Algorithmica (2007) 47: 1–25
DOI: 10.1007/s00453-006-1208-z

Algorithmica
© 2006 Springer Science+Business Media, Inc.

External-Memory Algorithms for Processing Line Segments in Geographic Information Systems[1]

Lars Arge,[2] Darren Erik Vengroff,[3] and Jeffrey Scott Vitter[4]

Abstract. In the design of algorithms for large-scale applications it is essential to consider the problem of minimizing I/O communication. Geographical information systems (GIS) are good examples of such large-scale applications as they frequently handle huge amounts of spatial data. In this paper we develop efficient external-memory algorithms for a number of important problems involving line segments in the plane, including trapezoid decomposition, batched planar point location, triangulation, red–blue line segment intersection reporting, and general line segment intersection reporting. In GIS systems the first three problems are useful for rendering and modeling, and the latter two are frequently used for overlaying maps and extracting information from them.

Key Words. Algorithms, External-memory, GIS, Line segment intersection.

1. Introduction. The Input/Output communication between fast internal memory and slower external storage is the bottleneck in many large-scale applications. The significance of this bottleneck is increasing as internal computation gets faster, and especially as parallel computing gains popularity [24]. Currently, technological advances are increasing CPU speeds at an annual rate of 40–60% while disk transfer rates are only increasing by 7–10% annually [26]. Internal memory sizes are also increasing, but not nearly fast enough to meet the needs of important large-scale applications, and thus it is essential to consider the problem of minimizing I/O communication.

Geographical information systems (GIS) are a rich source of important problems that require good use of external-memory techniques. GIS systems are used for scientific applications such as environmental impact, wildlife repopulation, epidemiology analysis, and earthquake studies and for commercial applications such as market analysis, facility location, distribution planning, and mineral exploration. In support of these applications, GIS systems store, manipulate, and search through enormous amounts of spatial data

[1] An extended abstract version of this paper was presented at the Third Annual European Symposium on Algorithms (ESA '95). This work by Lars Arge was done while at Duke University, supported in part by the U.S. Army Research Office under Grant DAAH04-96-1-0013. Darren Vengroff's work was done while a Visiting Scholar at Duke University, supported in part by the U.S. Army Research Office under Grant DAAH04-93-G-0076 and by the National Science Foundation under Grant DMR-9217290. This work was done while Jeffrey Vitter was at Duke University, supported in part by the National Science Foundation under Grants CCR-9007851 and CCR-9522047 and by the U.S. Army Research Office under Grants DAAH04-93-G-0076 and DAAH04-96-1-0013.

[2] Department of Computer Science, University of Aarhus, 8200 Aarhus N, Denmark. large@daimi.au.dk.

[3] Pelago, Inc., Seattle, WA 98118, USA. vengroff@pelago.com.

[4] Department of Computer Science, Purdue University, West Lafayette, IN 47907, USA. jsv@purdue.edu.

Received January 15, 1996; revised January 23, 1998. Communicated by J. S. Snoeyink and M.-Y. Kao. Online publication October 7, 2006.

[16], [28]. NASA's EOS project GIS system [16], for example, is expected to manipulate petabytes (thousands of terabytes or millions of gigabytes) of data!

Typical subproblems that need to be solved in GIS systems include point location, triangulating maps, generating contours from triangulated elevation data, and producing map overlays, all of which require manipulation of line segments. As an illustration, the computation of new scenes or maps from existing information—also called map overlaying—is an important GIS operation. Some existing software packages are completely based on this operation [28]. Given two thematic maps (piecewise linear maps with, e.g., indications of lakes, roads, pollution level), the problem is to compute a new map in which the thematic attributes of each location is a function of the thematic attributes of the corresponding locations in the two input maps. For example, the input maps could be a map of land utilization (farmland, forest, residential, lake), and a map of pollution levels. The map overlay operation could then be used to produce a new map of agricultural land where the degree of pollution is above a certain level. One of the main subproblems in map overlaying is "line-breaking," which can be abstracted as the red–blue line segment intersection problem.

In this paper we present efficient external-memory algorithms for large-scale geometric problems involving collections of line segments in the plane, with applications to GIS systems. In particular, we address region decomposition problems such as trapezoid decomposition and triangulation, and line segment intersection problems such as the red–blue segment intersection problem and more general formulations.

1.1. *The I/O Model of Computation.* The primary feature of disks that we model is their extremely long access time relative to that of solid state random-access memory. In order to amortize this access time over a large amount of data, typical disks read or write large blocks of contiguous data at once. Our problems are modeled by the following parameters:

$$\begin{aligned} N &= \text{\# of items in the problem instance;} \\ M &= \text{\# of items that can fit into internal memory;} \\ B &= \text{\# of items per disk block,} \end{aligned}$$

where $M < N$ and $1 \le B \le M/2$. Depending on the size of the data items, typical values for workstations and file servers in production today are on the order of $M = 10^6$ or 10^7 and $B = 10^3$. Large-scale problem instances can be in the range $N = 10^{10}$ to $N = 10^{12}$.

In order to study the performance of external-memory algorithms, we use the standard notion of I/O complexity [1]. We define an *input/output operation* (or simply *I/O* for short) to be the process of reading or writing a block of data to or from the disk. The I/O complexity of an algorithm is simply the number of I/Os it performs. For example, reading all of the input data requires N/B I/Os. We use the term *scanning* to describe the fundamental primitive of reading (or writing) all items in a set stored contiguously on external storage by reading (or writing) the blocks of the set in a sequential manner.

For the problems we consider we define two additional parameters:

$$\begin{aligned} K &= \text{\# of queries in the problem instance;} \\ T &= \text{\# of items in the problem solution.} \end{aligned}$$

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

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

We say that an algorithm uses a linear number of I/O operations if it uses at most $O(n)$ I/Os to solve a problem of size N.

An increasingly popular approach to increase the throughput of I/O systems further is to use a number of disks in parallel. The number D of disks range up to 10^2 in current disk arrays. One method of using D disks in parallel is *disk striping* [31], in which the heads of the disks are moved synchronously, so that in a single I/O operation each disk reads or writes a block in the same location as each of the others. In terms of performance, disk striping has the effect of using a single large disk with block size $B' = DB$. Even though disk striping does not in theory achieve asymptotic optimality [31] when D is very large, it is often the method of choice in practice for using parallel disks, especially when D is moderately sized [29].

1.2. *Our Results*. Early work on I/O algorithms concentrated on algorithms for sorting and permutation related problems; external sorting requires $\Theta(n \log_m n)$ I/Os,[5] which is the external-memory equivalent of the well-known $\Theta(N \log N)$ time bound for sorting in internal memory [1]. More recently, researchers have designed external-memory algorithms for a number of problems in different areas, such as in computational geometry, graph theory and string processing; some encouraging experimental results regarding the practical merits of the developed algorithms have also been obtained. Refer to recent surveys for references [30], [3].

In this paper we combine and modify in novel ways several of the previously known techniques for designing efficient algorithms for external memory. In particular we use the *distribution sweeping* and *batch filtering* paradigms of [19] and the *buffer tree* data structure of [4]. In addition we also develop a powerful new technique that can be regarded as an external-memory version of batched fractional cascading on an external-memory version of a segment tree. This enables us to improve on existing external-memory algorithms as well as to develop new algorithms and thus partially answer some open problems posed in [19].

In Section 2 we introduce the *endpoint dominance problem*, which is a subproblem of *trapezoid decomposition*. We introduce an $O(n \log_m n)$-I/O algorithm to solve the endpoint dominance problem, and we use it to develop an algorithm with the same asymptotic I/O complexity for trapezoid decomposition, *planar point location*, *triangulation of simple polygons* and for the *segment sorting* problem. In Section 3 we give external-memory algorithms for line segment intersection problems. First we show how our segment sorting algorithm can be used to develop an $O(n \log_m n + t)$-I/O algorithm for *red–blue line segment intersection*, and then we discuss an $O((n + t) \log_m n)$-I/O algorithm for the *general segment intersection* problem.

Our results are summarized in Table 1. For all but the batched planar point location problem, no algorithms specifically designed for external memory were previously

[5] For convenience we define $\log_m n = \max\{1, (\log n)/\log m\}$.

Table 1. Summary of results.

Problem	I/O bound of new result	Result using mod. internal memory alg.
Endpoint dominance	$O(n \log_m n)$	$O(N \log_B n)$
Trapezoid decomposition	$O(n \log_m n)$	$O(N \log_B n)$
Batched planar point location	$O((n+k) \log_m n)$	
Triangulation	$O(n \log_m n)$	$\Omega(N)$
Segment sorting	$O(n \log_m n)$	$O(N \log_B n)$
Red–blue line segment intersection	$O(n \log_m n + t)$	$O(N \log_B n + t)$
Line segment intersection	$O((n+t) \log_m n)$	$\Omega(N)$

known. The batched planar point location algorithm that was previously known [19] only works when the planar subdivision is monotone, and the problems of triangulating a simple polygon and reporting intersections between other than orthogonal line segments are stated as open problems in [19].

