External-Memory Computational Geometry

(Preliminary Version)

Michael T. Goodrich[*] — The Johns Hopkins University — goodrich.cs.jhu.edu

Jyh-Jong Tsay[†] — National Chung Cheng University — jjt@cs.ccu.edu.tw

Darren Erik Vengroff[‡] — Brown University — dev@cs.brown.edu

Jeffrey Scott Vitter[§] — Duke University — jsv@cs.duke.edu

Abstract

In this paper, we give new techniques for designing efficient algorithms for computational geometry problems that are too large to be solved in internal memory, and we use these techniques to develop optimal and practical algorithms for a number of important large-scale problems. We discuss our algorithms primarily in the context of single processor/single disk machines, a domain in which they are not only the first known optimal results but also of tremendous practical value. Our methods also produce the first known optimal algorithms for a wide range of two-level and hierarchical multilevel memory models, including parallel models. The algorithms are optimal both in terms of I/O cost and internal computation.

1 Introduction

Input/Output (I/O) communication between fast internal memory and slower secondary storage is the bottleneck in many large-scale information-processing applications, and its relative significance is increasing as parallel computing gains popularity. In this paper we consider the important application area of computational geometry and develop several paradigms for optimal geometric computation using secondary storage.

Large-scale problems involving geometric data are ubiquitous in spatial databases [24,32,33] , geographic information systems (GIS) [10,24,33], constraint logic programming [19,20], object oriented databases [38], statistics, virtual reality systems, and computer graphics [33]. As an example, NASA's soon-to-be petabyte-sized databases are designed to facilitate a variety of complex geometric queries [10]. Important operations on geometric data include range queries, constructing convex hulls, nearest neighbor calculations, finding intersections, and ray tracing, to name but a few.

1.1 Our I/O model

In I/O systems, data are usually transferred in units of *blocks*, which may consist of several kilobytes. This blocking takes advantage of the fact that the seek time is usually much longer than the time needed to transfer a record of data once the disk read/write head is in place. An increasingly popular way to get further speedup is to use many disk drives and/or many CPUs working in parallel [12,13,18,25,28,36]. We model such systems, examples of which are shown in Figure 1, using the following four parameters:

$$\begin{aligned} M &= \#\text{ items that can fit in internal memory;}\\ B &= \#\text{ records per block;}\\ P &= \#\text{ CPUs (internal processors);}\\ D &= \#\text{ disk drives.}\end{aligned}$$

For the problems we consider, we define three general parameters:

$$\begin{aligned} N &= \#\text{ items or updates in the problem instance;}\\ K &= \#\text{ query operations in the problem instance;}\\ T &= \#\text{ items in the solution to the problem.}\end{aligned}$$

[*]This research was supported in part by the National Science Foundation under Grants CCR–9300079 and IRI–9116843, and by the NSF and DARPA under Grant CCR–8908092.

[†]This research was supported by the National Science Council under grant NSC81–0408–E–194–508, Republic of China.

[‡]This research was supported in part by Army Research Office grant DAAL03–91–G–0035 and by National Science Foundation grant DMR–9217290. Author is currently visiting Duke University, e-mail: dev@cs.duke.edu

[§]This research was supported in part by National Science Foundation grant CCR–9007851 and by Army Research Office grant DAAL03–91–G–0035.

Figure 1: (a) The parallel disk model. Each of the D disks can simultaneously transfer B records to and from internal memory in a single I/O. The internal memory can store $M \geq DB$ records. (b) Multiprocessor generalization of the I/O model in (a), in which each of the $P = D$ internal processors controls one disk and has an internal memory of size M/P. The P processors are connected by some topology such as a hypercube or an EREW PRAM and their memories collectively have size M.

We will assume that $M < N$, $1 \leq P \leq M/\log M$, and $1 \leq DB \leq M/2$. The measures of performance that we would like to minimize simultaneously are the number of I/Os and the internal computation time. The relevant terms that enter the formulæ for the I/O bounds are often in units of blocks, such as N/B, M/B, and so on. For that reason we define the following shorthand notation:

$$\nu = \frac{N}{B}, \qquad \mu = \frac{M}{B}, \qquad \kappa = \frac{K}{B}, \qquad \tau = \frac{T}{B}.$$

In order to get across our techniques in the minimum space, we illustrate our results in this paper to the special case of the I/O model in which $P = 1$ and $D = 1$. Even in this simplified model our results are extremely significant, as $P = 1$ and $D = 1$ accurately models the vast majority of I/O systems currently installed and being produced, yet no optimal algorithms were previously known for the problems we discuss. Additionally, our results are optimal in the general I/O model and in the parallel hierarchy models. In particular, in the I/O model, using P processors reduces the internal computation time by a factor of P and using D disks reduces the number of I/O steps by a factor of D. This is discussed in greater detail in Section 6 and in the full version of this paper.

1.2 Our results

In this paper we present a number of general techniques for designing external-memory algorithms for computational geometry problems. These techniques include the following:

- *distribution sweeping*: a generic method for externalizing plane-sweep algorithms;
- *persistent B-trees*: an off-line methods for constructing an optimal-space persistent version of the B-tree data structure. For batched problems this gives a factor of B improvement over the generic persistence techniques of Driscoll *et al.* [11]);
- *batch filtering*: a general method for performing K simultaneous external-memory searches in data structures that can be modeled as a planar layered dags;
- *external marriage-before-conquest*: an external-memory analog to the well-known technique of Kirkpatrick and Seidel [22] for performing output-sensitive hull constructions.

