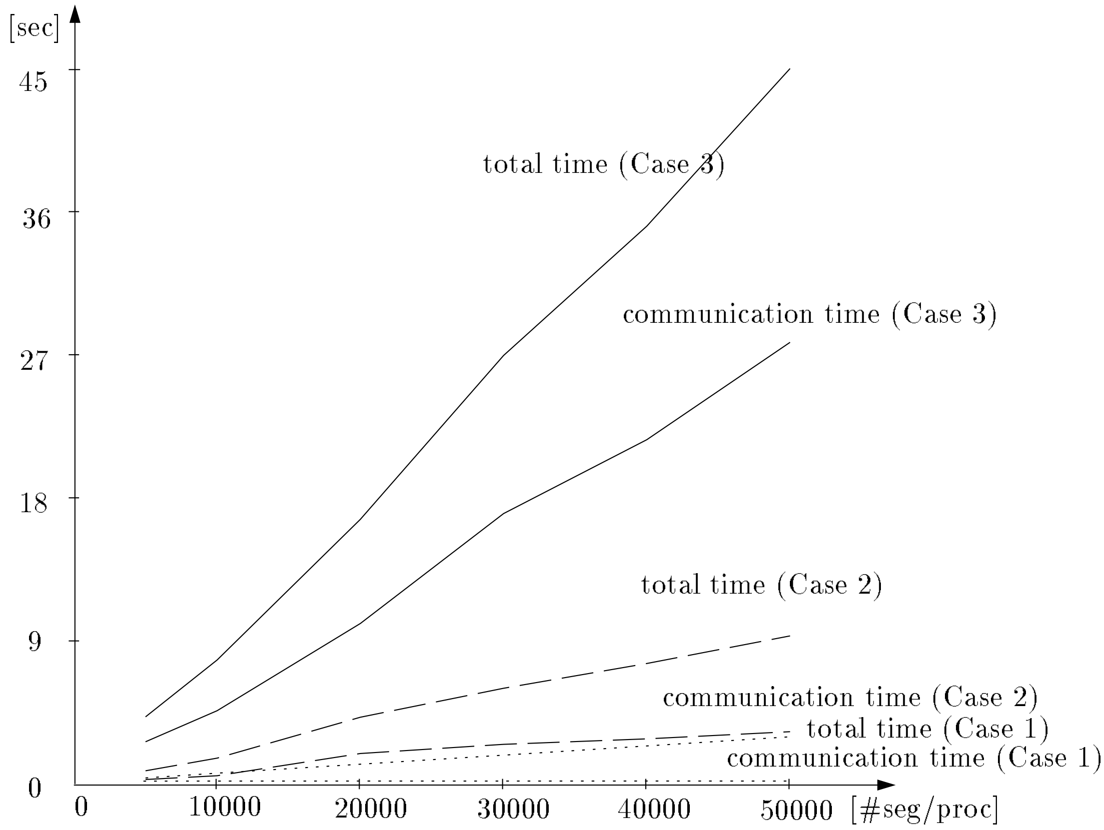

Figure 4: CM-5 running times of the lower envelope algorithm.

The two bottom curves in Figure 4 labeled "total time (Case 1)" and "communication time (Case 1)" describe the running time of our lower envelope algorithm applied to random line segments in a unit square. The two curves show the total running time and the time spent on communication only, respectively, depending on the number of line segments per processor.

The estimated speedup is about 15, i.e. a little less than half of the possible linear speedup (32). An exact measurement was not possible due to the memory limitation on a single processor which did not allow us to run the above mentioned sequential lower envelope code for the entire data set. We hence extrapolated the sequential times for fewer line segments. The speedup is essentially determined by the fact that the algorithm uses two rounds of local lower envelope computation.

Recall that the size of the lower envelope can range between 1 and $O(n\alpha(n))$. In our experiments we observed that for sets of random line segments the sizes of the lower envelopes created in Steps 1 and 5 of Algorithm 4 are very small compared to the initial line segment set. This drastic data reduction has a large positive impact on the running time and is one of the reasons why our algorithm is so extremely fast. While this massive data reduction is, in practice, a nice property of our algorithm, we were also interested in its running time without this additional advantage.