For the sake of contrast, our results are also compared with modified internal-memory algorithms for the same problems. In most cases these modified algorithms are plane-sweep algorithms modified to use B-tree-based dynamic data structures rather than binary tree-based dynamic data structures, following the example of a class of algorithms studied experimentally in [14]. Such modifications lead to algorithms using $O(N \log_B n)$ I/Os. For two of the algorithms the known optimal internal-memory algorithms [10], [11] are not plane-sweep algorithms and can therefore not be modified in this manner. It is difficult to analyze precisely how those algorithms perform in an I/O environment; however, it is easy to realize that they use at least $\Omega(N)$ I/Os. The I/O bounds for algorithms based on B-trees have a logarithm of base B rather than a logarithm of base m. However, the most important difference between such algorithms and our results is the fact that the updates to the dynamic data structures are handled on an individual basis, which leads to an extra multiplicative factor of B in the I/O bound, which is very significant in practice.

As mentioned, the red–blue line segment intersection problem is of special interest because it is an abstraction of the important map-overlay problem, which is the core of several vector-based GIS [2], [23], [28]. Although a time-optimal internal-memory algorithm for the general intersection problem exists [11], a number of simpler solutions have been presented for the red–blue problem [9], [12], [20], [23]. Two of these algorithms [12], [23] are not plane-sweep algorithms, but both sort segments of the same color in a preprocessing step with a plane-sweep algorithm. The authors of [23] claim that their algorithm will perform well with inadequate internal memory owing to the fact that data are mostly referenced sequentially. A closer look at the main algorithm reveals that it can be modified to use $O(n \log_2 n)$ I/Os in the I/O model, which is only a factor of $\log m$ from optimal. Unfortunately, the modified algorithm still needs $O(N \log_B n)$ I/Os to sort the segments.

In this paper we focus our attention on the single disk model. As described in Section 1.1, striping can be used to implement our algorithms on parallel disk systems with $D > 1$. Additionally, techniques from [22] and [21] can be used to extend many of our results to parallel disk systems. In the conference version of this paper we conjectured that all our results could be improved by the optimal factor of D on parallel disk systems with D disks, but it is still an open problem whether the required merges can be done efficiently enough to allow this.

2. The Endpoint Dominance Problem.

In this section we consider the *endpoint dominance problem* (EPD) defined as follows: Given N nonintersecting line segments in the plane, find the segment directly above each endpoint of each segment.

EPD is a powerful tool for solving other important problems as we will illustrate in Section 2.1. As mentioned in the Introduction a number of techniques for designing I/O-efficient algorithms have been developed in recent years, including distribution sweeping, batch filtering [19] and buffer trees [4]. However, we do not seem to be able to solve EPD efficiently using these techniques directly. Section 2.2 briefly reviews some of the techniques and during that process we try to illustrate why they are inadequate for solving EPD. Fortunately, as we will demonstrate in Section 2.3, we are able to combine the existing techniques with several new ideas in order to develop an I/O-efficient algorithm for the problem, and thus for a number of other important problems.

2.1. *Using EPD to Solve Other Problems.*

In this section with three lemmas we illustrate how an I/O-efficient solution to EPD can be used in the construction of I/O-efficient solutions to other problems.

LEMMA 1. *If EPD can be solved in $O(n \log_m n)$ I/Os, then the trapezoid decomposition of N nonintersecting segments can be computed in $O(n \log_m n)$ I/Os.*

PROOF. We solve two instances of EPD, one to find the segments directly above each segment endpoint and one (with all y coordinates negated) to find the segment directly below each endpoint—see Figure 1 for an example of this on a simple polygon. We then compute the locations of all $O(N)$ vertical trapezoid edges. This is done by scanning through the output of the two EPD instances in $O(n)$ I/Os. To construct the trapezoids explicitly, we sort all trapezoid vertical segments by the IDs of the input segments they lie on, breaking ties by x coordinate. This takes $O(n \log_m n)$ I/Os. Finally, we scan this sorted list, in which we find the two vertical edges of each trapezoid in adjacent positions. The total amount of I/O used is thus $O(n \log_m n)$. □

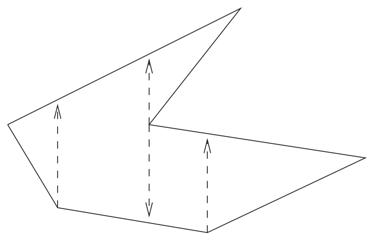

Fig. 1. Using EPD to compute the trapezoid decomposition of a simple polygon.

LEMMA 2. *If EPD can be solved in $O(n \log_m n)$ I/Os, then a simple polygon with N vertices can be triangulated in $O(n \log_m n)$ I/O operations.*

PROOF. In [18] an algorithm is given for triangulating a simple polygon in $O(N)$ time once its trapezoid decomposition has been computed. The algorithm first breaks the polygon up into *unimonotone* subpolygons using the trapezoidation. A polygon $(P_1, P_2, \ldots, P_N)$ is unimonotone if there is an i such that P_i and P_{i+1} are the vertices with minimum and maximum x coordinates and the x coordinates of $P_{i+2}, P_{i+3}, \ldots, P_{i-1}$ are in nondecreasing or nonincreasing order (all indices are modulo N). Every such subpolygon is then triangulated using a simple algorithm which takes a tour around the polygon, repeatedly cutting off convex corners. It is easy to realize that such a tour can be performed in a linear number of I/Os provided that the vertices are given in sorted order around the polygon.

In [18] the decomposition of a polygon into unimonotone subpolygons is performed by adding an edge between every pair of original vertices of the polygon which define a trapezoid in the trapezoid decomposition of the polygon, provided that they do not already share an edge. In order to create distinct subpolygons two identical edges are actually added in each relevant trapezoid (in Figure 1 two edges would be added in both of the nondegenerated trapezoids). The decomposition is done using a recursive procedure which may not be I/O-efficient. However, solving two instances of EPD and performing a constant number of sorting steps as in the proof of Lemma 1, it easy to add the edges in $O(n \log_m n)$ I/Os.

In order to rearrange the vertices such that the vertices of each unimonotone subpolygon is given in sorted order we first, in a linear number of I/Os, scan through the representation of the polygons and identify the leftmost and rightmost vertices of each polygon. Then we create a linked list L of vertices where each vertex, except for the leftmost vertex of each subpolygon, points to the one of its two neighboring vertices with the largest x coordinate. The leftmost vertex of a polygon is made to point to the rightmost vertex of another polygon such that only one left vertex points to each right vertex. L can easily be created in a linear number of I/Os. To create the desired representation we simply have to list rank L. This can be done in $O(n \log_m n)$ I/Os using algorithms from [15] and [4]. □

We define a segment $\overline{AB}$ in the plane to be above another segment $\overline{CD}$ if we can intersect both $\overline{AB}$ and $\overline{CD}$ with the same vertical line l, such that the intersection between l and $\overline{AB}$ is above the intersection between l and $\overline{CD}$. Note that two segments are incomparable if they cannot be intersected with the same vertical line. Figure 2 demonstrates that if two segments are comparable then it is enough to consider vertical lines through the four endpoints to obtain their relation. The problem of sorting N nonintersecting segments in the plane is to extending the partial order defined in the above way to a total order. This problem will become important in the solution to the red–blue line segment intersection problem in Section 3.1.

LEMMA 3. *If EPD can be solved in $O(n \log_m n)$ I/Os, then a total ordering of N nonintersecting segments can be found in $O(n \log_m n)$ I/Os.*

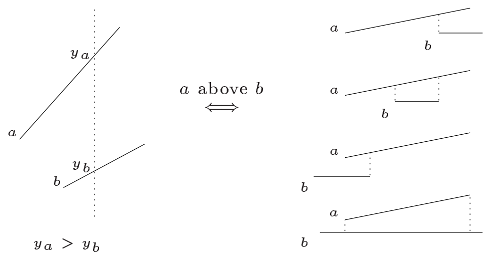

Fig. 2. Comparing segments. Two segments can be related in four different ways.

