Theory and Practice of I/O-Efficient Algorithms for Multidimensional Batched Searching Problems

(Extended Abstract)

Lars Arge* Octavian Procopiuc† Sridhar Ramaswamy‡
Torsten Suel § Jeffrey Scott Vitter¶

Abstract

We describe a powerful framework for designing efficient batch algorithms for certain large-scale dynamic problems that must be solved using external memory. The class of problems we consider, which we call *colorable external-decomposable problems*, include rectangle intersection, orthogonal line segment intersection, range searching, and point location. We are particularly interested in these problems in two and higher dimensions. They have numerous applications in geographic information systems (GIS), spatial databases, and VLSI and CAD design. We present simplified algorithms for problems previously solved by more complicated approaches (such as rectangle intersection), and we present efficient algorithms for problems not previously solved in an efficient way (such as point location and higher-dimensional versions of range searching and rectangle intersection).

We give experimental results concerning the running time for our approach applied to the red-blue rectangle intersection problem, which is a key component of the extremely important database operation spatial join. Our algorithm scales well with the problem size, and for large problems sizes it greatly outperforms the well-known sweepline approach.

1 Introduction

In the past few years much attention has been focused on the development of I/O-efficient algorithms. I/O communication is the bottleneck in many large-scale applications such as those arising in VLSI and CAD design, spatial databases, and geographic information systems (GIS). In this paper we consider I/O-efficient algorithms for batched searching problems. We consider both batched static and batched dynamic problems, and use the correspondence which often exists between a d-dimensional static problem and a $(d-1)$-dimensional dynamic problem to obtain a number of new d-dimensional algorithms.

One prominent example of the problems we consider is the *rectangle intersection* problem, which is a key component in VLSI design rule checking [31] and in the extremely important database operation *spatial join* [34]. We illustrate the practical significance of our algorithms by comparing the empirical performance of our algorithm for this problem with the well-known sweepline algorithm developed for internal memory.

1.1 Problem definition and memory model

A searching problem involves a question asked about a query object x with respect to a set V of objects. In a *batched static searching problem*, a number of queries is asked on a static object set V, and we are concerned only with the overall efficiency over the course of all the queries. In a *batched dynamic searching problem* we are given a sequence of insertions, deletions, and queries, and we must report all answers to the queries as the sequence of actions is performed. Clearly, a batched problem can be solved using a data structure on which the queries are answered one by one, but often much better performance can be obtained. Batched problems play an important role in large-scale performance sensitive applications, because the problem load is often too big to allow complicated online computation, and computation must then be delayed until the load is lighter. One example is a banking application where demand deposits (checks) are processed while the banks are closed at night. Another example is a database application where the index structure is recomputed (or "rebalanced") when the query load is low.

* Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708–0129. Supported in part by U.S. Army Research Office grant DAAH04–96–1–0013. Email: large@cs.duke.edu.

† Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708–0129. Supported in part by the U.S. Army Research Office under grant DAAH04–96–1–0013 and by the National Science Foundation under grant CCR–9522047. Email: tavi@cs.duke.edu.

‡ Bell Laboratories, Murray Hill, NJ 07974–0636. Email: sridhar@research.bell-labs.com.

§ Bell Laboratories, Murray Hill, NJ 07974–0636. Email: suel@research.bell-labs.com.

¶ Center for Geometric Computing, Department of Computer Science, Duke University, Durham, NC 27708–0129. Supported in part by the U.S. Army Research Office under grants DAAH04–93–G–0076 and DAAH04–96–1–0013 and by the National Science Foundation under grant CCR–9522047. Part of this work was done while visiting Bell Laboratories, Murray Hill, NJ. Email: jsv@cs.duke.edu.

In this paper we consider batched problems in the standard two-level I/O model [2], and we define the following parameters:

$$\begin{aligned}
N &= \text{\# of objects in the problem,}\\
K &= \text{\# of queries in the problem,}\\
T &= \text{\# of objects in the solution,}\\
M &= \text{\# of objects/queries fitting in main memory,}\\
B &= \text{\# of objects/queries per disk block,}
\end{aligned}$$

where $M < N$ and $1 \leq B \leq M/2$. For batched dynamic problems N is the number of updates in the problem. Computations can only be done on elements in internal memory. An *input/output operation* (or simply *I/O*) involves reading (or writing) a block from disk into (from) internal memory. Our measures of performance of an algorithm are the number of I/Os it performs and the amount of space (in units of disk blocks) it uses. We will for brevity not address the internal computation time of our algorithms, although they are efficient (and often optimal) in the RAM model.

Since each I/O can transmit B objects or queries simultaneously, it is convenient to introduce the following notation:

$$n = \frac{N}{B}, \qquad k = \frac{K}{B}, \qquad t = \frac{T}{B}, \qquad m = \frac{M}{B}.$$

As $n = N/B$ is the number of I/Os needed just to read N objects, we refer to $O(n)$ as the *linear* I/O bound. We assume that all I/O-bounds are at least linear.

