Cache-Oblivious Red-Blue Line Segment Intersection

Lars Arge[1,⋆], Thomas Mølhave[1,⋆⋆], and Norbert Zeh[2,⋆⋆⋆]

[1] MADALGO[†], Department of Computer Science, University of Aarhus, Denmark.
E-mail: {large,thomasm}@madalgo.au.dk
[2] Faculty of Computer Science, Dalhousie University, Halifax, Nova Scotia, Canada.
E-mail: nzeh@cs.dal.ca

Abstract. We present an optimal cache-oblivious algorithm for finding all intersections between a set of non-intersecting red segments and a set of non-intersecting blue segments in the plane. Our algorithm uses $O(\frac{N}{B}\log_{M/B}\frac{N}{B}+T/B)$ memory transfers, where N is the total number of segments, M and B are the memory and block transfer sizes of any two consecutive levels of any multilevel memory hierarchy, and T is the number of intersections.

1 Introduction

The memory systems of modern computers are becoming increasingly complex; they consist of a hierarchy of several levels of cache, main memory, and disk. The access times of different levels of memory often vary by orders of magnitude and, to amortize the large access times of memory levels far away from the processor, data is normally transferred between levels in large blocks. Thus, it is important to design algorithms that are sensitive to the architecture of the memory system and have a high degree of locality in their memory access patterns.

Building on the two-level *external-memory* model [1] introduced to model the large difference between the access times of main memory and disk, the *cache-oblivious* model [8] was introduced as a way of obtaining algorithms that are efficient on *all* levels of arbitrary memory hierarchies. In this paper, we develop a cache-oblivious algorithm for the *red-blue line segment intersection problem*,

⋆ Supported in part by the US Army Research Office through grant W911NF-04-01-0278, by an Ole Roemer Scholarship from the Danish National Science Research Council, a NABIIT grant from the Danish Strategic Research Council, and by the Danish National Research Foundation.
⋆⋆ Supported in part by an Ole Roemer Scholarship from the Danish National Science Research Council, a NABIIT grant from the Danish Strategic Research Council, and by the Danish National Research Foundation.
⋆⋆⋆ Supported by the Canada Research Chairs program, the Natural Sciences and Engineering Research Council of Canada, and the Canadian Foundation for Innovation.
† Center for Massive Data Algorithmics, a Center of the Danish National Research Foundation.

that is, for finding all intersections between a set of non-intersecting red segments and a set of non-intersecting blue segments in the plane. Our algorithm is optimal and, to the best of our knowledge, the first efficient cache-oblivious algorithm for any intersection problem involving non-axis-parallel objects.

External-memory model. In the two-level *external-memory model* [1], the memory hierarchy consists of an *internal memory* big enough to hold M elements and an arbitrarily large *external memory* partitioned into blocks of B consecutive elements. A *memory transfer* moves one block between internal and external memory. Computation can occur only on data in internal memory. The complexity of an algorithm in this model (an *external-memory algorithm*) is measured in terms of the number of memory transfers it performs. Aggarwal and Vitter proved that the number of memory transfers needed to sort N data items in the external-memory model is $\mathrm{Sort}(N) = \Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ [1]. Subsequently, a large number of algorithms have been developed in this model; see [2, 10] for an overview. Below we briefly review results directly related to our work.

In the first paper to consider computational geometry problems in external memory [9], Goodrich et al. introduced the *distribution sweeping* technique (a combination of M/B-way distribution sort and plane sweeping) and showed how it can be used to solve a large number of geometric problems in the plane using $O(\mathrm{Sort}(N) + T/B)$ memory transfers, where T is the output size of the problem (eg., number of intersections). The problems they considered include the orthogonal line segment intersection problem and other problems involving axis-parallel objects. Arge et al. developed an algorithm that solves the red-blue line segment intersection problem using $O(\mathrm{Sort}(N)+T/B)$ memory transfers [4], which is optimal. The algorithm uses the distribution sweeping technique [9] and introduces the notion of *multi-slabs*; if the plane is divided into vertical slabs, a multi-slab is defined as the union of any number of consecutive slabs. Multi-slabs are used to efficiently deal with segments spanning a range of consecutive slabs. The key is that, if there are only $\sqrt{M/B}$ slabs, there are less than M/B multi-slabs, which allows the distribution of segments into multi-slabs during a plane sweep using standard M/B-way distribution. Arge et al. also extended their algorithm to obtain a solution to the general line segment intersection problem using $O(\mathrm{Sort}(N + T))$ memory transfers [4].