PROOF. We first solve EPD on the input segments augmented with the segment S_∞ with endpoints $(-\infty, \infty)$ and (∞, ∞). The existence of S_∞ ensures that all input segment endpoints are dominated by some segment. We define an aboveness relation $\searrow$ on elements of a nonintersecting set of segments S such that $\overline{AB} \searrow \overline{CD}$ *if and only if* either $(C, \overline{AB})$ or $(D, \overline{AB})$ is in the solution to EPD on S. Here $(A, \overline{BC})$ denotes that $\overline{BC}$ is the segment immediately above A. Similarly, we solve EPD with negated y coordinates and a special segment $S_{-\infty}$ to establish a belowness relation $\nearrow$. As discussed sorting the segments corresponds to extending the partial order defined by $\searrow$ and $\nearrow$ to a total order.

In order to obtain a total order we define a directed graph $G = (V, E)$ whose nodes consist of the input segments and the two extra segments S_∞ and $S_{-\infty}$. The edges correspond to elements of the relations $\searrow$ and $\nearrow$. For each pair of segments $\overline{AB}$ and $\overline{CD}$, there is an edge from $\overline{AB}$ to $\overline{CD}$ iff $\overline{CD} \searrow \overline{AB}$ or $\overline{AB} \nearrow \overline{CD}$. To sort the segments we simply have to sort G topologically. As G is a planar s,t-graph of size $O(N)$ this can be done in $O(n \log_m n)$ I/Os using an algorithm of [15]. □

2.2. *Buffer Trees and Distribution Sweeping*. In internal memory EPD can be solved optimally with a simple plane-sweep algorithm: We sweep the plane from left to right with a vertical line, inserting a segment in a search tree when its left endpoint is reached and removing it again when the right endpoint is reached. For every endpoint we encounter we also do a search in the tree to identify the segment immediately above the point.

In [4] a number of external-memory data structures called buffer trees are developed for use in plane-sweep algorithms. Buffer trees are data structures that can support the processing of a batch of N updates and K queries on an initially empty dynamic data structure of elements *from a totally ordered set* in $O((n + k) \log_m n + t)$ I/Os. They can be used to implement plane-sweep algorithms in which the entire sequence of updates and queries is known in advance. The queries that such plane-sweep algorithms ask of their dynamic data structures need not be answered in any particular order; the only requirement on the queries is that they must all eventually be answered. Such problems are known as *batch dynamic problems* [17], [5]. The plane-sweep algorithm for EPD sketched above can be stated as a batched dynamic problem. However, the requirement that the elements stored in the buffer tree are taken from a totally ordered set is not fulfilled in the algorithm, as we do not know any total order of the segments. Actually, as demonstrated in Lemma 3, finding such an ordering is an important application of EPD.

Therefore, we cannot use the buffer tree as the tree structure in the plane-sweep algorithm and get an I/O-efficient algorithm. For the other problems we are considering in this paper, the known internal-memory plane-sweep solutions cannot be stated as batched dynamic algorithms (since the updates depend on the queries) or else the elements involved are not totally ordered.

In [19] a powerful external memory version of the plane-sweep paradigm called distribution sweeping is introduced. Unfortunately, direct application of distribution sweeping appears insufficient to solve EPD. In order to illustrate why distribution sweeping is inadequate for the task at hand, we briefly review how it works. We divide the plane into m vertical *slabs*, each of which contains $\Theta(n/m)$ input objects, for example points or line segment endpoints. We then sweep down vertically over all of the slabs to locate components of the solution that involve interaction of objects in different slabs or objects (such as line segments) that completely span one or more slabs. The choice of m slabs is to ensure that one block of data from each slab fits in main memory. To find components of the solution involving interaction between objects residing in the same slab, we recursively solve the problem in each slab. The recursion stops after $O(\log_m n/m) = O(\log_m n)$ levels when the subproblems are small enough to fit in internal memory. In order to get an $O(n \log_m n)$ algorithm one therefore needs to be able to do one sweep in $O(n)$ I/Os. Normally this is accomplished by sorting the objects by y coordinate in a preprocessing step. This e.g. allows one to avoid having to sort before each recursive application of the technique, because as the objects are distributed to recursive subproblems their y ordering is retained. The reason that distribution sweeping fails for EPD is that there is no necessary relationship between the y ordering of endpoints of segments and their endpoint dominance relationship. In order to use distribution sweeping to get an optimal algorithm for EPD we instead need to sort the segments in a preprocessing step which leaves us with the same problem we encountered in trying to use buffer trees for EPD.

As know techniques fail to solve EPD optimally we are led instead to other approaches as discussed in the next section.

2.3. *External-Memory Segment Trees.* The *segment tree* [8], [25] is a well-known dynamic data structure used to store a set of segments in one dimension, such that given a query point all segments containing the point can be found efficiently. Such queries are called *stabbing queries*. An external-memory segment tree based on the approach in [4] is shown in Figure 3. The tree is perfectly balanced over the endpoints of the segments it represents and has branching factor $\sqrt{m/4}$. Each leaf represents $M/2$ consecutive segment endpoints. The first level of the tree partitions the data into $\sqrt{m/4}$ intervals $\sigma_1, \sigma_2, \ldots, \sigma_{\sqrt{m/4}}$, separated by dashed lines in Figure 3—for illustrative reasons we call these intervals slabs. Multislabs are defined as contiguous ranges of slabs, that is, $[\sigma_1, \sigma_1]$, $[\sigma_1, \sigma_2], \ldots, [\sigma_1, \sigma_{\sqrt{m/4}}]$, $[\sigma_2, \sigma_2], \ldots, [\sigma_2, \sigma_{\sqrt{m/4}}], \ldots, [\sigma_{\sqrt{m/4}}, \sigma_{\sqrt{m/4}}]$. There are $m/8 - \sqrt{m}/4$ multislabs. The key point is that the number of multislabs is a quadratic function of the branching factor. The reason why we choose the branching factor to be $\Theta(\sqrt{m})$ rather than $\Theta(m)$ is that we now have room in internal memory for a constant number of blocks for each of the $\Theta(m)$ multislabs. The smaller branching factor at most doubles the height of the tree.

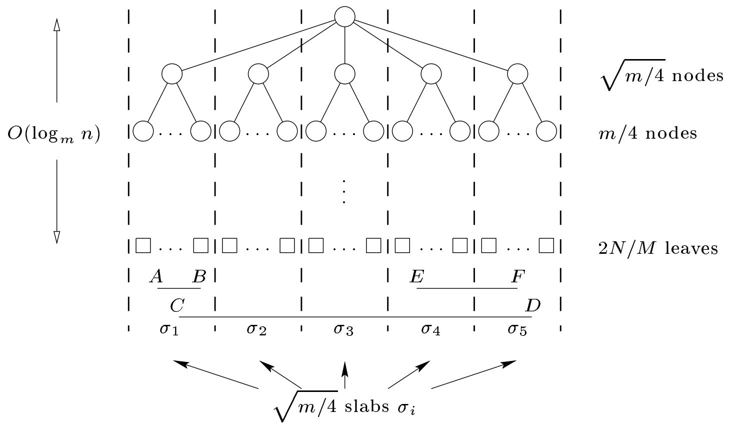

Fig. 3. An external-memory segment tree based on a buffer tree over a set of N segments, three of which, $\overline{AB}$, $\overline{CD}$, and $\overline{EF}$, are shown.

Segments such as $\overline{CD}$ that completely span one or more slabs are called *long segments*. A copy of each long segment is stored in a list associated with the largest multislab it spans. Thus, $\overline{CD}$ is stored in the *multislab list* of $[\sigma_2, \sigma_4]$. All segments that are not long are called *short segments* and are not stored in any multislab list. Instead, they are passed down to lower levels of the tree where they may span recursively defined slabs and be stored. $\overline{AB}$ and $\overline{EF}$ are examples of short segments. The portions of long segments that do not completely span slabs are treated as small segments. On each level of the tree there are at most two such synthetically generated short segments for each segment and total space utilization is thus $O(n \log_m n)$ blocks.

To answer a stabbing query, we simply proceed down a path in the tree searching for the query value. At each node we encounter, we report all the long segments associated with each of the multislabs that contain the query value.

Because of the size of the nodes and auxiliary multislab data, the buffer tree approach is inefficient for answering single queries. In batch dynamic environments, however, it can be used to develop optimal algorithms. In [4], techniques are developed for using external-memory segment trees in a batch dynamic environment such that inserting N segments in the tree and performing K queries requires $O((n + k) \log_m n + t)$ I/Os.

It is possible to come close to solving EPD by first constructing an external-memory segment tree over the projections of the segments onto the x-axis and then performing stabbing queries at the x coordinates of the endpoints of the segments. However, what we want is the single segment directly above each query point in the y dimension, as opposed to all segments it stabs. Fortunately, we are able to modify the external segment tree in order to answer a batch of this type of queries efficiently. The modification requires two significant improvements over existing techniques. First, as discussed in Section 2.3.1, we need to strengthen the definition of the structure, and the tree construction techniques of [4] must be modified in order to guarantee optimal performance when the structure is built. Second, as discussed in Section 2.3.2, the batched query algorithm must be augmented using techniques similar to fractional cascading [13].