The problems we will be interested in are all what is called *decomposable* [8, 17, 29]. Here we define an external memory version of this property called *external-decomposable.*

Definition 1 *Let $\mathcal{P}$ be a searching problem and let $\mathcal{P}(x,V)$ denote the answer to $\mathcal{P}$ with respect to a set of objects V and a query object x. $\mathcal{P}$ is called* external-decomposable, *if for any partition $A \cup B$ of the set V and for every query x, $\mathcal{P}(x,V)$ can be computed in $O(1)$ additional I/Os given $\mathcal{P}(x,A)$ and P(x,B) in appropriate form.*

A simple and important example of an external-decomposable problem is *one-dimensional range searching*: Given a set V of integers, build a data structure such that given a query range $[x_1, x_2]$, all points in V that lie within the range can be reported efficiently. The problem is external-decomposable as we easily can compute the result for $V = A \cup B$, given the result of a query on A and B. The problem can of course be generalized to higher dimensions where it is also external-decomposable. Another important external-decomposable problem which has received a lot of attention in the computational geometry literature is the d-dimensional *rectangle intersection problem*, that is, the problem of determining all intersecting pairs among a set of axis-parallel hyperrectangles in d-dimensional space [15, 17, 16, 35, 9]. The problem is a key component in VLSI design rule checking [31], and in databases it is a component in the fundamental *join* operator in relational [19], temporal [36], spatial [33, 34], and constraint [22] models.

1.2 Previous related results

As mentioned, considerable attention has recently been given to the development of provably I/O-efficient algorithms. Aggarwal and Vitter [2] considered sorting and permutation related problems in the two-level I/O model and proved that external sorting requires $\Theta(n \log_m n)$ I/Os.[1] I/O-efficient algorithms were later developed for several other problem domains, including computational geometry [1, 3, 6, 18], string problems [5] and graph theory [3, 12, 37, 25]. See the mentioned papers for more complete references; a recent survey is also included in [4]. In the database literature a lot of attention has also been given to I/O-efficient computation, but with more emphasis on practical performance on "real-life" data. Special attention has been given to the development of I/O-efficient spatial join algorithms [10, 20, 21, 24, 26, 27, 30].

A number of I/O-efficient algorithms have been developed for decomposable problems and most of them can be formulated as batched static or dynamic searching problems. Goodrich et al. [18] presented a technique called *distribution sweeping* and used it to develop I/O-efficient algorithms for a number of two-dimensional problems, including the batched range searching problem, the orthogonal line segment intersection problem, and the rectangle intersection problem. The first problem is a batched problem by definition, and the latter two can easily be transformed to one-dimensional batched dynamic problems using the plane sweep paradigm [31]. Arge [3] considered the three problems, and developed I/O-efficient algorithms by looking at them as batched dynamic one-dimensional problems and developing I/O-efficient data structures for such problems. His so-called *buffer trees* are only efficient in a batched setting and cannot be used to answer single queries efficiently. Recently, Arge et al. [6] considered a large number of problems involving line segments in the plane.

In the internal memory setting, batched dynamic problems were considered by Edelsbrunner and Overmars [17]. They were motivated by the fact that for a number of problems dynamic data structures were not

[1] All optimality claims in this paper are in the *comparison I/O-model*, where comparisons are the only allowed operations in internal memory.

known. Even for problems where dynamic structures were known they showed that batched techniques can sometimes be more efficient overall. In external memory the latter motivation plays an even bigger role because of fundamental computational limitations. A simple illustration of this is the one-dimensional range searching problem. The obvious data structure for this problem is the B-tree [7, 13]. A B-tree on N elements uses optimal $O(n)$ space, can be built in $O(n \log_B n)$ I/Os, and can be used to answer a range query in $O(\log_B n + t)$ I/Os. It is easy to realize that the query bound is optimal. The batched static version of the problem can thus obviously be solved by building a B-tree on V and then performing the K queries one by one. This results in an $O((n + K) \log_B n + t)$ solution. However, unlike in internal memory where a similar approach using a balanced binary search tree results in an optimal $O((K + N) \log_2 N + T)$-time solution, we can get a better (and optimal) $O((n + k) \log_m n + t)$-I/O solution by using distribution sweeping or buffer trees.

Recently, some research has also been done on the practical merits of the algorithms. Chiang [11] implemented the orthogonal line segment intersection algorithm developed in [18] using distribution sweeping and showed that it outperforms internal memory solutions even on moderately sized instances. Vengroff [38, 39] designed TPIE (*Transparent Parallel I/O programming Environment*), a set of C++ functions and templated classes that allow for a simple and efficient implementation of two-level external-memory algorithms. Efficient TPIE implementations for a variety of sorting and scientific computing applications are given in [40].

1.3 The results in this paper

The main result in this paper is a technique for designing I/O-efficient and space-efficient batched dynamic algorithms for external-decomposable problems. Our technique works for a wide range of problems that we call "colorable." We define the colorable property in Section 2 and show that a number of natural one-dimensional problems such as range searching are colorable.

In Section 3 we describe our technique and use it to obtain algorithms for some of the two-dimensional problems also considered in [3, 18]. Our algorithms have the same $O(n \log_m n + t)$ optimal performance as the previously developed algorithms, but in some sense our technique provides a general framework for the solution of the problems. We also show how our technique can be used to obtain new I/O-efficient algorithms by providing the first dynamic version of external planar point location. In Section 3 we also show that our technique can be used recursively, and thus we obtain the first known I/O-efficient algorithms for d-dimensional batched range searching, orthogonal line segment intersection, and rectangle intersection. Our algorithms use $O(n \log_m^{d-1} n + t)$ I/Os and linear space. We believe that our technique will prove useful in the design of other I/O-efficient algorithms. In Section 3 we give one further application of our result to a batched dynamic problem for which no solution was previously known.

In Section 4 we demonstrate the practical merits of our approach, by comparing the empirical performance of an $O(n \log_m n+t)$-I/O algorithm developed using our technique with an optimal $O(N \log_2 N + T)$-time sweep algorithm developed for internal memory. The problem we consider is a special case of the rectangle intersection problem, and one of the two subproblems of spatial join. Algorithms with a similar I/O bound for this problem can be developed using known techniques. However, our algorithm is very simple and practical and is readily implemented in TPIE. Our experiments show that the sweep algorithm "breaks down" once the size of the sweepline structure becomes bigger than the available internal memory, whereas our external algorithm scales well.