We apply these techniques to derive optimal external-memory algorithms for the following fundamental problems in computational geometry: computing the pairwise intersection of N orthogonal segments, answering K range queries on N points, constructing the 2-d and 3-d convex hull of N points, performing K point location queries in a planar subdivision of size N, finding all nearest neighbors for a set of N points in the plane, finding the pairwise intersections of N rectangles, computing the measure of the union of N rectangles, computing the visibility of N segments from a point, performing K ray-shooting queries in CSG models of size N, as well as several geometric dominance problems. Our results are summarized in the following theorem, individual parts of which are discussed in the remaining sections of the paper.

Theorem 1.1: *Each of the problems mentioned in the preceding paragraph can be solved in external memory using $O((\nu + \kappa)\log_\mu \nu + \tau)$ I/Os. If D disks are used in parallel, the number of I/Os required can be reduced by a factor of D.*

For problems in which there are no queries as part of the problem instance, we use $K = 0$ (and thus $\kappa = 0$); if the output (solution) size is fixed, we use $T = 1$ (and thus $\tau = 1/B = o(1)$).

2 Distribution sweeping

The well-known *plane sweep* paradigm [30] is a powerful approach for developing computational geometry algorithms that are efficient in terms of internal computation. In this section we develop a new plane sweep approach that for the first time achieves optimal I/O performance (and a subsequent improvement in practice) for a large number of large-scale off-line problems in computational geometry. Specific examples that we explore include orthogonal segment intersection, batched range queries, computing all nearest neighbors, rectangle intersection and union computations, visibility among a set of non-intersecting line segments, and 3-d maxima.

We assume we are given a sequence $\sigma = \sigma_1\sigma_2\ldots\sigma_N$, such that each σ_i is an update operation of the form $insert(x)$ or $delete(x)$, where each such x is taken from a known total order ω. The sequence σ corresponds to the sequence of update operations during the sweep. In addition, we are given a sequence $\rho = \rho_1\rho_2\ldots\rho_K$, such that each ρ_i is a query operation $query(j)$, which is defined as some kind of search in a search tree defined on ω by performing the operations $\sigma_1\sigma_2\ldots\sigma_j$. We assume, without loss of generality, that the ρ_i's are sorted by their j arguments. The problem is to determine the answer to each ρ_i query.

One obvious external-memory solution to this problem is to implement the search tree as a dynamic B-tree [6,9] and to perform the queries in ρ in an on-line fashion while performing the updates in σ. Unfortunately, this requires $\Theta((N+K)\log_\mu \nu) = \Theta(B(\nu+\kappa)\log_\mu \nu)$ I/O operations in the worst case, which is prohibitive. Previous work using lazy batched updates on the B-tree yielded algorithms with $O((\nu+\kappa)\log_2 \nu)$ I/Os [34].

In Sections 2.1 and 2.2 we show how to perform the queries using only $O((\nu+\kappa)\log_\mu \nu)$ I/Os, which is optimal. The lower bound follows by a simple reduction from the sorting problem, which has the same I/O bound as a lower bound [3]. Our new method uses an off-line top-down implementation of the sweep, which in turn is based upon a novel application of the subdivision technique used by in the "distribution sort" algorithms of [3,27,37]. It is for this reason that we refer to our technique as *distribution sweeping.*

2.1 An example: orthogonal segment intersection reporting

Before discussing the distribution sweeping as a general technique, let us begin with a simple example illustrating the main ideas involved. The example we will use is the problem of reporting all intersecting pairs from a set of N orthogonal line segments. This problem is important in graphics and VLSI design systems.

We define the sweep in terms of a vertical sweep of the plane by a horizontal line. The x-coordinates of the vertical segments define the total order ω, and the y-coordinates of their endpoints define the sweep-line update events in σ. The y-coordinates of the horizontal segments define the sweep-line query events in ρ.

Initially, we use an optimal sorting algorithm so that endpoints of all segments are in two sorted lists, one sorted by x and the other by y. We now partition the points into $\lceil \mu \rceil$ vertical strips γ_i. Now the sweeping begins, progressing from top to bottom. As we sweep, we will report some of the segment intersections and distribute data to recursive subproblems that we can solve to find the rest. When the top endpoint of a vertical segment is encountered, the segment is inserted into an *active list* A_i associated with the strip γ_i in which the segment lies, and later when the bottom endpoint is encountered, the segment is deleted from A_i. When we encounter a horizontal segment R, we consider the strips that R passes completely through and report all the vertical segments in the active lists of those strips. Note that horizontal segments are only distributed to the two strips containing their endpoints, thus at each level of recursion each segment is represented only twice. Once the number of points in a recursive subproblem falls below M, we simply solve the problem in main memory. This process is illustrated in Figure 2.