2.3.1. *Constructing extended external segment trees*. We construct what we call an *extended external segment tree* using an approach based on distribution sweeping. An extended external segment tree is just an external segment tree as described in the last section built on nonintersecting segments in the plane, where the segments in each of the multislab lists are sorted according to the aboveness relation. Note that when we are building an external segment tree on *nonintersecting* segments we can compare all segments in the same multislab just by comparing the order of their endpoints on one of the boundaries. Before discussing how to construct an extended external segment tree I/O-efficiently we show a crucial property, namely that the segments stored in the multislab lists of a node in such a structure can be sorted efficiently. We use this extensively in the rest of the paper. When we talk about sorting segments in the multislab lists of a node we imagine that they are "cut" to the slab boundaries, that is, that we have removed the part of the segments that are stored recursively further down the structure. Note that this might result in another total order than if we considered the original segments.

LEMMA 4. *The set of N segments stored in the multislab lists of an internal node of an extended external segment tree can be sorted in $O(n)$ I/O operations*.

PROOF. We claim that we can construct a sorted list of the segments by repeatedly looking at the top segment in each of the multislabs, and selecting one which is not dominated by any of the others to go to the sorted list.

To prove the claim, assume for the sake of contradiction that there exists a top segment s in one of the multislab lists which is above the top segment in all the other multislab lists it is comparable with, but which must be below another segment t in a total order. If this is the case there exist a series of segment $s_1, s_2 \dots, s_i$ such that t is above s_1 which is above s_2 and so on ending with s_i being above s. However, if s_i is above s then so is the top segment in the multislab list containing s_i contradicting the fact that s is above the top segment in all multislab lists it is comparable with.

As the number of multislab lists is $O(m)$ there is room for a block from each of them in internal memory. Thus the sorted list can be constructed in $O(n)$ I/Os by performing a standard external-memory merge of $O(m)$ sorted lists into a single sorted list. □

In order to construct an extended external segment tree on N segments, we first use an optimal sorting algorithm to create a list of all the endpoints of the segments sorted by x-coordinate. This list is used during the whole construction algorithm to find the medians we use to split the interval associated with a given node into $\sqrt{m/4}$ vertical slabs. We now construct the $O(m)$ sorted multislab lists associated with the root in the following way: First we scan through the segments and distribute the long segments to the appropriate multislab lists. This can be done in $O(n)$ I/Os because we have enough internal memory to hold a block of segments for each multislab list. Then we sort each of these lists individually with an optimal sorting algorithm. Finally, we recursively construct an extended external segment tree for each of the slabs. The process continues until the number of endpoints in the subproblems falls below $M/2$.

Unfortunately, this simple algorithm requires $O(n \log_m^2 n)$ I/Os, because we use $O(n \log_m n)$ I/Os to sort the multislab lists on each level of the recursion. To avoid this problem, we modify our algorithm to construct the multislab lists of a node not only from a list of segments but also from two other *sorted* lists of segments. One sorted list consists of segments that have one endpoint in the x range covered by the node under construction and one to the left thereof. The other sorted list is similar but contains segments entering the range from the right. Both lists are sorted by the y coordinate at which the segments enter the range of the node being constructed. In the construction of the structure the two sorted lists will contain segments which were already stored further up the tree. We begin to build a node just as we did before, by scanning through the unsorted list of segments, distributing the long segments to the appropriate multislab lists, and then sorting each such list. Next, we scan through the two sorted lists and distribute the long segments to the appropriate multislab lists. Segments will be taken from these lists in sorted order, and can thus be merged into the previously sorted multislab lists at no additional asymptotic cost. This completes the construction of the sorted multislab lists, and now we simply have to produce the input for the algorithm at each of the $\sqrt{m/4}$ children of the current node. The $\sqrt{m/4}$ unsorted lists are created by scanning through the list of segments as before, distributing the segments with both endpoints in the same slab to the list associated with the slab in question. In this process we also handle segments with endpoints in two neighboring slabs by breaking them at the boundary between the slabs. Note that each segment will be broken at most once. The $2\sqrt{m/4}$ sorted lists of segments are constructed from the sorted multislab lists generated at the current level: First we use a linear number of I/Os to sort the segments (Lemma 4), and then the $2\sqrt{m/4}$ lists can be constructed by scanning through this sorted list of segments, distributing the segments to the appropriate of $2\sqrt{m/4}$ lists. These lists will be sorted automatically.

In the above process all the distribution steps can be done in a linear number of I/Os, because the number of lists we distribute into is always $O(m)$, which means that we have enough internal memory to hold a block of segments for each output list. Thus, each of the $O(\log_m n)$ levels of recursion uses $O(n)$ I/Os plus the number of I/Os used on sorting. Each segment participates in a sorting only once, namely the first time it is distributed to a multislab list. Thus, if N_i is the number of segments participating in the ith sorting, the overall number of I/Os used on sorting is $O(\sum_i n_i \log_m n_i) = O(n \log_m n)$:

LEMMA 5. *An extended external segment tree on N nonintersecting segments in the plane can be constructed in $O(n \log_m n)$ I/O operations.*

2.3.2. *Filtering queries through an extended tree.* Having constructed an extended external segment tree, we can now use it to find the segment directly above each of a series of K query points. In solving EPD, we have $K = 2N$, and the query points are the endpoints of the original segments. To find the segment directly above a query point p, we examine each node on the path from the root of the tree to the leaf containing p's x coordinate. At each such node, we find the segment directly above p by examining the sorted segment list associated with each multislab containing p. This segment can then be compared with the segment that is closest to the query point p so far, based on segments seen further up the tree, to see if it is the new globally closest segment. All K queries can be processed through the tree at once using a technique similar to batch filtering [19], in which *all* queries are pushed through a given level of the tree before moving onto the next level.

Unfortunately, the simple approach outlined in the preceding paragraph is not efficient. There are two problems that have to be dealt with. First, we must be able to find the position of a query point in many of the multislabs lists corresponding to a given node simultaneously. Second, searching for the position of a point in the sorted list associated with a particular multislab may require many I/Os, but as we are looking for an $O(n \log_m n)$ solution we are only allowed to use a linear number of I/Os to find the positions of *all* the query points. To solve the first problem we take advantage of the internal memory that is available to us. The second problem is solved with a notion similar to fractional cascading [12], [13]. The basic idea behind fractional cascading on internal-memory segment trees [27] is that instead of searching for the same element in a number of sorted lists of different nodes, we augment the list at a node with sample elements from lists at the children nodes. We then build "bridges" between elements in the augmented list and the corresponding elements in the lists of the children nodes. These bridges obviate the need for full searches in the lists at the children. We take a similar approach for our external-memory problem, except that we send sample elements from parents to children. Furthermore, we do not use explicit bridges. Our approach uses ideas similar to ones used in [6] and [7].

As a first step towards a solution based on fractional cascading, we preprocess the extended external segment tree in the following way (corresponding to "building bridges"): For each internal node, starting with the root, we produce a set of sample segments. For each of the $\sqrt{m/4}$ slabs (*not* multislabs) we produce a list of samples of the segments in the multislab lists that span it. The sample list for a slab consists of every $(2\sqrt{m/4}\,)$th segment in the sorted list of segments that spans it, and we "cut" the segments to the slab boundaries. All the samples are produced by scanning through the sorted list of all segments in the node produced as in Lemma 4, distributing the relevant segments to the relevant sample lists. This can be done efficiently simply by maintaining $\sqrt{m/4}$ counters during the scan, counting how many segments so far have been seen spanning a given slab. For every slab we then augment the multislab lists of the corresponding child by merging the sampled list with the multislab list of the child that contains segments spanning the whole x-interval. This merging happens before we proceed to preprocess the next level of the tree. At the lowest level of nodes, the sampled segments are passed down and stored in the leaves.

We now prove a crucial lemma about the I/O complexity of the preprocessing steps and the space use of the resulting data structure.