2 Batched static colorable problems

In this section we define the notion of *colorability* and show that a number of simple one-dimensional batched static problems are colorable.

Definition 2 *Let $\mathcal{P}$ be an external-decomposable batched searching problem. Consider the problem $\mathcal{P}_C$ where a color chosen from a set C is associated with each query x, and where a set of colors C_v is associated with each object $v \in V$. Only objects where $color(x) \in C_v$ are considered when answering x. Problem $\mathcal{P}$ is called $(I(N,K), S(N,K))$ $m^{1/c}$-*colorable *if the following two conditions hold:*

1. *For all colorings where $|C| = \Theta(\sqrt{m^{1/c}})$ and where the number of different color sets C_v is $O(m^{1/c})$, for some constant $c \geq 1$, P_C can be solved in $O(I(N,K)+t)$ I/O operations and $O(S(N,K))$ space after an initial sorting step, and*
2. *If (V_1, Q_1) and (V_2, Q_2) are two valid instances of $\mathcal{P}$ then $(V_1 \cup V_2, Q_1 \cup Q_2)$ is also a valid instance.*

We call an m-colorable problem just a *colorable* problem. Note that such a problem is $m^{1/c}$-colorable for any c. We can show that the one-dimensional batched range searching problem as well as a number of other simple one-dimensional problems are $(n + k, n + k)$ colorable. Here we consider a more general problem, namely, the batched *interval intersection searching* problem, where a query and the objects in V are integer intervals. A query with interval e must return all intervals in V having a point in common with e. The

1. Sort the intervals according to their left endpoints.
2. Scan the sorted list of intervals, maintaining $O(m)$ (initially empty) active lists, one *color list* C for each of the $\Theta(\sqrt{m})$ colors and one *set list* S for each of the $O(m)$ different color sets. For every interval r do the following:
   (a) If r is a query then add r to color list $C_{color(r)}$, and scan through all set lists S_U corresponding to color sets U containing $color(r)$ one at a time, reporting intersections between r and intervals in S_U and removing intervals from S_U that do not intersect r.
   (b) If $r \in V$ then add r to the set list corresponding to r's color set and scan through every color list C_c corresponding to colors in the color set, reporting intersections between intervals in C_c and r and removing intervals from C_c that do not intersect r.

Figure 1: Algorithm for the batched colored interval intersection searching problem.

algorithm showing that the problem is $(n+k, n+k)$ colorable is given in Figure 1. After an initial sorting step, the sorted list of intervals is scanned and intersections are reported using a number of *active lists*.

Lemma 1 *The batched static one-dimensional interval intersection searching and range searching problems are $(n+k, n+k)$ colorable.*

Proof: In order to establish the correctness of the algorithm we observe that an interval $v \in V$ and a query q that intersect can be classified into two cases: (i) v begins before q and (ii) q begins before v. Step 2a of the algorithm reports all intersections between a query interval and currently "active" intervals from V with a relevant colorset, thus handling case (i), while Step 2b similarly handles case (ii). Note that when an interval is removed from an active list we are sure that it will not intersect relevant intervals processed later in the scan.

The complete scan after the initial sorting step can be performed in $O(n+t)$ I/Os, as can be seen by the following reasoning: The number of active lists is $O(m)$, so there is room for one block from each of the lists in internal memory. We collect intervals inserted into an active list in internal memory and only write them to disk once B of them have been collected. Thus $N+K$ insertions can be processed in $O(n+k)$ I/Os. An interval is added only once to an active list, and in each subsequent scan of the list the interval is either removed permanently or contributes an intersection to the interval that initiated the scan. A simple amortization argument completes the proof. □

3 Batched dynamic problems

In this section we develop a general technique for solving a batched dynamic version of a colorable problem in an I/O-efficient manner. Our approach is inspired by an approach used by Edelsbrunner and Overmars [17]. An instance of a batched dynamic problem $\mathcal{P}$ consists of a sequence of N actions $a_1, a_2, \ldots, a_N$, where each action is either an insertion of a set object, a deletion of a set object, or a query with a query object. For an action a_i we can regard i as the *time* at which the action is performed. For each object v that ever belongs to the set V, there is a time i_1 (possibly $-\infty$) when it is inserted and a time i_2 (possibly $+\infty$) when it is deleted; we refer to $[i_1, i_2]$ as v's *existence interval*. When a query q is performed at time j, it should be performed relative to the set of objects present at that time, that is, relative to all objects whose existence interval contains j. The basic idea in [17] is to use a segment tree [9, 31] to find these objects.

Here we use the same basic idea, but the use of an external segment tree [3] complicates things considerably. The base structure of an external segment tree is a perfectly balanced tree with branching factor $\sqrt{m}$ over the $N+K$ action times. Each leaf represents M consecutive action times and thus the tree has height $O(\log_{\sqrt{m}}((N+K)/M)) = O(\log_m(n+k))$. (See Figure 2.) The first level of the tree partitions the action times into $\sqrt{m}$ intervals σ_i—for illustrative reasons we call them *slabs*—separated by dotted lines in Figure 2. Existence intervals such as $\overline{CD}$ in Figure 2 that completely span at least one slab are called *long intervals*; a copy of the object corresponding to each long existence interval is stored in the root. Existence intervals that are not long are called *short intervals*; the objects corresponding to such intervals are not stored in the root, but are passed recursively down to lower levels of the tree where their existence intervals span slabs. $\overline{AB}$ and $\overline{EF}$ are examples of such existence intervals. Furthermore, we imagine that we "cut" each long existence intervals at the leftmost (rightmost) boundary of the leftmost (rightmost) slab it completely spans, and treat the portions that do not span a slab as small intervals. For example $\overline{CE}$ is cut at the boundary between slabs σ_0 and σ_1 and between slabs σ_3 and σ_4, and the portions in slabs σ_0 and σ_4 are stored further down the tree. Note that at most $M/2$ objects are stored in a leaf. Each object can be stored in several nodes of the structure, but at most twice on each level; thus, the total space utilization is $O(n \log_m(n+k))$.