Insertions and vertical segments can be processed efficiently using blocks. With the exception of deleting segments from active lists, the total number of I/Os performed by this method is optimal $O(\nu\log_\mu \nu + \tau)$, where $\tau = T/B$ and T is the number of intersections reported. If "vigilant" deletion is used to delete each segment as soon as the sweepline reaches the bottom endpoint, a nonoptimal $O(N) = O(B\nu)$ term is added to the I/O bound. Instead we use the following lazy approach: For each strip, we maintain a stack of insertions processed so far. When a new segment is inserted, we simply add it to the stack. We keep all but the most recently added B elements of this stack in blocks of size B in external memory. When we are asked to output the active list, we scan the entire stack, outputting the segments still current and removing the segments whose deletion time has passed. A simple amortization argument shows that this method achieves the bound of Theorem 1.1.

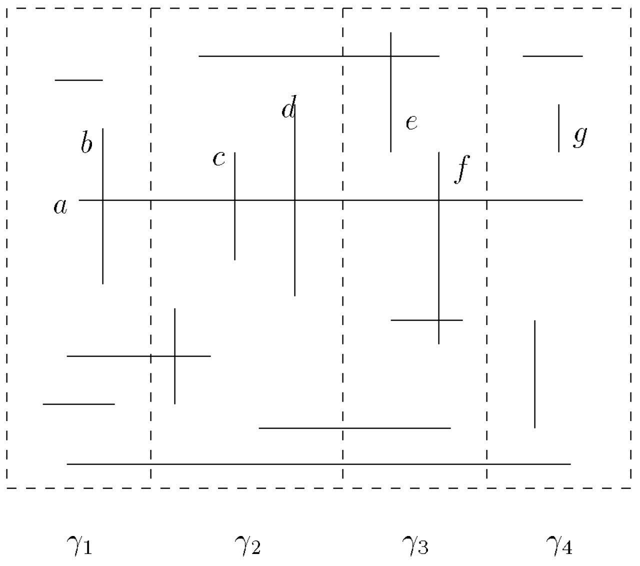

Figure 2: Distribution sweeping for orthogonal segment intersection. Suppose the sweep line (moving down from the top) has reached horizontal segment a. At this point the active lists for the four strips are $A_1 = \{b\}$, $A_2 = \{c, d\}$, $A_3 = \{e, f\}$, and $A_4 = \{g\}$. Note that this assumes lazy deletion. At this point, the intersections of a and c, d, and f are reported because a fully spans γ_2 and γ_3. Note that the intersection of a and b will not be reported until the problem is solved recursively on γ_1. e and g are now deleted from A_3 and A_4 respectively. Finally, we distribute the points to the subproblems corresponding to the strips.

2.2 General distribution sweeping

In this section we present the distribution sweeping technique at a high level. Our method is based upon the following "distribution lemma," which we will use to distribute input data into recursive subproblems.

Lemma 2.1: [3,37] *Let $\mathcal{S}$ be a set of N elements taken from some total order ω, and let $\mathcal{S}$ be stored in $\nu = N/B$ blocks in external memory. Then, using only $O(\nu)$ I/O operations, $\mathcal{S}$ can be partitioned into $s = \lceil\sqrt{\mu}\rceil = \lceil\sqrt{M/B}\rceil$ subsets $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_s$ such that, for all i, $\frac{1}{2}\frac{N}{s} \le |\mathcal{S}_i| \le \frac{3}{2}\frac{N}{s}$ and each element in $\mathcal{S}_i$ is less than each element in $\mathcal{S}_{i+1}$.*

Note that this lemma divides the input into $\lceil\sqrt{\mu}\rceil$ subsets, whereas in the example in the previous section we divided the input into $\lceil\mu\rceil$ strips. The difference is that this lemma does not require the input to be sorted by the ω ordering before it is partitioned; in fact this lemma was originally proven for use in distribution sorting algorithms. Asymptotically, partitioning the input in this manner does not affect the overall running time of an algorithm since it increases the depth of recursion by only a constant factor of 2.

We make use of this lemma to process the updates σ and the queries ρ as follows: We first merge σ and ρ into a single list ordered by index. This can be done in $O(\nu)$ I/Os by a simple merging procedure. Moreover, during this same merging procedure we can construct the set $\mathcal{X}$ of all elements referenced in σ (possibly with duplicates). We then apply Lemma 2.1 to partition $\mathcal{X}$ into $\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_s$, by the ω ordering. Since the operations in σ and ρ are defined for a tree structure ordered by ω, this distribution immediately implies that each σ_i and ρ_i can be decomposed into suboperations $\sigma_{i,1}, \sigma_{i,2}, \ldots, \sigma_{i,s}$ and $\rho_{i,1}, \rho_{i,2}, \ldots, \rho_{i,s}$, such that $\sigma_{i,j}$ (resp., $\rho_{i,j}$) is defined on $\mathcal{X}_j$. The specific definitions of these suboperations will, of course, depend upon the application. Note that in the example given in Section 2.1 they were recursively solving the problem in the strips γ_i. Since σ and ρ are defined for a tree structure ordered by ω, only $O(1)$ suboperations $\sigma_{i,j}$ (resp., $\rho_{i,j}$) involve a recursive search in $\mathcal{X}_j$, for any i. The other suboperations for $\mathcal{X}_j$ involve an update or query that can be performed by a single scan (ordered by i) through these non-recursive $\sigma_{i,j}$ and $\rho_{i,j}$ operations. Since s is $O(\sqrt{\mu})$ we may perform these non-recursive suboperations, as well as construct an ordered list of each sequence of recursive suboperations, by a single pass through the combined σ-ρ list. We form all the recursive sublists simultaneously in main memory using s "buckets" of size $\sqrt{MB} \ge B$. As soon as a bucket fills, we "empty" its contents out to external memory. During this pass, for each j, we also maintain an *active* bucket in main memory that stores information (such as counts, sums, or pointers into external memory) for the elements in $\mathcal{X}_j$ present after performing the operations on the elements of $\mathcal{X}_j$ up to the current one (a $\sigma_{i,j}$ or $\rho_{i,j}$). The active buckets are used to answer the non-recursive suboperations. Having performed all the non-recursive suboperations, we then recursively perform each sequence of recursive suboperations in turn. A single pass requires only $O(\nu)$ I/Os and results in s subsequences made up of $O(\nu/s)$ blocks each, which requires only $O(\nu)$ blocks for all the subsequences combined. Thus, the total number of I/Os needed is $O(\nu \log_s \nu) = O(\nu \log_\mu \nu)$.