Cache-oblivious model. In the *cache-oblivious model* [8], the idea is to design a standard RAM-model algorithm that has not knowledge of the parameters of the memory hierarchy but analyze it in the external-memory model assuming that an offline optimal paging strategy performs the memory transfers necessary to bring accessed elements into memory. Often it is also assumed that $M \geq B^2$ (the *tall-cache assumption*). The main advantage of the cache-oblivious model is that it allows us to reason about a simple two-level memory model but prove results about an unknown, multi-level memory hierarchy [8].

Frigo et al. [8] developed optimal cache-oblivious sorting algorithms, as well as algorithms for a number of other fundamental problems. Subsequently, algorithms and data structures for a range of problems have been developed [3]. Relevant to this paper, Bender et al. [5] developed a cache-oblivious algorithm

that solves the offline planar point location problem using $O(\text{Sort}(N))$ memory transfers; Brodal and Fagerberg [6] developed a cache-oblivious version of distribution sweeping and showed how to use it to solve the orthogonal line segment intersection problem, as well as several other problems involving axis-parallel objects, cache-obliviously using $O(\text{Sort}(N) + T/B)$ memory transfers. To the best of our knowledge, no cache-oblivious algorithm was previously known for any intersection problem involving *non-axis-parallel* objects.

Our results. We present a cache-oblivious algorithm for the red-blue line segment intersection problem that uses $O(\text{Sort}(N) + T/B)$ memory transfer. This matches the bound of the external-memory algorithm of [4] and is optimal.

As discussed, the external-memory algorithm for this problem [4] is based on an extended version of distribution sweeping utilizing multi-slabs. Our new algorithm borrows ideas from both the external-memory algorithm for the red-blue line segment intersection problem [4] and the cache-oblivious algorithm for the orthogonal line-segment intersection problem [6]. In order to obtain a useful notion of sweeping the plane top-down or bottom-up, we utilize the same total ordering as in [4] on a set of non-intersecting segments, which arranges the segments intersected by any vertical line in the same order as the y-coordinates of their intersections with the line. In the case of axis-parallel objects, such an ordering is equivalent to the y-ordering of the vertices of the objects; in the non-axis-parallel case, this ordering is more difficult to obtain [4]. Similar to the cache-oblivious orthogonal line-segment intersection algorithm [6], we employ the cache-oblivious distribution sweeping paradigm, which uses two-way merging rather than $\sqrt{M/B}$-way distribution. While this eliminates the need for multi-slabs, which do not seem to have an efficient cache-oblivious counterpart, it also results in a recursion depth of $\Theta(\log_2 N)$ rather than $\Theta(\log_{M/B} N)$. This implies that one cannot afford to spend even $1/B$ memory transfers per line segment at each level of the recursion. For axis-parallel objects, Brodal and Fagerberg [6] addressed this problem using the so-called k-merger technique, which was introduced as the central idea in Funnel Sort (ie., cache-oblivious Merge Sort) [8]. This technique allows N elements to be passed through a $\log_2 N$-level merge process using only $O(\text{Sort}(N))$ memory transfers, but generates the output of each merge process in bursts, each of which has to be consumed by the next merge process before the next burst is produced. This creates a new challenge, as a segment may have intersections with all segments in the output stream of a given merge process and, thus, needs access to the entire output stream to report these intersections. To overcome this problem, Brodal and Fagerberg [6] provided a technique to detect, count, and collect intersected segments at each level of recursion that ensures that the number of additional accesses needed to report intersections is proportional to the output size.

Our main contribution is the development of non-trivial new methods to extend the counting technique of Brodal and Fagerberg [6] to the case of non-axis-parallel line segments. These ideas include a *look-ahead* method for identifying certain critical segments ahead of the time they are accessed during a merge, as well as an *approximate counting* method needed because exact counting of

intersected segments (as utilized in the case of axis-parallel objects) seems to be no easier than actually reporting intersections.