Therefore we also applied our algorithm to several non random line segment sets were the output size was considerably larger. Two cases were considered, which are referred to in Figure 4 as "Case 2" and "Case 3".

Case 2: We selected only segments (inside the unit square) of a fixed very short length. The larger the line segment set generated, the smaller was the chosen length, such that the product of the length and the number of line segments was always a constant $c = 10,000$.

Case 3: We selected n segments such that the lower envelope had a size of approximately $3n$. An arrangement of 3 segments with a lower envelope of 6 segments was replicated and the entire arrangement intersected by a long horizontal segment.

The timings in Figure 4 confirm the theoretical analysis. For fixed p, Theorem 4 implies that the total time grows proportional to $O(n \log n)$ while the communication time grows proportional to $O(n)$. As $\log n$ grows very slowly, $O(n \log n)$ and $O(n)$ are usually very similar in practice. This exactly what we observe in Figure 4: the total time and the communication time are essentially linear in n, with different constants.

Also the absolute times were interesting in practice. Note that, even for Case 3 (no data reduction), the lower envelope for $32 \times 50,000 = 1,600,000$ line segments was reported in 45 sec. As indicated above, these timings were obtained in a time sharing environment and on an experimental installation. Hence, we can expect further improvements.

The results for Case 3 also give a clue about the running times of the other algorithms studied in this paper. They do not have the above mentioned data reduction property. Otherwise, all other algorithms have a similar structure, except for the fact that they may use up to twice as many global sorts and sometimes larger records to be sorted. This leads us to conjecture that the communication times for those algorithms will have, within a small constant factor, a similar growth rate. Note that the sequential algorithms for the other problems have time complexities that are also larger, by small constant factors, than the sequential lower envelope computation time.

We also implemented the lower envelope algorithm on an Intel iPSC/860 hypercube with 8 processors. We used a public domain sorting code for the iPSC and Intel's standard FORTRAN compiler. (We did not have available a high performance i860 compiler.) The sequential lower envelope code was based on a plane sweep algorithm. The results for random line segment sets in a unit square (Case 1) are shown in Figure 5.

Both experiments are not really comparable since the programming was done by different people using different code for sorting and local computations. However, they illustrate nicely a big architectural difference between the CM-5 and the iPSC/860. Within the range for n considered in our experiments, the communication time for the iPSC is a large constant with considerable influence on the total time. The effect is that the total time is essentially constant for $\frac{n}{p} \leq 1024$ and shows linear linear grows only for larger values of n. This large communication time even for small values of n is due to the fact that the iPSC was designed primarily for sending large data packets but, on the other hand, needs considerably more

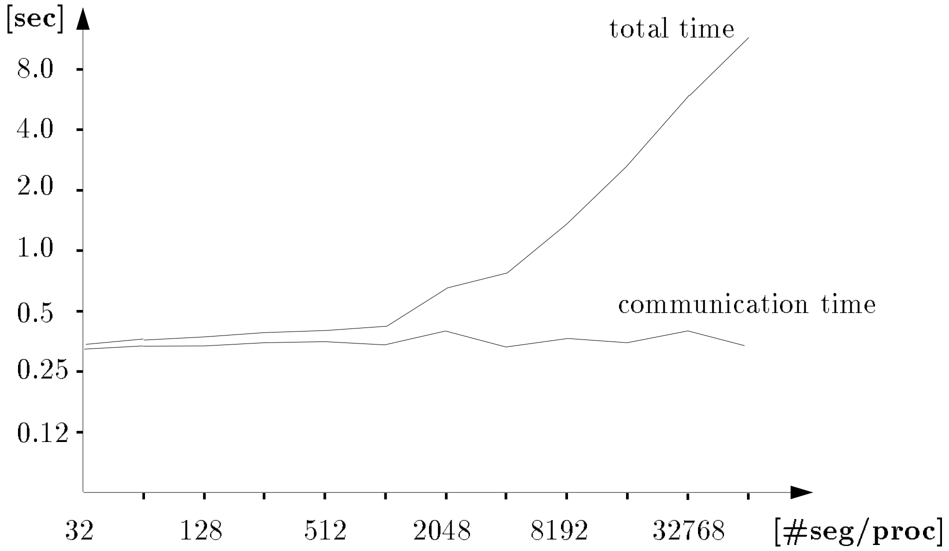

Figure 5: iPSC/860 running times of the lower envelope algorithm.