2.3 Other applications of distribution sweeping

Though space precludes a full exposition, the distribution sweeping method can be used to solve a number of other off-line problems in computational geometry that are traditionally solved by plane sweep techniques. The resulting algorithms use an optimal $O((\nu + \kappa)\log_\mu \nu + \tau)$ I/Os. Problems in this category include batched range queries, finding all nearest neighbors, computing the visibility from a point in the plane, finding pairwise rectangle intersections, computing the measure of a union of rectangles, and the 3-d maxima problem. These problems are discussed in greater detail in the full version of this paper.

3 Persistent B-trees

The B-tree data structure [6,9] is a fundamental structure for maintaining a dynamically-changing dictionary in external memory. In some cases, however, it may be advantageous to be able to access previous versions of the data structure. Being able to access such previous versions is known as *persistence*, and there exist very general techniques for making most data structures persistent [11]. Persistence can be implemented either in an on-line fashion (i.e., where the tree updates are coming on-line) or in an off-line fashion (i.e., where one is given the sequence of tree updates in advance).

For the on-line case, the method of Driscoll *et al.* [11] can be applied to hysterical B-trees as described by Maier and Salveter [26]. Since it is on-line, this structure requires $O(N \log_\mu \nu)$ I/Os to construct, which is optimal in an on-line setting. Unfortunately, this is a factor of B away from optimal for the sort of batch geometric problems we would like to consider. For these we need an off-line strategy that requires only $O(\nu \log_\mu \nu)$ I/Os. In the following section we describe just such a method.

3.1 Off-line persistence

In the off-line case we can build a persistent tree by the distribution sweep method. We slightly modify our application of distribution sweeping for this construction, however, in that we follow the recursive calls on the sequences of suboperations by a non-recursive "merge" step.

We begin by applying the Lemma 2.1 to divide the set $\mathcal{X}$ of elements mentioned in σ into s groups of size roughly N/s each, where $s = \lceil\sqrt{\mu},\rceil$. This, of course, divides σ into s subsequences, one for each group. We then recursively construct a persistent data structure for each subsequence. Each such recursive call returns a list of "roots" of s-way trees, each of which is marked with a time stamp that represents the index in σ when this root was created. We mark every sth element in each list as a "bridge" element and we merge these bridge elements into a single list $\mathcal{Y}$. We store pointers from each element $y \in \mathcal{Y}$ to all its bridge predecessors in the recursively-constructed lists. The list $\mathcal{Y}$, together with these pointers, defines the roots of the persistent structure. Since we only choose every sth element from each list as a bridge, it is easy to see that total space needed is $O(\nu)$ blocks, and the depth of the resulting (layered dag) persistent structure is $O(\log_\mu \nu)$.

A search in the past, say at time position i, begins by locating the root active for time i and searching down in the structure from there, always searching in nodes whose time stamp is the largest time stamp less than or equal to i. Performing only one such search would not be an efficient strategy, however, unless $s = \sqrt{\mu}$ is $O(B)$. Nevertheless, as we show in the next section, this is a very efficient data structure (e.g., for point location) if it is searched using the batched filtering technique.

4 Batch filtering

In this section we demonstrate how, for many query problems in computational geometry, we can represent a data structure of size N in ν disk blocks in such a way that K constant sized output queries of the data structure can be answered in $O((\nu + \kappa)\log_\mu \nu)$ I/O operations. Because we represent the data structure as a dag through which the K queries filter down from source to sinks, we call this technique *batch filtering*.

Given a data structure that supports queries, we can often model the processing of a query as the traversal of a decision dag isomorphic to the data structure. We begin at a source node in the dag, and at each node we visit, we make a decision based on the outcome of comparisons between the query value and some number d of values stored at the node. We then make a decision as to which of the node's $O(d)$ children to visit next. This process continues until we reach a sink in the dag, at which point we report the outcome of the query.

By restricting the class of such dags we are willing to consider, we are able to prove the following lemma, which will serve as a building block for optimal algorithms to solve a number of important geometric problems.

Lemma 4.1: *Let $G = (V, E)$ be a planar layered decision dag with a single source such that the maximum out degree of any node is μ. Let the graph be represented in ν blocks, with the nodes ordered by level and the nodes within a level ordered from left to right. Let*

$N = |V|$ and let h be the height of G. We can filter K inputs through G in $O(\nu + h\kappa)$ I/O operations.