LEMMA 6. *The preprocessing described above uses $O(n \log_m n)$ I/Os. After the preprocessing there are still $O(N)$ segments stored in the multilists on each level of the structure. Furthermore, each leaf contains less than M segments.*

PROOF. Before any samples are passed down the tree, we have at most $2N$ segments represented at each level of the tree. Let N_i be the number of long segments, both original segments and segments sent down from the previous level, in all the nodes at level i of the tree after the preprocessing step. At the root, we have $N_0 \le 2N$. We send at most $N_i/(2\sqrt{m/4}\,) \cdot \sqrt{m/4} = N_i/2$ segments down from level i to level $i+1$. Thus, $N_{i+1} \le 2N + N_i/2$. By induction on i, we can show that for all i, $N_i \le \left(4 - (1/2)^{i-1}\right) N = O(N)$. From Lemma 4 and the fact that the number of multislab lists is $O(m)$—and that we thus can do a distribution or a merge step in a single pass of the data—it follows that each segment on a given level is read and written a constant number of times during the preprocessing phase. The number of I/Os used at level i of the tree is thus $O(n_i)$, where $n_i = N_i/B$. Since there are $O(\log_m n)$ levels, we in total use $O(n \log_m n)$ I/Os.

Before preprocessing, the number of segments stored in a node is less than the number of endpoints in the leaves below the node. To be precise, a leaf contains less than $M/2$ segments and a node i levels up the tree from a leaf contains less than $M/2 \cdot (\sqrt{m/4})^i$ segments. After preprocessing, the number of segments H_l in a leaf at level l in the tree must be $H_l \le M/2 + H_{l-1}/2\sqrt{m/4}$, where H_{l-1} is the maximal number of segments in a node at level $l-1$. This is because at most every $(2\sqrt{m/4})$th of these segments is sent down to the leaf. Thus,

$$H_l \le \frac{M}{2} + \frac{M/2 \cdot \sqrt{m/4} + H_{l-2}/2\sqrt{m/4}}{2\sqrt{m/4}} \le \frac{M}{2} + \frac{M}{4} + \frac{H_{l-2}}{(2\sqrt{m/4})^2}$$

and so on, which means that $H_l < M$. □

Having preprocessed the tree, we are now ready to filter the K query points through it. We assume without loss of generality that $K = O(N)$. If $K = \Omega(N)$ we break the queries into K/N groups of $K' = N$ queries and process each group individually. For EPD, we have $K = 2N$, so this grouping is not necessary. However, as we will see later, grouping reduces the overall complexity of processing a batch of queries when K is very large. Since our fractional cascading construction is done "backwards" (sampled segments sent downwards), we filter queries from the leaves to the root rather than from the root to the leaves. To start, we sort the K query points by their x coordinates using $O(k \log_m k)$ I/Os. We then scan the sorted list of query points to determine which leaf a given query belongs to, producing a list of queries for each leaf as indicated on Figure 4(a)). Next we iterate through the leaves, and for each query point assigned to a given leaf we find the dominating segment among the segments in the leaf. (Note that each query point must have a dominating segment since the added sample segments span the entire x-range of the leaf.) This is done by loading the entire set of segments stored at that leaf (which fits in memory according to Lemma 6), and then using an internal-memory algorithm to find the dominating segment for each query. As the total size of the data in all the leaves is $O(N)$, the total I/O complexity of the process is $O(k + n)$.

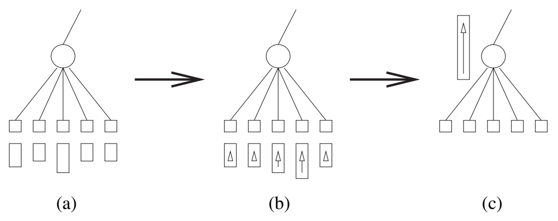

Fig. 4. Filtering queries through the structure. An arrow in a list indicates that it is sorted according to dominating segment.

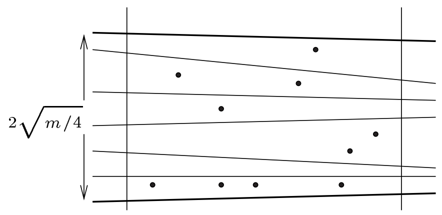

Fig. 5. All queries between sampled segments (indicated by thick lines) must appear together in the list of queries for the slab.

In order to prepare for the general step of moving queries up the tree, we sort the queries that went into each leaf by dominating segment, ending up in a situation as indicated in Figure 4(b)). This takes $O(k \log_m k)$ I/Os.

Each filtering step of the algorithm begins with a set of queries at a given level, partitioned by the nodes at that level and ordered within the nodes by the order of the segments found to be directly above them on the level. This is exactly what the output of the leaf processing was. The filtering step should produce a similar configuration on the next level up the tree. This is indicated for one node in Figure 4(c)). Remember that throughout the algorithm we also keep track of the segment found to be closest to a given query point so far, so that when the root is reached we have found the dominating segment of all query points.

To perform one filtering step on a node we merge the list of queries associated with its children (slabs) and the node's multislab lists. The key property that allows us to find the dominating segments among the segments stored in the node in an I/O-efficient manner, and sort the queries accordingly, is that the list of queries associated with a child of the node cannot be too unsorted relative to their dominating segment in the node. This is indicated in Figure 5.

In order to produce, for each slab of a node, a list of the queries in the slab, sorted according to dominating segment in the node, we again produce and scan through a sorted list of segments in the multislab lists of the node, just like when we generated the samples that were passed down the tree in the preprocessing phase. This time, however, instead of generating samples to pass down the tree, we insert a given segment in a list for each slab it spans. Thus if a segment completely spans four slabs it is inserted in four lists. When, during the scan, we encounter a segment which was sampled in slab s in the sampling phase, we stop the scan and process the queries in the list of queries for s between the sampled segment just encountered and the last sampled segment. As previously discussed these queries appear together in the sorted (according to dominating segment on the last level) list of queries for s. When this is done we clear the list of segments spanning s and continue the scan. The scan continues until all multislab segments have been processed. The crucial property is now that during the scan we can hold all the relevant segments in the main memory because at no time during the scan do we store more than $2\sqrt{m/4}$ segments for each slab, that is, $2\sqrt{m/4} \cdot \sqrt{m/4} = m/2$ segments in total. Thus we can perform the scan, not counting the I/Os used to process the queries, in a linear number of I/Os.

To process the queries in a slab between two sampled segments we maintain $2\sqrt{m/4}$ output blocks, each of which corresponds to a segment between the two sampled segments. The block for a segment is for queries with the segment as the dominating segment among the segments in the multislab list. As we read queries from the output of the child, we place them in the appropriate output block for the slab. If these output blocks become full, we write them back to the disk. Once all queries between the two sampled segments have been processed, we concatenate the outputs associated with each of the segments between the samples. This results in a list of queries sorted according to dominating segment in the node, and this list is appended to an output list for the slab. All of the above is done in a number of I/Os linear in the number of queries processed.

When we finish the above process, in a linear number of I/Os we merge the sorted output query lists of all the slabs to produce the output of the current node.

As discussed above, once this process has reached the root, we have the correct answers to all queries. The total I/O complexity of the algorithm is given by the following theorem.

THEOREM 1. *An extended external segment tree on N nonintersecting segments in the plane can be constructed, and K query points can be filtered through the structure in order to find the dominating segments for all these points, in $O((n+k)\log_m n)$ I/O operations.*

PROOF. According to Lemmas 5 and 6 construction and preprocessing together require $O(n\log_m n)$ I/Os.

Assuming $K \leq N$, sorting the K queries takes $O(n\log_m n)$ I/Os. Filtering the queries up one level in the tree takes $O(n)$ I/Os for the outer scan and $O(k)$ I/Os to process the queries. This occurs through $O(\log_m n)$ levels, giving an overall I/O complexity of $O(n\log_m n)$.

When $K > N$, we can break the problem into $K/N = k/n$ sets of N queries. Each set of queries can be answered as shown above in $O(n\log_m n)$ I/Os, giving a total I/O complexity of $O(k\log_m n)$. □

Theorem 1 immediately gives us the following bound for EPD, for which $K = 2N$.

COROLLARY 1. *The endpoint dominance problem can be solved in $O(n\log_m n)$ I/O operations.*

We then immediately get the following from Lemmas 1–3:

COROLLARY 2. *The trapezoid decomposition and the total order of N nonintersecting segments in the plane, as well as the triangulation of a simple polygon, can all be computed in $O(n\log_m n)$ I/O operations.*

It remains open whether a simple polygon can be triangulated in $O(n)$ I/Os when the input vertices are given by their order on the boundary of the polygon, which would match the linear internal-memory bound [10].

As a final direct application of our algorithm for EPD we consider the *multipoint planar point location problem*. This is the problem of reporting the location of K

query points in a planar subdivision defined by N line segments. In [19] an $O((n + k)\log_m n)$-I/O algorithm for this problem is given for *monotone* subdivisions of the plane. Using Theorem 1 we can immediately extended the result to arbitrary planar subdivisions.