To answer a query q at time j, we search down the tree to the leaf containing j and in each visited node we answer the query relative to all the "relevant" objects

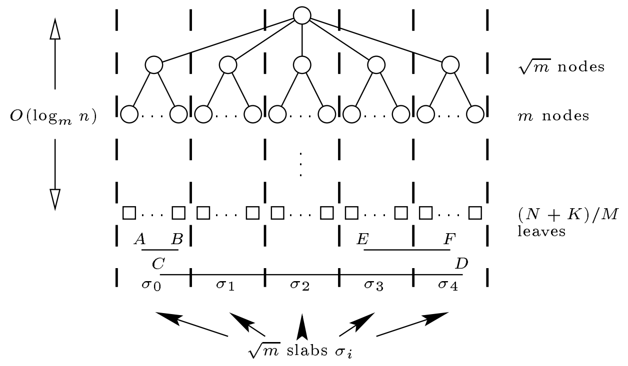

Figure 2: External-memory segment tree

stored in the node. The fact that $\mathcal{P}$ is decomposable is used when combining the results of the individual node queries. In a node where q lies in slab σ_i, we define the *relevant* objects to be objects corresponding to existence intervals that completely span σ_i. Each object present at time i will be relevant exactly once on the search path. In the internal memory solution [17] where the segment tree is binary, all objects in a node are relevant for all queries passing the node. This is not the case in the external memory setting, which is one reason why the problem is more challenging to solve in external than in internal memory. Another is that a single query cannot be answered I/O-efficiently, and therefore we perform all the queries simultaneously (normally referred to as *batched filtering* [18]) and take advantage of the fact that when we do so the problem we need to solve in each node is a batched static "colored" version of $\mathcal{P}$. To realize this fact, consider a node r in the segment tree, the set of objects V_r assigned to it, and the set of queries Q_r passing through it. Imagine that we associate a distinct color with each of the $\sqrt{m}$ slabs and color each of the queries in Q_r with the color of the slab containing it. Now consider an object e in V_r. The existence interval for e completely spans a set of slabs and e is relevant for all queries in these slabs. We associate with e the colors of the slabs that its existence interval spans. The key property is that the maximum number of distinct contiguous ranges of slabs (what is normally called *multislabs* [3, 6])—and thus the maximum number of color sets associated with the objects in V_r—is a quadratic function of the branching factor, and thus is $O(m)$. Therefore, the problem we need to solve in r is a colored batched static version of $\mathcal{P}$ on the objects V_r and queries Q_r. A sketch of the complete algorithm is given in Figure 3.

Theorem 1 *The batched dynamic version of an $(I(N,K), S(N,K))$ colorable problem $\mathcal{P}$ can be solved in $O(I(N,K) \cdot \log_m(n+k)+t)$ I/O operations using $O(S(N,K))$ space.*

Proof: The number of I/Os used to construct the external segment tree and distribute the objects and queries to the nodes of it (Steps 1 and 2 in Figure 3) is $O(n \log_m(n+k))$: The number of levels of the structure is $O(\log_m n)$ and to construct one level we scan the nodes on the previous level and the sorted list of $N+K$ objects and queries, both of which can be done in $O(n+k)$ I/Os.

Consider the nodes $r_1, r_2, \ldots, r_l$ on one level of the structure and let N_{r_i} and K_{r_i} denote the number of objects and the number of queries assigned to r_i, respectively. The number of I/Os used to solve the colored batched static problems on the level (not counting the initial sorting step) is then $\sum_i I(N_{r_i}, K_{r_i}) + t$, which is $O(I(N,K)+t)$ as I is at least linear and as $\sum_i N_{r_i} \le N$ and $\sum_i K_{r_i} \le K$. Furthermore, by presorting the queries and objects (using $O((n+k)\log_m(n+k))$ I/Os) and distributing them in sorted order to the nodes, we can avoid sorting when solving a batched static problem, and thus the total number of I/Os used in Step 3 is $O(I(N,K) \cdot \log_m(n+k)+t)$. A space bound of $O(S(N,K) \cdot \log_m(n+k))$ follows from a similar argument, but the space can be improved to $O(S(N,K))$ by noting that if we solve the batched static problem for a level of nodes before going on to build the next level, there is no need to store more than one level of the segment tree at any time (in [17] this idea is called *streaming*). $\square$

3.1 Simple applications to one and two-dimensional problems

Theorem 1 together with Lemma 1 immediately give us efficient solutions for one-dimensional batched dynamic range searching and interval intersection. It is well known that a number of two-dimensional batched static problems can be regarded as one-dimensional batched dynamic problems using the plane sweep technique. For example, the orthogonal line segment intersection problem (namely, given a set of N line segments in the plane parallel to the axes, report all intersecting orthogonal pairs) can be reduced to solving a batched dynamic one-dimensional range searching problem. Similarly, the two-dimensional batched range searching problem can be regarded as a simple version of batched dynamic interval intersection, where the queries are points. We can thus also obtain efficient algorithms for these two problems, and combining them we obtain a solution to the rectangle intersection problem.