Proof Sketch: We traverse the levels one by one, sending all K inputs to the i'th level before any are sent to the $i+1$'st. We do this by maintaining two FIFO queues, one for the current level and one for the next level. Each such queue is left to right list of edges between its level and the next one. If less than B inputs traverse an edge then they are explicitly stored in the queue. If B or more traverse the edge, then the queue contains a pointer to a linked list of blocks storing them. Since the graph is planar, there exists an efficiently blocked method of producing one queue form the previous queue. □

Luckily, the restrictions imposed on the type of decision dags we can handle with batch filtering is not too severe. In particular, many computations use decision trees, which clearly constitute a special case of the lemma. Often these trees are binary, but we can divide a binary tree into layers of height $O(\log \mu)$ and then store each node on a layer boundary along with all its descendants in the layer below it as a single node with branching factor μ. This allows us to reduce h by a factor of $O(\log \mu)$ yet still meet the conditions of the lemma. We will see this approach used in solving subproblems of the 3-d convex hull problem in Section 5.3.

Another way of using batch filtering, as is to be seen in Section 4.1, is by structuring more complicated decision dags as recursive constructions to get around the planarity restrictions of the lemma.

4.1 Application: multiple-point planar point location

Planar point location is one of the fundamental problems of computational geometry. In the version of the problem considered here, we are given a monotone planar decomposition having N vertices, and a series of K query points. For each query point, we are to return the identifier of the region in which it lies. In main memory, this problem can be solved in optimal time $O((N+K)\log N)$ using fractional cascading [7,8]; $O(N \log N)$ is spent on preprocessing and $O(K \log N)$ is needed to perform the queries.

Tamassia and Vitter [35] have demonstrated a technique by which the fractional cascading used to solve this problem can be implemented in parallel. Their technique can solve our problem in $O((N/p + K)\log_p N)$ time on a PRAM with p processors. We can use a method based on their construction, but using μ in place of p to get a data structure that looks like a μ-ary tree augmented with catalogs. Clearly we can apply the technique of Lemma 4.1 to the main tree, but the bridge pointers connecting the catalogs make the dag non-planar. To get around this, we note that as queries traverse the edges between nodes in the main tree, they are ordered by the catalog values they query. This ordering is established at the root of the data structure, where a μ-ary tree is used to locate the queries in the first catalog. By relying on this ordering, we can efficiently process the queries that arrive at each node of the main tree. The overall complexity of this technique is thereby maintained at $O((\nu + \kappa)\log_\mu \nu)$. Full details of the construction appear in the full version of the paper.

5 Convex hull algorithms

The convex hull problem is that of computing the smallest convex polytope completely enclosing a set of N points in d-dimensional space. This problem has important applications ranging from statistics to graphics to metallurgy. In this section we will examine the problem in external memory for two and three dimensions. The three-dimensional case is particularly interesting because of the number of two-dimensional geometric structures closely related to it, such as Voronoi diagrams and Delaunay triangulations. In fact, by well-known reductions [17], our 3-d convex hull algorithm immediately gives external-memory algorithms for planar Voronoi diagrams and Delaunay triangulations with the same I/O performance.

In main memory the lower bound for computing the convex hull of N points in dimension $d > 1$ is $\Omega(N \log N)$ in the worst case [30]. In secondary memory, this bound becomes $\Omega(\nu \log_\mu \nu)$. In this section we give optimal algorithms that match this lower bound. For the two-dimensional case we show how to beat this lower bound for the case when the output size T is much smaller than N (in the extreme case, $T = O(1)$). We develop an output-sensitive algorithm based upon an external-memory version of the marriage-before-conquest paradigm of Kirkpatrick and Seidel [22].

Our 3-d convex hull is somewhat esoteric, so we also describe a simplified version which, although not optimal asymptotically, is simpler to implement, and will be faster for the vast majority of practical cases.

5.1 A worst-case optimal two-dimensional convex hull algorithm

For the two-dimensional case, a number of main memory algorithms are known that operate in optimal time $O(N \log N)$ [30]. A simple way to solve the problem optimally in external memory is to modify one of the main memory approaches, namely Graham's scan [16]. Graham's scan requires that we sort the points, which can be done in $O(\nu \log_\mu \nu)$ I/O operations, and then scan linearly through them, at times backtracking, but only over each input point at most once. Clearly this scanning stage can be accomplished in $O(\nu)$ I/O operations, so the overall complexity of the algorithm is $O(\nu \log_\mu \nu)$.

5.2 An output-sensitive two-dimensional convex hull algorithm

If the output size T is significantly smaller than N (for example, T can be $O(1)$) then we can do better than the Graham scan approach. In this section we show how to construct a two-dimensional convex hull using a number of I/Os that is output-size sensitive in a stronger sense than any of the algorithms discussed thus far. Note that when $T = o(N)$, we actually do better than Theorem 1.1 indicates. Our results are optimal, as stated in the following theorem.