2 Vertically Sorting Non-Intersecting Segments

In this section, we briefly sketch a cache-oblivious algorithm to vertically sort a set S of N non-intersecting segments in the plane. Let s_1 and s_2 be segments in S. We say that s_2 is *above* s_1, denoted $s_1 <_A s_2$, if there exists a vertical line intersecting s_1 and s_2 in points (x, y_1) and (x, y_2), respectively, and $y_1 < y_2$. Some segments in S may be incomparable under $<_A$, and the problem of vertically sorting S is to extend the partial order $<_A$ to a total order $<_t$ such that $s_1 <_A s_2$ implies $s_1 <_t s_2$ [4]. We call $<_t$ a *vertical ordering* of the segments.

Our cache-oblivious algorithm for vertically sorting S is an adaptation of the corresponding external-memory algorithm [4]. The main ingredients are an algorithm for finding the segments immediately above and below every segment endpoint and an algorithm for topologically sorting the resulting planar st-graph. The former can be solved using an offline cache-oblivious point location algorithm [5]; for the latter we use a cache-oblivious adaptation of the external-memory algorithm [7]. Details will appear in the full paper.

Theorem 1. *A vertical ordering of N non-intersecting line segments in the plane can be computed cache-obliviously using $O(\mathrm{Sort}(N))$ memory transfers and linear space.*

3 Red-Blue Line Segment Intersection

In this section, we give an overview of our algorithm for finding all intersections between a set R of non-intersecting red segments and a set B of non-intersecting blue segments. For simplicity we assume that the x- and y-coordinates of all endpoints are distinct. Sections 4 and 5 present the details of our algorithm.

The $\sqrt{N}$-merger. Our algorithm uses the $\sqrt{N}$-merger technique [6,8] extensively. A $\sqrt{N}$-merger merges $\sqrt{N}$ sorted input streams of length $\sqrt{N}$ into one sorted output stream. It is defined recursively in terms of smaller k-mergers. A k-merger takes k sorted input streams of total length at least k^2 and produces a sorted output stream by merging the input streams. The cost of merging k^2 elements using a k-merger is $O(\mathrm{Sort}(k^2))$, which is $O(\mathrm{Sort}(N))$ for $k = \sqrt{N}$ [6,8].

A k-merger is a complete binary tree over $k/2$ leaves with a buffer associated with each edge. If $k = 2$, the merger consists of a single node with two input streams and one output stream; see Fig. 1(a). Otherwise, it consists of $\sqrt{k}+1$ $\sqrt{k}$-mergers as shown in Fig. 1(b); the buffers associated with the edges between the top merger and the bottom mergers have size k. The merge process is performed by invoking a FILL operation on the root of the merger. A FILL operation on a node u fills the output buffer $S(u)$ of u (the buffer between u and its parent) by

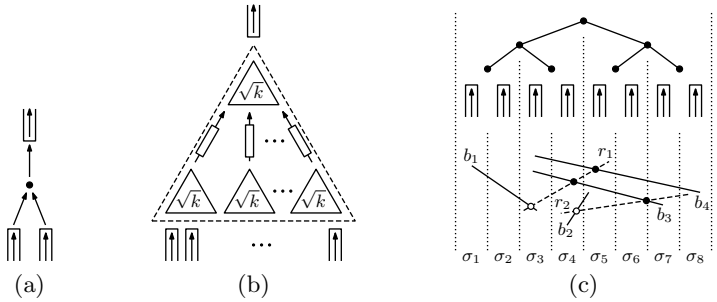

Fig. 1. (a) A 2-merger. (b) A k-merger for $k > 2$. (c) Slabs and intersection types.

repeatedly removing the minimum element from $S(l(u))$ or $S(r(u))$ and placing it into $S(u)$, where $l(u)$ and $r(u)$ denote the left and right children of u. When $S(l(u))$ or $S(r(u))$ becomes empty, a FILL operation is invoked recursively on the corresponding child before continuing to fill $S(u)$. The FILL operation returns when $S(u)$ is full or there are no elements left in any buffer below u. Since the root's output buffer has size N, only one FILL operation on the root is required to place all elements in the input streams into a sorted output stream.