LEMMA 7. *The multipoint planar point location problem can be solved using* $O((n + k)\log_m n)$ *I/O operations.*

3. Line Segment Intersection.

In this section we design algorithms for line segment intersection reporting problems. In Section 3.1 we develop an I/O-efficient algorithm for the red–blue line segment intersection problem and in Section 3.2 we develop an algorithm for the general line segment intersection problem.

3.1. *Red–Blue Line Segment Intersection.*

Using our ability to sort segments (according to the aboveness relation) as described in Section 2, we can now overcome the problems in solving the red–blue line segment intersection problem with distribution sweeping. Given input sets S_r of nonintersecting red segments and S_b of nonintersecting blue segments, we construct two intermediate sets:

$$T_\mathrm{r} = S_\mathrm{r} \cup \bigcup_{(p,q)\in S_b} \{(p, p), (q, q)\},$$

$$T_\mathrm{b} = S_\mathrm{b} \cup \bigcup_{(p,q)\in S_r} \{(p, p), (q, q)\}.$$

Each new set is the union of the input segments of one color and the endpoints of the segments of the other color (or rather zero length segments located at the endpoints). Both T_r and T_b are of size $O(|S_\mathrm{r}| + |S_\mathrm{b}|) = O(N)$. We sort both T_r and T_b using the algorithm from the previous section, and from now on assume they are sorted. This preprocessing sort takes $O(n \log_m n)$ I/Os.

We now locate intersections between the red and blue segments with a variant of distribution sweeping with a branching factor of $\sqrt{m}$. As discussed in Section 2.2, the structure of distribution sweeping is that we divide the plane into $\sqrt{m}$ slabs, not unlike the way the plane was divided into slabs to build an external segments tree in Section 2.3. We define *long segments* as those crossing one or more slabs. Segment that are not long are called *short segments*. Furthermore, we shorten the long segments by "cutting" them at the right boundary of the slab that contain their left endpoint, and at the left boundary of the slab containing their right endpoint. This may produce up to two new short segments for each long segment, and below we show how to update T_r and T_b accordingly in $O(n)$ I/Os. We also show how to report all T_i intersections between the long segments of one color and the long and short segments of the other color in $O(n + t_i)$ I/Os. Next, we use one scan to partition the sets T_r and T_b into $\sqrt{m}$ parts, one for each slab, and we recursively solve the problem on the short segments contained in each slab to locate their intersections. Each original segment is represented at most twice at each level of recursion, thus the total problem size at each level of recursion remains $O(N)$ segments. Recursion continues through $O(\log_m n)$ levels until the subproblems are of size $O(M)$ and thus can be solved in internal memory. This gives us the following result.

THEOREM 2. *The red–blue line segment intersection problem on N segments can be solved in $O(n \log_m n + t)$ I/O operations.*

Now, we simply have to fill in the details of how we process the segments on one level of the recursion. First, we consider how to insert the new points and segments generated when we cut a long segment at the slab boundaries into the sorted orders T_{r} and T_{b}. Consider a cut of a long red segment s into three parts. Changing T_{r} accordingly is easy, as we just need to insert the two new segments just before or after s in the total order. In order to insert all new red endpoints generated by cutting long red segments (which all lie on a slab boundary) in T_{b}, we first scan through T_{r} generating the points and distributing them to $\sqrt{m}$ lists, one for each boundary. The lists will automatically be sorted and therefore it is easy to merge them into T_{b} in a simple merge step. Altogether we update T_{r} and T_{b} in $O(n)$ I/Os.

Next, we consider how intersections involving long segments are found. We divide the algorithm into two parts; reporting intersections between long and short segments of different colors and between long segments of different colors.

Because T_{r} and T_{b} are sorted, we can locate interactions between long and short segments using the distribution-sweeping algorithm used to solve the orthogonal segment intersection problem in [19]. We use the algorithm twice and treat long segments of one color as horizontal segments and short segments of the other color as vertical segments. We sketch the algorithm for long red and blue short segments (details can be found in [19]): We sweep from top to bottom by scanning through the sorted list of red segments and blue endpoints T_{r}. When a top endpoint of a short blue segment is encountered, we insert the segment in an *active list* (a stack where we keep the last block in internal memory) associated with the slab containing the segment. When a long red segment is encountered we then scan through all the active lists associated with the slabs it completely spans. During this scan we know that every short blue segment in the list either is intersected by the red segment or will not be intersected by any of the following red segments (because we process the segments in sorted order), and can therefore be removed from the list. A simple amortization argument then shows that we use $O(n+t_i)$ I/Os to do this part of the algorithm.

Next we turn to the problem of reporting intersections between long segments of different colors. We define a multislab as in Section 2.3.1 to be a slab defined by two of the $\sqrt{m}$ boundaries. In order to report the intersections we scan through T_{r} and distribute the long red segments into the $O(m)$ multislabs. Next, we scan through the blue set T_{b}, and for each long blue segment we report the intersections with the relevant long red segments. This is the same as reporting intersections with the appropriate red segments in each of the multislab lists. Now consider Figure 6. A long blue segments can "interact" with a multislab in three different ways. It can have one endpoint in the multislab, it can cross the multislab completely, or it can be totally contained in the multislab. First, we concentrate on reporting intersections with red segments in multislabs for which the blue segment intersects the left boundary. Consider a blue segment b and a multislab containing its right endpoint, and define y_p to be the y coordinate of a point p. We have

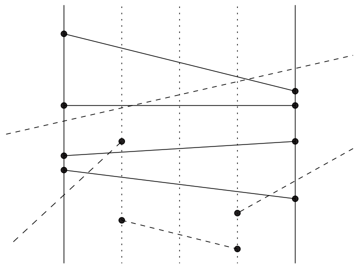

Fig. 6. Long blue segments (dashed lines) can interact with multislab in three ways.

the following:

LEMMA 8. *If a blue segment b intersects the left boundary of a multislab at point p then all blue segments reached after b in a top-down processing of the segments, if intersecting the boundary, will have the intersection point q below p.*

Let r be the left endpoint of a red segment in the multislab list. If $y_r \geq y_p$ and b intersects the red segment, then b intersects all red segments in the multislab list with left endpoints in the y-range $[y_p, y_r]$. The case $y_r \leq y_p$ is symmetric.

PROOF. The first part follows immediately from the fact that we process the segments in sorted order. Figure 7 demonstrates that the second part holds. □

Using this lemma we can now complete the design of the algorithm for our problem using a merging scheme. As discussed above, we process the blue segments in T_b one at a time and report intersections with red segments in multislab lists where the blue segments intersect the left boundary. For each such multislab list we do the following: (1) We scan *forward* from the current position in the list until we find the first red segment s_r whose left endpoint lies below the intersection between the blue segment and the multislab boundary. (2) Then we scan backward or forward as necessary in the multislab list in order to report intersections. Lemma 8 shows that the algorithm reports all intersections because all intersected segments lies consecutively above or belove s_r. Furthermore, it shows that we can use blocks efficiently. In total we in (1) only scan through each

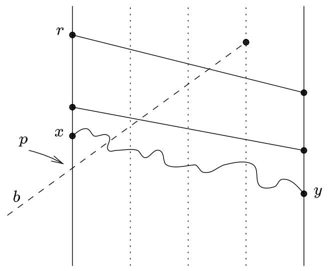

Fig. 7. Proof of Lemma 8: a segment between x and y must intersect b.

multislab list once without reporting intersections. When reporting intersections in (2) for a given blue segment and multislab list, if the report size is at least one full block we can charge a possible last nonfull block of intersections to that full block. If the report size is less than one block no further I/Os are needed. Thus, our algorithm uses a total of $O(n + t_i)$ I/Os.

This takes care of the cases where the blue segment completely spans a multislab or where it has its right, and only the right, endpoint in the multislab. The case where the blue segment only has its left endpoint in the multislab can be handled analogously. The remaining case can be handled with the same algorithm, just by distributing the blue segments instead of the red segments, and then processing one long red segment r at a time, reporting intersections with blue segments in multislab lists corresponding to multislabs completely spanned by r. Only completely spanned multislabs are considered in order to avoid reporting the same intersection twice. To summarize, we have shown how to perform one step of the distribution sweeping algorithm in $O(n + t_i)$ I/Os, and thus proven Theorem 2.

3.2. *General Line Segment Intersection*. The general line segment intersection problem cannot be solved by distribution sweeping as in the red–blue case, because the $\nearrow$ and $\searrow$ (Lemma 3) relations for sets of intersecting segments are not acyclic, and thus the preprocessing phase to sort the segments cannot be used to establish an ordering for distribution sweeping. However, as we show below, extended external segment trees can be used to establish enough order on the segments to make distribution sweeping possible. The general idea in our algorithm is to build an extended external segment tree on all the segments, and during this process to eliminate any inconsistencies that arise because of intersecting segments *on the fly*. This leads to a solution for the general problem that integrates all the elements of the red–blue algorithm into one algorithm. In this algorithm, intersections are reported both during the construction of an extended external segment tree and during the filtering of endpoints through the structure.