Theorem 5.1: *Let S be a set of N points in the plane whose convex hull has T extreme points. If μ^ϵ is $O(B)$ for some constant $\epsilon > 0$ then the convex hull of S can be computed in $O(\nu \log_\mu \tau)$ I/Os, which is optimal.*

We omit details in this preliminary version, but the main idea of our method is as follows: First, we observe that we may restrict our attention to the upper hull (i.e., edges with normals with positive y-components) without loss of generality. We use the Lemma 2.1 to divide the set of input points into $s = \lceil \sqrt{\mu} \rceil$ buckets divided by vertical lines. We then use an external-memory implementation of a method of Goodrich [15] for combining prune-and-search bridge finding [22] with the Graham scan technique [16] to find all the upper hull edges intersecting our given vertical lines. Our implementation uses $O(\nu)$ I/Os. Given these hull edges we may then recurse on any buckets that are not completely covered by the hull edges we just discovered. Our analysis is based on the fact that in any such divide step we either reduce the number of points under consideration by a constant fraction or we will discover $\Theta(s)$ hull edges (possibly both). If μ^ϵ is $O(B)$ for some constant $\epsilon > 0$, this implies that the total number of I/Os is $O(\nu \log_\mu \tau)$, which is optimal for any value of T.

5.3 Three-dimensional convex hulls

Even in main memory, sweep plane algorithms fail to solve the 3-d convex hull problem, and we must resort to more advanced divide and conquer approaches [29]. One idea is to use a plane to partition the points into equally sized sets, recursively construct the convex hull for each set, and then stitch the recursive solutions together in linear time. Unfortunately, we know no way of implementing an algorithm of this type in secondary memory; the problem is that we cannot adequately anticipate all possible paths through the data that might be traversed during the combining phase. Another obstacle is that we need to be able to stitch together $O(\mu^\varepsilon)$ recursive solutions in linear time, rather than just two. If we use any fewer, then the depth of the recursion will not be small enough to give us an optimal algorithm.

In order to get around the problems associated with a merging approach, we use a novel formulation of the distribution method. We consider the dual of the convex hull problem, namely that of computing the intersection of a set of N half spaces all of which contain the origin. Standard geometric duality transformations [30] are used to show the equivalence of convex hull and halfspace intersection. Once we are dealing with the dual problem, we can use a distribution based approach along the lines of that proposed by Reif and Sen for computing 3-d convex hulls in parallel [31].

Let S be a set of N halfspaces all of which contain the origin. Let the boundary of a halfspace $h_i \in S$ be denoted P_i. Suppose we have a subset $S_0 \subset S$ such that $|S_0| = N^\varepsilon$. Let $I_0 = \bigcap_{h_j \in S_0} h_j$. A face of I_0 might have up to N^ε edges. We can reduce this complexity by trangulating each face, which can be done by sorting the vertices of I_0 along a vector not perpendicular to any face and then sweeping a plane along this sorted order. By Euler's law the size of the resulting set of faces is at most $O(N^\varepsilon)$. We can now decompose I_0 into $O(N^\varepsilon)$ cones C_i, each of which has one of these faces as a base and the origin as an apex. An obvious way of distributing the halfspaces into subproblems is to create a subproblem for each cone C_i consisting of finding the intersection of all halfspaces $h_j \in S \backslash S_0$ whose bounding planes P_j intersect C_i. Unfortunately, a given P_j may intersect many cones, so it is not clear that we can continue to work through the $O(\log \log N)$ required levels of recursion without causing a very large blow up in the total size of the subproblems. Luckily, using a form of random sampling called polling and eliminating redundant planes from within a cone prior to recursion [31], we can with high probability get around this problem. (In this discussion, the phrase "with high probability" means with probability $1 - N^{-\alpha}$, for some constant α.)

Algorithm 5.1 is the resulting distribution algo-

```
Halfspace Intersection
Input: A set $S$ of $N$ halfspaces in 3-d space.
Output: The set of all halfspaces $h_i \in S$ whose bounding
planes lie on the boundary of the intersection $\bigcap_{h_j \in S_0} h_j$

1. For $j = 1$ to $\Theta(\log_\mu \nu)$, take a random sample $S_j$ of
   $S$, where $|S_j| = N^\varepsilon$ for a constant $0 < \varepsilon < 1$.
2. Recursively solve the halfspace intersection problem
   on each sample $S_j$, giving a set of solutions $I_j$.
3. Use polling ([31]) to estimate the size of the partition
   of $S - S_j$ that each sample solution $I_j$ will induce.
   Let $S_r$ be the sample whose solution $I_r$ generates the
   smallest such partition.
4. For each cone $C_i$ of $I_r$, compute $R_i$, the set of halfs-
   paces in $S - S_r$ whose boundaries intersect $C_i$.
5. Eliminate redundant planes from each $R_i$, yielding
   $R_i^*$.
6. Recursively solve the halfspace intersection problem
   on each set $R_i^*$.
```

Algorithm 5.1: An algorithm for computing the 3-d convex hull of a set of points..

rithm for computing the intersection of all $h_i \in S$. Step 1 can be completed with $O(\nu \log_\mu \nu)$ I/Os by making a linear pass through S for each sample, as suggested by Knuth [23]. Step 2 consists of recursive calls that will be considered later. In Step 3 we decompose each S_j into cones using a plane sweep. This takes $O((|S_j|/B)\log_\mu(|S_j|/B))$ I/Os. We then take a random sample from $S - S_j$ for each S_j. This takes $O(\nu \log_\mu \nu)$ I/Os. Finally, we solve a tree structured point location problem on all elements of the sample. This is done by batch filtering as described in Section 4. The number of I/O operations needed by Step 4 is $O(\frac{r}{B} \log_\mu \frac{r}{B})$, where $r = \sum_i |R_i|$. In Step 5, redundant planes are eliminated using a variant of the 3-d maxima algorithm from Section 2 and a 2-d convex hull algorithm. Both require $O(\frac{r}{B} \log_\mu \frac{r}{B})$ I/O operations. Finally, Step 6 recursively solves the subproblems.