The basic concept in the analysis of a $\sqrt{N}$-merger is that of a *base tree*, which is the largest subtree in the recursive definition of a $\sqrt{N}$-merger such that the entire tree plus one block for each of its input and output buffers fit in memory. The central observation is that, in order to achieve the $O(\mathrm{Sort}(k^2))$ merge bound, a FILL operation on a base tree root can afford to load the whole base tree into memory and perform $O(1)$ memory transfers per node in the base tree; note that this means that FILL operations on other nodes of the base tree are free. It also means that we can associate $O(1)$ auxiliary buffers with each merger node u and that we can assume that a FILL operation at node u can access the first $O(1)$ blocks of each auxiliary buffer without any memory transfers. See [6] for details.

Distribution sweeping. To find all intersections between red and blue segments, we start by dividing the plane into $q = \sqrt{N}$ vertical slabs $\sigma_1, \ldots, \sigma_q$ containing $2\sqrt{N}$ segment endpoints each, where $N = |R| + |B|$ is the total number of segments. We recurse on each slab σ_i to find the intersections in σ_i between segments with at least one endpoint in this slab; these intersections are shown using white dots in Fig. 1(c). Each of the remaining intersections, shown as black dots in Fig. 1(c), involves at least one segment that completely spans the slab containing the intersection. To find these intersections, we use a $\sqrt{N}$-merger whose input streams are sorted lists of segments and/or segment endpoints associated with slabs $\sigma_1, \ldots, \sigma_q$. We also associate slabs with the nodes of the merger. The slab σ_u associated with a node u is the union of the slabs corresponding to the input streams of u's subtree. We use $l(\sigma_u)$ and $r(\sigma_u)$ to denote its left and right boundaries, respectively. We call a segment with an endpoint in σ_u *long* wrt. slab $\sigma_{l(u)}$ if it spans $\sigma_{l(u)}$ (segment b_3 in Fig. 2(a)), and *short* otherwise

(segments b_1, b_2, b_4 in Fig. 2(a)). We call an intersection in $\sigma_{l(u)}$ *long-long* if it involves two long segments wrt. slab $\sigma_{l(u)}$ (point p_3 in Fig. 2(a)), and *short-long* if it involves a short and a long segment (points p_1 and p_2 in Fig. 2(a)). Short and long segments and short-long and long-long intersections in slab $\sigma_{r(u)}$ are defined analogously. It is easy to see that every intersection in a slab σ_i that involves a segment spanning σ_i is long-long or short-long at exactly one merger node. Hence, our goal in merging the streams corresponding to slabs $\sigma_1, \ldots, \sigma_q$ is to report all long-long and short-long intersections at each merger node.

Throughout this paper, we only discuss finding, at every merger node u, short-long and long-long intersections inside $\sigma_{l(u)}$. The intersections in $\sigma_{r(u)}$ can be found analogously. Our algorithm finds short-long and long-long intersections separately and finds each intersection type using several applications of the $\sqrt{N}$-merger to appropriate input streams associated with slabs $\sigma_1, \ldots, \sigma_q$. We call one such application a *pass* through the merger. In the process of merging the input streams of the merger, each pass either reports intersections or performs some preprocessing to allow a subsequent pass to report intersections. As we show in Sect. 4 and 5, $O(1)$ passes are sufficient to report all short-long and long-long intersections, and each pass uses $O(\mathrm{Sort}(N) + T_s/B)$ memory transfers and linear space, where T_s is the number of reported intersections. Let N_i denote the number of short segments in slab σ_i, T_i the number of intersections between these segments, and $C(N, T)$ the complexity of our algorithm on N segments that have T intersections. Then the complexity of our algorithm is given by the recurrence $C(N, T) = \sum_{i=1}^{\sqrt{N}} C(N_i, T_i) + O(\mathrm{Sort}(N) + T_s/B)$, which solves to $C(N, T) = O(\mathrm{Sort}(N) + T/B)$ because each original segment participates as a non-spanning segment in at most two slabs on each level of the recursion.