In order to develop the algorithm we need an external-memory priority queue [4]. Given m_p blocks of internal memory, N insert and delete-min operations can be performed on such a structure in $O(n \log_{m_p} n)$ I/Os. If we chose m_p to be m^c for some constant c $(0 < c < 1)$, we can perform the N operations using $O(n \log_m n)$ I/Os. In the construction of an extended external segment tree for general line segment intersection, we use two priority queues for each multislab. In order to have enough memory to do this, we reduce the fan-out of the extended segment tree from $\sqrt{m/4}$ to $(m/4)^{1/4}$. This does not change the asymptotic height of the tree, but it means that each node will have less than $\sqrt{m}/4$ multislabs. We chose m_p to be $\sqrt{m}$. Thus, with two priority queues per multislab, each node of the external segment tree still requires less than $m/2$ blocks of internal memory. Exactly what goes into the priority queues and how they are used will become clear as we describe the algorithm.

3.2.1. *Constructing the extended external segment tree*. In the construction of an extended external segment tree in Section 2.3.1 we used the fact that the segments did not intersect to establish an ordering on them. The main idea in our algorithm is a mechanism for breaking *long segments* into smaller pieces every time we discover an intersection during construction of the multislab lists of a node. In doing so we manage to construct an extended segment tree with no intersections between long segments stored in the multislab lists of the same node.

In order to construct the extended external segment tree on the N (now possibly intersecting) segments, we as in Section 2.3.1 first in $O(n \log_m n)$ I/Os create a sorted list of all the endpoints of the segments. The list is sorted by x coordinate, and used during the whole algorithm to find the medians we use to split the interval associated with a node into $(m/4)^{1/4}$ vertical slabs. Recall that in Section 2.3.1 one node in the tree was constructed from three lists, one sorted list of segments for each of the two extreme boundaries and one unsorted list of segments. In order to create a node we start as in the nonintersecting case by scanning through the unsorted list of segments, distributing the long segments to the appropriate multislab lists. Next, we sort the multislab lists individually according to the left segment endpoint. Finally, we scan through the two sorted lists and distribute the segments from these lists. The corresponding multislab lists will automatically be sorted according to the endpoint on one of the boundaries.

Now we want to remove inconsistencies by removing intersections between long segments stored in the multislab lists. We start by removing intersections between segments stored in the same list. To do so we initialize two external priority queues for each of the multislabs, one for each boundary. Segments in these queues are ordered according to the order of the their endpoint on the boundary in question, and the queues are structured such that a delete-min operation returns the topmost segment. We process each of the multislab lists individually as follows: We scan through the list and check if any two consecutive segments intersect. Every time we detect an intersection we report it, remove one of the segments from the list, and break it at the intersection point as indicated in Figure 8. This creates two new segments. If either one of them is long we insert it in both the priority queues corresponding to the appropriate multislab list. Any short segments that are created are inserted into a special list of segments which is distributed to the children of the current node along with normal short segments. The left part of s in Figure 8 between s_1 and s_3 is for example inserted in the queues for multislab $[s_1, s_3]$, and the part to the right of s_3 is inserted in the special list. It should be clear that after processing a multislab list in this way the remaining segments are nonintersecting (because every consecutive pair of segments are nonintersecting), and it will thus be consistently sorted. As we only scan through a multislab list once the whole process can be done in a linear number of I/Os in the number of segments processed, plus the I/Os used to manipulate the priority queues.

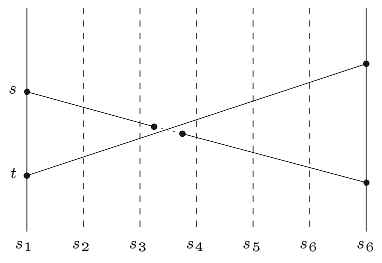

Fig. 8. Breaking a segment.

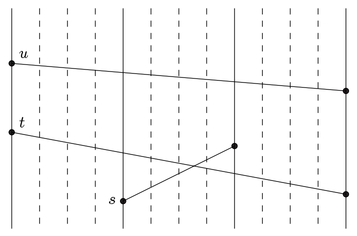

Fig. 9. Proof of Lemma 9.

Unfortunately, we still have inconsistencies in the node because segments in different multislab lists can intersect each other. Furthermore, the newly produced long segments in the priority queues can intersect each other as well as segments in the multislab lists. In order to remove the remaining intersections we need the following lemma.

LEMMA 9. *If the minimal* (*top*) *segments of all the priority queues and the top segments of all the multislab lists are all nonintersecting*, *then the topmost of them is not intersected by any long segment in the queues or lists*.

PROOF. First, consider the top segment in the two priority queues corresponding to the two boundaries of a single multislab. If these two segments do not intersect, then they must indeed be the same segment. Furthermore, no other segment in these queues can intersect this top segment. Now consider the top segment in the multislab list of the same multislab. As the two segments are nonintersecting one of them must be completely above the other. This segment is not intersected by any segment corresponding to the same multislab. Now consider this top segment in all the multislabs. Pick one of the topmost of these nonintersecting segments and call it s. Now consider Figure 9. Assume that s is intersected by another segment t in one of the queues or multislab lists. By assumption t is not the top segment in its multislab. Call the top segment in this multislab u. Because u does not intersect either t or s, and as it is above t, it also has to be above s. This contradicts the assumption that s is above all the top segments. □

Our algorithm for finding and removing intersections now proceeds as follows. We repeatedly look at the top segment in each of the priority queues and multislab lists. If any of these segments intersect, we report the intersection and break one of the segments as before. If none of the top segments intersect we know from Lemma 9 that the topmost segment is not intersected at all. This segment can then be removed and stored in a list that eventually becomes the final multislab list for the node in question. When we have processed all segments in this way, we end up with $O(m)$ sorted multislab lists of nonintersecting segments. We have enough internal memory to buffer a block from each of the lists involved in the process, so we only need a number of I/Os linear in the number of segments processed (original and newly produced ones), plus the number of I/Os used to manipulate the priority queues.

Finally, as in Section 2.3.1, we produce the input to the next level of recursion by distributing the relevant segments (remembering to include the newly produced short segments) to the relevant children. As before, this is done in a number of I/Os linear in the number of segments processed. We stop the recursion when the number of *original* endpoints in the subproblems fall below $M/4$.

If the total number of intersections discovered in the above construction process is T then the number of new segments produced is $O(T)$, and thus the number of segments stored on each level of the structure is bounded by $O(N + T)$. As in Section 2.3.1 we can argue that each segment is only contained in one list being sorted and thus we use a total of $O((n + t) \log_m(n + t)) = O((n + t) \log_m n)$ I/Os to sort the segments. In constructing each node we only use a linear number of I/Os, plus the number of I/Os used on priority queue operations. Since the number of priority queue operations is $O(T)$, the total number of I/Os we use to construct the whole structure is bounded by $O((n + t) \log_m n)$.

3.2.2. *Filtering queries through the structure*. We have now constructed an extended external segment tree on the N segments, and in the process of doing so we have reported some of the intersections between them. The intersections that we still have to report must be between segments stored in different nodes. In fact, intersections involving segments stored in a node v can only be with segments stored in nodes below v or in nodes on the path from v to the root. Therefore we will report all intersections if, for all nodes v, we report intersections between segments stored at v and segments stored in nodes below v. However, in v segments stored in nodes below v must be similar to the segments we called short in the red–blue line segment algorithm. Thus, if in each node v we had a list of endpoints of segments stored in nodes below v, sorted according to the long segment in v immediately on top of them, we could report the remaining intersections with the algorithm that was used in Section 3.1 to report intersections between long segments of one color and short segments of the other color

In order to report the remaining intersections we therefore preprocess the structure and filter the endpoints of the $O(N + T)$ segments through the structure as we did in Section 2.3.2. At each node the filtering process constructs a list of endpoints below the node sorted according to the dominating segment among the segments stored in the node. At each node we can then scan this list to collect the relevant endpoints, merge them into the sorted list of long segments, and then report intersections with the algorithm used in Section 3.1. For all nodes on one level of the structure the cost of doing so is linear in the number of segments and endpoints processed, that is, $O(n + t)$ I/Os, plus a term linear in the number of new intersections reported.