By methods analogous to the approach of Reif and Sen [31] for the parallel case, we can develop the following recurrence for the running time of our algorithm:

$$T(n) = O(\nu \log_\mu \nu) + T(N^\varepsilon)\log_\mu \nu + \sum_i T(|R_i|).$$

The first term on the right-hand side is the I/O cost for sampling and partitioning, the second term is the I/O cost for sorting the samples, and the last term is for the recursive calls. In the recurrence the $|S_i|$ terms are actually random variables. It suffices to use Karp's method for solving probabilistic recurrence relations [21] to get the optimal solution $T(n) = O(\nu \log_\mu \nu)$ with high probability.

The "distribution" approach used here is different from those of the distribution sort algorithms for the various I/O and memory hierarchy models [3,27,37], but has the same asymptotic I/O complexity. In the distribution sorting, there are two sets of recursive calls rather than one, but the time to partition is faster.

If desired, the randomization in our algorithm can be removed by an external memory implementation of the technique in [14]. Details are omitted for brevity.

In the full version of this paper we demonstrate how, for problems of any reasonable practical size, we can improve upon this algorithm by using samples of size μ instead of N^ε. The result is an algorithm that has asymptotic I/O performance of $O(\nu \log_\mu^2 \nu)$, but is far simpler to implement than Algorithm 5.1 and will generally perform better in practice. The main reason for the increase in performance is that to do polling efficiently the algorithm requires $\varepsilon < 1/8$ (see [31]) and thus in most practical situations $\mu < N^\varepsilon$.

6 Parallel and multi-level extensions

Up to this point our discussion has centered on the case where $D = 1$ and $P = 1$. As has been mentioned, even in this restricted model the results presented here are of tremendous practical importance. Our results are even more significant, however, because the paradigms described in this paper continue to work even when parallelism is added and D and P increase. Furthermore, they can be made to work optimally on hierarchical models having more than two levels; these include the well known HMM [1], BT [2], and UMH [4] (pictured in Figure 3), and their parallelizations [27,37] (pictured in Figure 4).

Details of the algorithms for these models are discussed in the full version of this paper. To a large extent they are based on modified versions of two of the main paradigms discussed above, namely distribution sweeping and batch filtering. We can also rely on the many-way divide-and-conquer approach of Atallah and Tsay [5], which can be extended to the I/O model. To implement distribution sweeping in these models we take advantage of deterministic distribution techniques recently developed by Nodine and Vitter [27] for optimal deterministic sorting. To implement batch filtering, we can use disk striping [28].

Figure 4: Parallel multilevel memory hierarchies. The H hierarchies (of any of the types listed in Figure 3) have their base levels connected by H interconnected processors.

7 Conclusion

We have given a number of paradigms for external-memory computational geometry that yield the first known I/O optimal algorithms for several interesting large-scale problems in computational geometry. Because they are simple and practical both on currently common systems ($P = 1$, $D = 1$), and the parallel I/O systems likely to replace them in the not too distant future, we are convinced that the methods will gain widespread use.

Nevertheless, there are a number of interesting problems that remain open:

- Is there a data structure for 2-d on-line range queries that achieves $O(\log_B \nu + \tau)$ I/Os for updates and range queries using $O(\nu)$ blocks of space? (The off-line version of the problem is solved optimally in this paper.) Updates and three-sided range queries can be handled by metablock trees [20] in $O(\log_B \nu + \log B + \tau)$ I/Os using $O(\nu)$ space. Two-sided range queries anchored on the diagonal can be done in $O(\log_B \nu + \tau)$ I/Os per query and $O(\log_B \nu + (\log_B \nu)^2/B)$ I/Os per (semidynamic) insert [20].
- Can an N-vertex polygon be triangulated using $O(N/B)$ I/Os? Under what conditions?
- Can we find all intersecting pairs of N non-orthogonal segments using $O(\nu \log_\mu \nu + \tau)$ I/Os?

References

[1] A. Aggarwal, B. Alpern, A. K. Chandra & M. Snir, “A Model for Hierarchical Memory,” *Proc. 19th ACM STOC*, New York, NY (1987).

[2] A. Aggarwal, A. Chandra & M. Snir, “Hierarchical Memory with Block Transfer,” *Proc. 28th IEEE FOCS*, Los Angeles, CA (1987).

[3] A. Aggarwal & J. S. Vitter, “The input/output complexity of sorting and related problems,” *Comm. ACM* 31 (1988), 1116–1127.

[4] B. Alpern, L. Carter, E. Feig & T. Selker, “The Uniform Memory Hierarchy Model of Computation,” *Proc. 31st IEEE FOCS*, St. Louis, MO (1990), To appear in *Algorithmica*.