time to initiate a data transfer. It underlines the importance of designing scalable parallel algorithms such that only few large data packets are exchanged, which is one of the properties of all our algorithms presented in this paper.

10 Acknowledgements

The authors would like to thank Anton Saarimaki for his implementation work on the iPSC/860.

References

[1] M.J. Atallah, F. Dehne, R. Miller, A. Rau-Chaplin, and J.-J. Tsay Multisearch techniques for implementing data structures on a mesh-connected computer. *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 204–214, 1991.

[2] M. J. Atallah and J.-J. Tsay. On the parallel-decomposability of geometric problems. *Proc. 5th Annu. ACM Symposium on Computational Geometry*, pages 104–113, 1989.

[3] K.E. Batcher. Sorting networks and their applications. *Proc. AFIPS Spring Joint Computer Conference*, pages 307–314, 1968.

[4] J.L. Bentley. Algorithms for Klee's rectangle problems. Carnegie-Mellon Univ., Penn., Dept. of Comp. Sci. Unpublished notes, 1977.

[5] D.P. Bertsekas and J.N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[6] R. Cypher and C.G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. *Proc. ACM Symposium on Theory of Computing*, pages 193–203, 1990.

[7] F. Dehne and A. Rau-Chaplin. Implementing data structures on a hypercube multiprocessor and applications in parallel computational geometry. *Journal of Parallel and Distributed Computing*, Vol. 8, No. 4, pages 367–375, 1990.

[8] F. Dehne and J.-R. Sack. Translation separability of sets of polygons. *The Visual Computer 3*, pages 227–235, 1987.

[9] M.T. Goodrich, J.J. Tsay, D.E. Vengroff, and J.S. Vitter. External-memory computational geometry. *Proc. 34th IEEE Symposium on Foundations of Computer Science*, pages 714–723, 1993.

[10] *Grand Challenges: High Performance Computing and Communications.* The FY 1992 U.S. Research and Development Program. A Report by the Committee on Physical, Mathematical, and Engineering Sciences. Federal Council for Science, Engineering, and Technology. To Supplement the U.S. President's Fiscal Year 1992 Budget.

[11] R. I. Greenberg and C. E. Leiserson. Randomized routing on fat-trees. *Advances in Computing Research*, 5, pages 345–374, 1989.

[12] J. Hershberger. Finding the upper envelope of n line segments in $O(n \log n)$ time. *Information Processing Letters* 33, pages 169–174, 1989.

[13] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[14] J.M. Marberg and E. Gafni. Sorting in constant number of row and column phases on a mesh. *Proc. Allerton Conference on Communication, Control and Computing*, pages 603-612, 1986.

[15] M.H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *Journal on Computer System Science*, 23, pages 166–204, 1981.

[16] F.P. Preparata and M.I. Shamos. *Computational Geometry: an Introduction.* Springer-Verlag, New York, NY, 1985.

[17] J.H. Reif and L.G. Valiant. A Logarithmic Time Sort for Linear Size Networks. *Journal of the ACM*, Vol. 34, 1, pages 60–76, 1987.

[18] A. Tridgell and R.P. Brent. An implementation of a general-purpose parallel sorting algorithm. *CS Laboratory, Australian National University. Technical Report TR-CS-93-01, 1993.*

[19] J.-J. Tsay, Ph.D. Thesis, Purdue University.

[20] J. van Leeuwen and D. Wood. The measure problem for rectangular ranges in d-space. *Journal of Algorithms*, 2, pages 282–300, 1981.