Recall, that the preprocessing of the structure in Section 2.3.2 consisted of a sampling of every $(2\sqrt{m/4})$th segment of every slab in a node, which was then augmented to the segments stored in the child corresponding to the slab. The process was done from the root towards the leaves. We will do the same preprocessing here, except that because we decreased the fanout to $(m/4)^{1/4}$ we only sample every $(2(m/4)^{1/4})$th segment in a slab. However, as we are building the structure on intersecting segments we should be careful not to introduce intersections between segments stored in the multislab lists of a node when augmenting the lists with sampled segments. Therefore we do the preprocessing while we are building the structure. Thus, in the construction process described in the previous section, after constructing the sorted multislab lists of a node, we sample every $(2(m/4)^{1/4})$th segment in each slab precisely as in Section 2.3.2. We then send these segments down to the next level together with the other "normal" segments that need to be recursively stored further down the tree. However, we want to make sure that the sampled segments are not broken, but stored on the next level of the structure. Otherwise we cannot I/O-efficiently filter the query points through the structure, as the sampled segments are stored on the next level to make sure that the points are not too unsorted relative to the segments stored in a node. Therefore we give the sampled segments a special mark and make sure that we only break unmarked segments. We can do so because two marked segments can never intersect—otherwise they would have been broken on the previous level.

By the same argument used in Section 2.3.2 to prove Lemma 6 we can prove that the augmentation of sampled segments does not asymptotically increase the number of segments stored on each level of the structure. Also all the sampling and augmentation work can be done in a linear number of I/Os on each level of the structure. This means that the number of I/Os used to construct the structure is kept at $O((n+t)\log_m n)$, even when the preprocessing is done as an integrated part of it.

After the construction and preprocessing we are ready to filter the $O(N+T)$ endpoints through the $O(\log_m n)$ levels of the structure. Recall by referring to Figure 4 that in order to do the filtering we first sort the points by x coordinate and distribute them among the leaves. Then for each leaf in turn we find the dominating segments of the points assigned to the leaf and sort the points accordingly. Finally, the points are repeatedly filtered one level up until they reach the root.

The sorting of the points by x coordinate can be done in $O((n+t)\log_m(n+t)) = O((n+t)\log_m n)$ I/Os. Also each of the filtering steps can be done in a linear number of I/Os by the same argument as in Section 2.3.2 and the previous discussion. However, our structure lacks one important feature which we used in Section 2.3.2 to find the dominating segments in the leaves. As the tree is constructed based on the sorted set of x coordinates of the *original* N segments, we can argue (as in Section 2.3.2) that a leaf represents less than $M/4$ endpoints of the *original* segments, but as $O(T)$ new segments and thus endpoints are introduced during the construction of the structure we cannot guarantee that the number of segments stored in a leaf is less than $M/2$. Potentially, all $O(T)$ additional endpoints could be assigned to one leaf, and there is no way of predicting this when the initial sorting (to construct the tree) is performed. Therefore, we cannot find the dominating segments by just loading all segments stored in a leaf into the internal memory and use an internal memory algorithm. Also, the segments stored in a leaf may intersect each other and we need to find and reports such intersections. However, assuming that we can report such intersections and produce the sorted list of endpoints for each leaf, the rest of the algorithm runs in $O((n+t)\log_m n + t')$ I/Os, where T' is the number of intersections found during the filtering of the endpoints through the structure. If $T_1 = T + T'$ is the total number of intersections reported then this number is clearly $O((n+t_1)\log_m n)$.

In order to overcome the problems with leaves containing more than M segments we treat the segments in each such leaf as a new subproblem, which we recursively solve by building an extended external segment tree on the segments and filter the relevant endpoint through this structure. The recursion process can be viewed as "growing" the leaves containing too many segments until we finally have a unified tree where all leaves contain less than M segments. In order to analyze the I/O use of the complete algorithm, consider the first level of recursion, that is, the trees build upon the relevant segments in the leaves of the initial tree. Let x_i denote the number of segments in the ith leaf of the initial tree containing more than M segments. The number of such leaves is bounded by $T/(M - M/2) = T/(M/2)$, which means that $\sum_i x_i \leq T + (T/(M/2))(M/2) = 2T \leq 2T_1$. If T_{2i} denotes the number of intersections reported when growing the subtree of leaf i and $T_2 = \sum_i T_{2i}$, the total I/O use on the first level of recursion is $\sum_i O((x_i + t_{2i}) \log_m x_i) \leq \sum_i O((x_i + t_{2i}) \log_m n) \leq O((2t_1 + t_2) \log_m n)$. The same argument can be used on the second level of recursion which involves less than $2T_2$ segments. The total number of I/Os used is thus $O(n \log_m n + 2 \sum_j t_j \log_m n) = O((n + t_t) \log_m n)$, where T_t is the total number of intersections reported.

THEOREM 3. *All T intersections between N line segments in the plane can be reported in $O((n + t) \log_m n)$ I/O operations.*

4. Conclusions.

In this paper, we have presented efficient external-memory algorithms for large-scale geometric problems involving collections of line segments in the plane. We have obtained these algorithms by combining buffer trees and distribution sweeping with a powerful new variant of fractional cascading designed for external memory.

A number of important problems, which are related to those we have discussed in this paper, remain open. Most notably if it is possible to solve the general line segment intersection reporting problem in $O(n \log_m n + t)$ I/O operations, and if it is possible to triangulate a polygon in $O(n)$ I/Os, given the N vertices in the order they appear around the perimeter of the polygon.

Acknowledgment. The authors thank an anonymous referee for comments that improved the presentation of the results in this paper.

References

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

[2] D. S. Andrews, J. Snoeyink, J. Boritz, T. Chan, G. Denham, J. Harrison, and C. Zhu. Further comparisons of algorithms for geometric intersection problems. In *Proc. 6th International Symposium on Spatial Data Handling*, 1994.

[3] L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer, Dordrecht, 2002.

[4] L. Arge. The buffer tree: a technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[5] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Theory and practice of I/O-efficient algorithms for multidimensional batched searching problems. In *Proc. ACM–SIAM Symposium on Discrete Algorithms*, pages 685–694, 1998.

[6] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: a technique for designing parallel algorithms. *SIAM Journal on Computing*, 18(3):499–532, 1989.

[7] H. Baumgarten, H. Jung, and K. Mehlhorn. Dynamic point location in general subdivisions. *Journal of Algorithms*, 17:342–380, 1994.

[8] J. L. Bentley. Algorithms for Klee's rectangle problems. Unpublished notes, Department of Computer Science, Carnegie Mellon University, 1977.

[9] T. M. Chan. A simple trapezoid sweep algorithm for reporting red/blue segment intersections. In *Proc. 6th Canadian Conference on Computational Geometry*, pages 263–268, 1994.

[10] B. Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6(5):485–524, 1991.

[11] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39:1–54, 1992.

[12] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir. Algorithms for bichromatic line-segment problems and polyhedral terrains. *Algorithmica*, 11:116–132, 1994.

[13] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[14] 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*, pages 346–357. LNCS 955. Springer-Verlag, Berlin, 1995.

[15] 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 Symposium on Discrete Algorithms*, pages 139–149, 1995.

[16] R. F. Cromp. An intellegent information fusion system for handling the archiving and querying of terabyte-sized spatial databases. In S. R. Tate (editor), *Report on the Workshop on Data and Image Compression Needs and Uses in the Scientific Community*, pages 75–84. CESDIS Technical Report Series, TR-93-99, 1993.

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

[18] A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Transactions on Graphics*, 3(2):153–174, 1984.

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

[20] H. G. Mairson and J. Stolfi. Reporting and counting intersections between two sets of line segments. In R. Earnshaw (editor), *Theoretical Foundation of Computer Graphics and CAD*, pages 307–326. NATO ASI Series, Vol. F40. Nijhoff, The Hague, 1988.

[21] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 120–129, 1993.

[22] M. H. Nodine and J. S. Vitter. Greed Sort: optimal deterministic sorting on parallel disks. *Journal of the ACM*, 42(4):919–933, 1995.

[23] L. Palazzi and J. Snoeyink. Counting and reporting red/blue segment intersections. In *Proc. Workshop on Algorithms and Data Structures*, pages 530–540. LNCS 709. Springer-Verlag, Berlin, 1993.

[24] Y. N. Patt. The I/O subsystem—a candidate for improvement. (Guest Editor's Introduction.) *IEEE Computer*, 27(3):15–16, 1994.

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

[26] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[27] V. K. Vaishnavi and D. Wood. Rectilinear line segment intersection, layered segment trees, and dynamization. *Journal of Algorithms*, 3:160–176, 1982.

[28] M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer (editors). *Algorithmic Foundations of GIS*. LNCS 1340. Springer-Verlag, Berlin, 1997.

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

[30] J. S. Vitter. External memory algorithms and data structures: dealing with MASSIVE data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[31] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.