All the above external-decomposable problems are of a type where the solution to $\mathcal{P}(x, A \cup B)$ is just the concatenation of $\mathcal{P}(x, A)$ and $\mathcal{P}(x, B)$. One problem where this in not the case is the batched static version of the external-decomposable *immediate obstacle problem* [17, 28], where we are given a set of N points and vertical line segments in the plane, and for each point p we should compute the first segment hit by a horizontal

1. Sort the list A of the $N+K$ actions by time.
2. Construct the external segment tree and distribute the objects and queries to the relevant node:
   (a) Create the $(n+k)/m$ leaves by scanning through A.
   (b) Repeatedly, scan through the last level of nodes created and the list A, creating one more level of the segment tree and distributing the objects and queries in A to the relevant (newly created) nodes.
3. For each node r in turn solve a colored batched static version of $\mathcal{P}$ on the colored objects and queries associated with r.

Figure 3: Algorithm (sketch) for the batched dynamic problem $\mathcal{P}$.

ray originating in p and going right. In the immediate obstacle problem the solution to $\mathcal{P}(x, A \cup B)$ is obtained by comparing the two segments obtained as solutions to $\mathcal{P}(x, A)$ and $\mathcal{P}(x, B)$ to see which one is closest to x. It is easily realized that the problem can be solved using a batched dynamic version of a simple colorable one-dimensional search problem, and Theorem 1 thus again applies.

Corollary 1 *The one-dimensional batched dynamic range searching and interval intersection problems can both be solved in $O((n+k)\log_m(n+k)+t)$ I/Os and $O(n+k)$ space. The orthogonal line segment intersection, 2d batched range searching, and 2d rectangle intersection problems can all be solved in $O(n\log_m n+t)$ I/Os and $O(n)$ space. The batched static version of the immediate obstacle problem can be solved in $O((n+k)\log_m(n+k))$ I/Os and $O(n+k)$ space.*

It should be noted that similar I/O and space bounds have been, or can easily be, obtained using distribution sweeping [18]. In the next subsection we will extend our technique and use it to obtain solution to problems for which no I/O-efficient algorithms were previously known.

3.2 Advanced and higher dimensional applications

By decreasing the fan-out of the segment tree used in the previous section from $\sqrt{m}$ to $\sqrt[4]{m^{1/c}}$ we can prove the following (proof omitted for brevity).

Theorem 2 *The batched dynamic version $\mathcal{P}_{bd}$ of an $(I(N,K), S(N,K))$ $m^{1/c}$-colorable problem $\mathcal{P}$ can be solved in $O(I(N,K)\log_m(n+k)+t)$ I/O operations using $O(S(N,K))$ space. $\mathcal{P}_{db}$ is $(I(N,K), S(N,K))$ $\sqrt{m^{1/c}}$-colorable.*

Theorem 2 can now immediately be applied to our algorithm for the batched static immediate obstacle problem in Section 3.1 to make the algorithm work on the batched dynamic problem, and by using the theorem recursively the algorithms discussed in Section 3.1 can be extended to work in d dimensions.

Corollary 2 *For each constant $d > 1$, the d-dimensional batched range searching problem and the d-dimensional rectangle intersection problem can both be solved in $O(n\log_m^{d-1} n+t)$ I/Os and $O(n)$ space. The batched dynamic immediate obstacle problem can be solved in $O(n\log_m^2 n)$ I/Os and $O(n)$ space.*

A more complicated application of Theorem 2 is to the *batched dynamic planar point location* problem. In our formulation of this problem we are given a set of N non-intersecting (and not necessarily orthogonal) line segments in the plane and a set of K points. The goal is to find for each point the first segment hit by an upwards ray originating in the point. In [6] a rather complicated $O((n+k)\log_m n)$-I/O solution to the *static* problem is given. The solution shows that the problem is $((n+k)\log_m(n+k), n+k)$ $\sqrt{m}$-colorable, except for the condition that given two valid instances of the problem their union should also be a valid instance. (Details will be given in the full paper.) The union condition does not hold, since the segments in the union can be intersecting. However, if we restrict our attention to instances where only insertions or deletions are allowed (the so called *semidynamic* problems), the segments must be non-intersecting and the condition is fulfilled.

Corollary 3 *The batched semidynamic planar point location problem can be solved in $O(n+k)$ space and $O((n+k)\log_m^2(n+k))$ I/Os.*

4 Experimental results

In this section we illustrate the practical significance of our algorithms by comparing the empirical performance of an I/O-optimal algorithm developed using our technique with an optimal internal memory sweep algorithm. The problem we consider is a variant of the two-dimensional rectangle intersection problem that we call *red-blue rectangle intersection*: Given a set of axis-parallel red rectangles and a set of axis-parallel blue rectangles in the plane, report all red-blue intersecting rectangles. The problem has been extensively studied in the database literature as one of the two subproblems of spatial join, which is a core operation in spatial database systems such as geographic information systems [10, 20, 27, 26, 24, 30].

A simple plane sweep internal memory algorithm can be easily derived from the algorithms for the rectangle intersection problem [9, 17, 15, 16, 21, 35]. This well-known $O(N \log_2 N + K)$-time algorithm sweeps the plane with a vertical line, while maintaining and querying two interval trees [15]. After the initial sorting step the algorithm can be viewed as a red-blue version of the batched dynamic interval intersection problem. This problem is the same as the interval intersection problem we have considered, except that the intervals are colored red and blue, and only red-blue intersections should be reported. It is easy to modify the algorithm in Figure 1 to work for the red-blue problem, and Theorem 1 immediately gives an optimal $O(n \log_m n + t)$-I/O algorithm for the red-blue rectangle intersection problem. It should be noted that this is not a new theoretical result, as previous solutions for the batched range searching problem and the orthogonal line segment intersection problem [3, 18] can be combined to obtain an optimal algorithms for the problem. However, our technique suggests a new *practical* algorithm that solves the problem in one go. If we imagine building the segment tree structure level by level from the top, and solving the batched static problem on each level, the algorithm can be viewed as a distribution sweeping [18] algorithm, that divides the plane into m slabs, performs a vertical sweep over the slabs to locate intersections, and then recursively solves the problem in each slab.