[5] M. J. Atallah & J. -J. Tsay,, "On the Parallel-Decomposability of Geometric Problems,," *Algorithmica* 8 (1992), 209–231.

[6] R. Bayer & E. McCreight, "Organization of large ordered indexes," *Acta Informatica* 1 (1972), 173–189.

[7] B. Chazelle & L. J. Guibas, "Fractional Cascading: II. Applications," *Algorithmica* 1 (1986), 163–191.

[8] B. Chazelle & L. J. Guibas, "Fractional Cascading: I. A Data Structuring Technique," *Algorithmica* 1 (1986), 133–162.

[9] D. Comer, "The ubiquitous B-tree," *Comput. Surveys* 11 (1979), 121–137.

[10] R. F. Cromp, "An Intellegent Information Fusion System for Handling the Archiving and Querying of Terabyte-sized Spatial Databases ," in *Report on the Workshop on Data and Image Compression Needs and Uses in the Scientific Community*, S. R. Tate, ed., CESDIS Technical Report Series #TR-93-99, CESDIS, 1993, 75–84.

[11] J. R. Driscoll, N. Sarnak, D. D. Sleator & R. E. Tarjan, "Making data structures persistent," *J. Comput. System Sci.* 38 (1989), 86–124.

[12] G. Gibson, L. Hellerstein, R. M. Karp, R. H. Katz & D. A. Patterson, "Coding Techniques for Handling Failures in Large Disk Arrays," U. C. Berkeley, UCB/CSD 88/477, December 1988.

[13] D. Gifford & A. Spector, "The TWA Reservation System," *Comm. ACM* 27 (July 1984), 650–665.

[14] M. H. Goodrich, "Geometric partioning made easier, even in parallel," *Proc. 9th ACM Comp. Geo.*, San Diego, CA (1993).

[15] M. T. Goodrich, "Constructing the convex hull of a partially sorted set of points," *Computational Geometry: Theory and Applications* 2 (1993), 267–278.

[16] R. L. Graham, "An efficient algorithm for determining the convex hull of a finite planar set," *Inform. Process. Lett.* 1 (1972), 132–133.

[17] L. J. Guibas & J. Stolfi, "Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams," *ACM Trans. Graphics* 4 (1985), 74–123 .

[18] W. Jilke, "Disk Array Mass Storage Systems: The New Opportunity," Amperif Corporation, September 1986.

[19] P. C. Kanellakis, G. M. Kuper & P. Z. Revesz, "Constraint query languages," *Proc. 9th ACM PODS*, (1990).

[20] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff & J. S. Vitter, "Indexing for Data Models with Constraints and Classes," *Proc. 12th ACM PODS*, Washington, DC (1993).

[21] R. M. Karp, "Probabilistic recurrence relations," *Proc. 23rd ACM STOC*, New Orleans, LA (1991).

[22] D. G. Kirkpatrick & R. Seidel, "The ultimate planar convex hull algorithm?," *SIAM J. Comput.* 15 (1986), 287–299.

[23] D. E. Knuth, *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*, Addison Wesley, Reading, MA, 1973.

[24] R. Laurini & D. Thompson, *Fundamentals of Spatial Information Systems*, A.P.I.C. Series, Academic Press, New York, NY, 1992.

[25] N. B. Maginnis, "Store More, Spend Less: Mid-Range Options Around," *Computerworld* (November 16, 1987).

[26] D. Maier & S. C. Salveter, "Hysterical B-trees," *Inform. Process. Lett.* 12 (1981), 199–202.

[27] M. H. Nodine & J. S. Vitter, "Deterministic distribution sort in shared and distributed memory Multiprocessors," *Proc. 5th ACM SPAA* (1993).

[28] D. A. Patterson, G. Gibson & R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *Proc. 1988 ACM SIGMOD*, (1988).

[29] F. P. Preparata & S. J. Hong, "Convex hulls of finite sets of points in two and three dimensions," *Comm. ACM* 20 (1977), 87–93.

[30] F. P. Preparata & M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, New York–Heidelberg–Berlin, 1985.

[31] J. R. Reif & S. Sen, "Optimal parallel randomized algorithms for three-dimensional convex hulls and related problems," *SIAM J. Comput.* 21 (1992), 466–485.

[32] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison Wesley, Reading, MA, 1989.

[33] H. Samet, *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*, Addison Wesley, Reading, MA, 1989.

[34] R. Tamassia & J. S. Vitter, mentioned in invited paper by J. S. Vitter, "Efficient memory access in large-scale computation," *STACS '93*, Hamburg (February 1991).

[35] R. Tamassia & J. S. Vitter, "Optimal Cooperative Search in Fractional Cascaded Data Structures," *Proc. 2nd ACM SPAA* (1990).

[36] University of California at Berkeley, "Massive Information Storage, Management, and Use (NSF Institutional Infrastructure Proposal)," Technical Report No. UCB/CSD 89/493, January 1989.

[37] J. S. Vitter & E. A. M. Shriver, "Optimal disk I/O with parallel block transfer," *Proc. 22nd ACM STOC*, Baltimore, MD (1990), To appear in *Algorithmica.*

[38] S. B. Zdonik & D. Maier, eds., *Readings in Object-Oriented Database Systems*, Morgan Kauffman, 1990.