As mentioned, the red-blue rectangle intersection problem has been extensively studied in the database literature. The proposed algorithms can be roughly classified into two groups: those that use an indexing structure (typically an R-tree variant) built on the two rectangle sets [10, 20] and those that do not [27, 30, 24]. There has been a trend towards analyzing algorithms that do not rely on an index. For example, the PBSM (Partition Based Spatial-Merge) algorithm by Patel and DeWitt [30] has been shown to outperform those based on an R-tree index when the cost of building the index is counted. PBSM can be viewed as a special case of our approach, in which the objects that cross slabs are duplicated in each slab rather than handled with a sweep and recursion. Other algorithms try to avoid too much duplication by using sophisticated partition methods but they are still vulnerable to skewed data [27, 24]. The performance of our algorithm can therefore be said to be similar to these algorithms, except that our algorithm is not prone to skewed locations or shapes of rectangles.

4.1 Implementation considerations

Before we present our empirical results in Sections 4.2 and 4.3, a few notes should be made about our implementations of the two algorithms. We based both of our implementations on the TPIE system [38, 39]. As mentioned, TPIE is a collection of templated functions and classes, and the basic data structure in TPIE is a *stream*, representing a list of objects of the same type. The system contains I/O-efficient implementations of algorithms for scanning, merging, distributing, and sorting streams. Looking at the two algorithms, we quickly see that all the building blocks we need—scanning, sorting, distributing—are already implemented in TPIE. This made the implementation of the algorithm relatively easy and facilitated modular design.[2]

In order to improve practical performance and provide a fair comparison, we made a number of modifications in our implementations relative to the theoretical descriptions of the two algorithms. In the sweepline algorithm, which we call `internal_join`, we improved the performance of the sorting step that is done before the sweep by using TPIE's built-in I/O-efficient sorting algorithm. In the sweep itself, we did not delete an interval from the interval tree when the sweepline left the corresponding rectangle; instead we performed a "lazy deletion" operation on the tree while processing queries to remove "expired" intervals. We implemented a simplified version of the interval tree similar to that described in [14] but used a randomized skip list [32] as the underlying structure instead of a balanced tree structure. Finally, we chose to use a horizontal sweep line instead of a vertical one in order to have the two algorithms sweep in the same direction.

The external algorithm, called `external_join`, was modified to used random sampling to divide the x-interval into slabs instead of presorting the data an extra time. Note that it is possible to solve the problem quickly in internal memory even when the problem size N is much larger than internal memory size M, because the data structure size is related to the maximum number of rectangles that intersect a sweepline, which may be less than M. For example, the rectangles may be small and uniformly distributed, in which case relatively few rectangles would intersect any given sweepline. Therefore we used an optimistic implementation in `external_join` that began each subproblem by running `internal_join`, hoping that the interval trees would fit in internal memory during the sweep. If TPIE detected that the available memory was exhausted, the sweep was aborted and we proceeded with the external approach.

4.2 Experimental data

Along the lines of Chiang [11], we generated four types of input data sets with $N/2$ red and $N/2$ blue rectan-

[2] The TPIE system can be downloaded from http://www.cs.duke.edu/TPIE/. The algorithms described in this paper will be included in the next distribution of TPIE.

gles each, placed in a $[0, N] \times [0, N]$ square. In order to guarantee that the reporting cost would not dominate the $O(n \log_m n)$ searching cost, the rectangles were chosen so that the number of intersections between red and blue rectangles was $O(N)$. As discussed, an important parameter for the efficiency of the two algorithms is the average number of rectangles cut by a horizontal sweep line during a sweep (the *average overlap*). Intuitively, this parameter decides not only the size of the interval trees in the sweep, but also the size of the active lists in the external algorithm. Thus we generated data with a varying number of overlaps.

In the first data set, which we call `small_rect`, we generated data meant to resemble GIS data (small and relatively uniform distributed rectangles). We chose the width and height of the rectangles randomly in $[0, \sqrt{N}]$, and the x and y coordinates of the lower left corner were chosen randomly in $[0, N - \sqrt{N}]$. It can be shown that the expected number of intersections between red and blue rectangles in such a set is approximately $N/4$, while the expected average overlap and expected maximum overlap are approximately $\sqrt{N}/4$ [23]. Details will appear in the full paper. The second data set, `tall_rect`, represents a "hard" instance as it consists of long and skinny vertical rectangles with a large average overlap: We used a fixed width h (10 in the experiments), chose the height uniformly in $[0, N/2]$, and chose the x and y coordinates of the lower left corner uniformly in $[0, N - h]$ and $[0, N/2]$, respectively. The fixed width ensures that the expected number of intersections is approximately $hN/3$, while the expected average overlap is approximately $N/4$. In order to investigate the influence of the average overlap, we produced the third set, `wide_rect`, simply by rotating the previous data set 90 degrees. The `wide_rect` data set consists of long and horizontally skinny rectangles, with the same number of intersections as before, but with an expected average overlap of approximately $h/2$. The fourth data set, `wide&tall_rect`, consists of both wide and tall rectangles. The wide rectangles were placed in $[0, N/2] \times [0, N]$, and the tall rectangles in $[N/2, N] \times [0, N]$. The expected average number of intersections in this set is approximately $hN/4$, and the expected average overlap approximately $N/8 + h/2$. In the full paper we provide a full analysis of the data sets.

4.3 Empirical results

We performed our experiments on a Sun SparcStation20 running Solaris 2.5, with 32 Mbytes of internal memory. In order to avoid network activity, a local disk was used for the input files as well as for scratch files. While we did not restrict the amount of internal memory `internal_join` could use (and thus relied on the virtual memory system), the amount of internal memory used

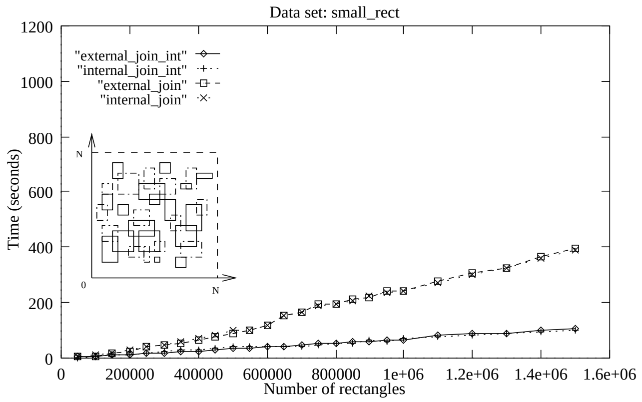

Figure 4: `small_rect`: Average no. of intersections $\approx N/4$. Average no. overlaps $\approx \sqrt{N}/4$.

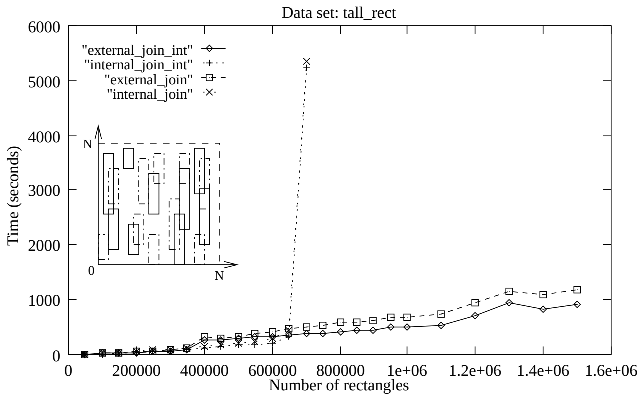

Figure 5: `tall_rect`: Average no. of intersections $\approx hN/3$. Average no. overlaps $\approx N/4$.

by `external_join` was limited to a fixed amount. Experimenting with different values for this parameter, as well as with different values of the logical block size used by TPIE, we found that the best performance was obtained with the main memory use (by TPIE) restricted to 12Mbytes and with a block size of 20 times the physical block size (4Kbytes).

We ran the two programs on the four data sets, with the number of rectangles varying between 50,000 and 1,500,000. Each rectangle consisted of an integer identifier and two corner points represented by two doubles each. Each rectangle thus used 40 bytes and the real size of the data sets varied approximately between 2Mbytes and 60Mbytes. Figures 4 to 7 show the running times of the two programs for each of the data sets. The `external_join` and `internal_join` curves represent the total running times (including sorting), while the `external_join_int` and `internal_join_int` curves represent the times needed to compute the intersections of already sorted inputs.

Our experiments show that our external memory algorithm is very efficient in practice and that the running times of both algorithms depend heavily on the

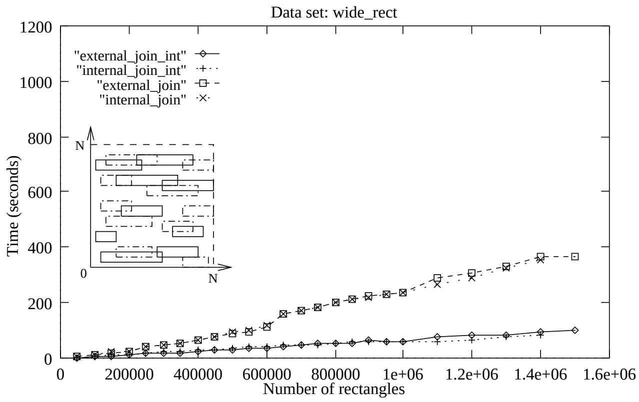

Figure 6: `wide_rect`: Average no. of intersections $\approx hN/3$. Average no. overlaps $\approx h/2$.

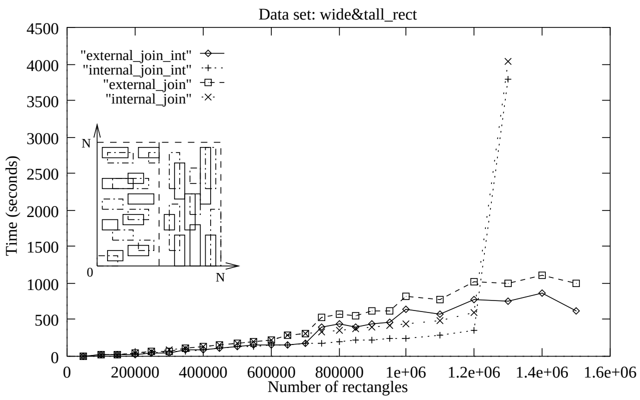

Figure 7: `wide&tall_rect`: Average no. of intersections $\approx hN/4$. Average no. overlaps $\approx N/8 + h/2$.

average overlap. The `external_join` algorithm has a steady and efficient performance on the two data sets with a large number of overlaps, but `internal_join` "breaks down" when the size of the problem becomes bigger than the available main memory. On the data sets with small overlap the performance of the two programs is comparable, and the sorting time dominates the overall running time.

In the following we make a few comments on the results for each data set: The two algorithms perform similarly on the `small_rect` data set, in which the rectangles exhibit locality and have a small number of overlaps. On these inputs, the `external_join` algorithm never breaks the plane up into slabs and is thus basically the same as `internal_join`. The small difference is the two performance curves is due to variations in the experimental conditions (operating systems interference, randomization in the interval tree, etc.). The number of rectangles would need to be more than 1 trillion before internal memory would be exhausted. Note however, that if a sorting algorithm developed for internal memory had been used instead of the TPIE algorithm, the "breakdown" of `small_rect` would have occurred much before that. In `tall_rect` the average overlap is large, and the point where the interval trees do not fit in internal memory is quickly reached, making `internal_join` thrash. As can be seen from the graph, the thrashing point is reached around 700,000 rectangles. When the number of rectangles is between 400,000 and 600,000, `internal_join` performs slightly better than `external_join`. The reason is that the latter algorithm detects that not enough memory is available, aborts the sweep algorithm, and starts the external algorithm; the runtime penalty for the aborted sweep counteracts the benefits of the later distribution sweep. On the third data set, `wide_rect`, the algorithms perform as on `small_rect` because the average number of nodes in the interval trees is small (constant). Finally, on `wide&tall_rect`, which is a mixture of the previous two, the breakdown of `external_join` occurs at around 1,300,000 rectangles, which is to be expected, since the average number of overlaps is half of that of `tall_rect`.

5 Conclusions

We have demonstrated a fairly general technique for developing batched dynamic algorithms that are efficient in an I/O setting for a variety of decomposable problems. Our empirical study of algorithms for the red/blue rectangle intersection problem suggests that our approach is fast in practice and outperforms currently used methods, especially when the problem size gets too large for internal memory.

There are several avenues of research regarding practical implementation. We are currently studying other algorithms for red/blue rectangle intersection, as well as improving our current implementation. One way of doing so could be to try to predict for a given (sub) problem (e.g. using sampling) if the interval trees in the sweepline approach would fit in memory. We are also implementing batched dynamic algorithms for other decomposable problems.

References

[1] P. K. Agarwal, L. Arge, T. M. Murali, K. Varadarajan, and J. S. Vitter. I/O-efficient algorithms for contour line extraction and planar graph blocking. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, 1998.

[2] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995. A complete version appears as BRICS technical report RS-96-28, University of Aarhus.

[4] L. Arge. *Efficient External-Memory Data Structures and Applications.* PhD thesis, University of Aarhus, February/August 1996.

[5] L. Arge, P. Ferragina, R. Grossi, and J. Vitter. On sorting strings in external memory. In *Proc. ACM Symp. on Theory of Computation*, pages 540–548, 1997.

[6] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica (to appear in special issues on Geographical Information Systems)*, 1998.

[7] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1972.

[8] J. L. Bentley. Decomposable searching problems. *Information Processing Letters*, 8:244–251, 1979.

[9] J. L. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, 29:571–577, 1980.

[10] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. SIGMOD Intl. Conf. on Management of Data*, 1993.

[11] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 346–357, 1995.

[12] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.

[13] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[15] H. Edelsbrunner. A new approach to rectangle intersections, part I & II. *Int. J. Computer Mathematics*, 13:209–229, 1983.

[16] H. Edelsbrunner and H. A. Maurer. On the intersection of orthogonal objects. *Information Processing Letters*, 1981.

[17] H. Edelsbrunner and M. Overmars. Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6:515–542, 1985.

[18] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 714–723, 1993.

[19] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 1993.

[20] O. Günther. Efficient computation of spatial joins. In *Proc. IEEE International Conference on Data Engineering*, pages 50–60, 1993.

[21] R. H. Güting and W. Schilling. A practical divide-and-conquer algorithm for the rectangle intersection problem. *Information Science*, 42:95–112, 1987.

[22] P. C. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. In *Proc. ACM Symp. Principles of Database Systems*, pages 299–313, 1990.

[23] C. M. Kenyon and J. S. Vitter. Maximum queue size and hashing with lazy deletion. *Algorithmica*, 6:597–619, 1991.

[24] N. Koudas and K. C. Sevcik. Size separation spatial join. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 324–335, 1997.

[25] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, 1996.

[26] M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 209–220, 1994.

[27] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 247–258, 1996.

[28] E. McCreight. <problem 81-8>. *Journal of Algorithms*, 2:314, 1981.

[29] M. H. Overmars. *The Design of Dynamic Data Structures.* Springer-Verlag, LNCS 156, 1983.

[30] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 259–270, 1996.

[31] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[32] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 35:668–676, 1990.

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

[34] H. Samet. *The Design and Analyses of Spatial Data Structures.* Addison Wesley, MA, 1989.

[35] H. W. Six and D. Wood. Counting and reporting intersections of d-ranges. *IEEE Transactions on Computers*, 31:181–187, 1982.

[36] A. U. Tanzel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal Databases: Theory, Design and Implementation.* The Benjamin/Cummings Publishing Company Inc., 1993.

[37] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intellegence*, 3:331–360, 1991.

[38] D. E. Vengroff. A transparent parallel I/O environment. In *Proc. DAGS Symposium on Parallel Computation*, 1994.

[39] D. E. Vengroff. *TPIE User Manual and Reference.* Duke University, 1995. Available via WWW at `http://www.cs.duke.edu/TPIE`.

[40] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proceedings of the Goddard Conference on Mass Storage Systems and Technologies*, NASA Conference Publication 3340, Volume II, pages 553–570, 1996.