Theorem 2. *The red-blue line segment intersection problem can be solved cache-obliviously using $O(\mathrm{Sort}(N) + T/B)$ memory transfers and linear space, where N is the total number of line segments and T is the number of intersections.*

4 Short-Long Intersections

In this section, we discuss how to find all short-long intersections at all merger nodes using $O(1)$ passes through the merger. Recall that we focus only on intersections inside $\sigma_{l(u)}$. We call such an intersection between a long red segment r and a short blue segment b *upward* if b has at least one endpoint in $\sigma_{l(u)}$ that is below r (points p_2, p_3, p_5 in Fig. 2(b)); otherwise, the intersection is *downward* (points p_1 and p_4 in Fig. 2(b)). We focus on finding upward short-long intersections between long red and short blue segments in the remainder of this section. The other types of short-long intersections can be found analogously. We discuss first how to find these intersections in the desired number of memory transfers using linear extra space per merger node. Then we discusses how to reduce the space bound to $O(N)$ in total.

Our algorithm uses two passes through the $\sqrt{N}$-merger. The first pass associates a *red list* $R(u)$ of size N (big enough to hold all segments in the input

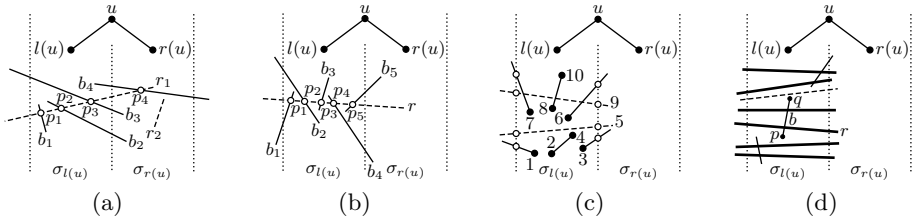

Fig. 2. (a) Short-long and long-long intersections. (b) Upward and downward intersections. (c) Detecting long segments involved in upward short-long intersections. (d) Reporting upward short-long intersections. Dashed segments are not in $R(u)$.

if necessary) with every merger node u and populates it with all red segments that are long wrt. $\sigma_{l(u)}$ and are involved in upward short-long intersections at node u. The second pass uses these red lists to report all upward short-long intersections. Both passes merge segment streams sorted by the vertical segment ordering from Sect. 2. More precisely, we construct a set R' containing all red segments and one zero-length segment per blue segment endpoint and use the vertical ordering on R' as a total ordering of red segments and blue segment endpoints, bottom-up. The *rank* of a red segment or blue segment endpoint is its position in this ordering.

Populating red lists. To populate all red lists, we initialize the input streams of the merger so that the stream corresponding to slab σ_i stores all red segments whose right endpoints are in σ_i, as well as all blue segment endpoints in σ_i. The entries of the stream are sorted bottom-up (by increasing rank). Now we merge these streams to produce one sorted output stream, where the output stream of each merger node u contains all red segments with right endpoints in σ_u and all blue segment endpoints in σ_u, again sorted bottom-up. The FILL operation at a node u is the standard FILL operation of a $\sqrt{N}$-merger, except that, when placing a red segment r into u's output stream $S(u)$, we check whether r is involved in an upward short-long intersection at node u. If it is, we also append segment r to u's red list $R(u)$.

To see how this test is performed, consider an upward short-long intersection between a short blue segment b and a long red segment r. Segment b must have at least one endpoint in $\sigma_{l(u)}$ that is below r (has lower rank than r). Since b and r intersect in $\sigma_{l(u)}$, either b's other endpoint q also lies in $\sigma_{l(u)}$ and is above r (has higher rank than r), or b intersects one of the slab boundaries of $\sigma_{l(u)}$ above r; see Fig. 2(c). Since we merge segments and segment endpoints at each node u bottom-up, we process (ie., place into $S(u)$) all short blue segment endpoints below r before we process r. We call a blue segment *processed* if we have processed at least one of its endpoints. A segment b with one endpoint in $\sigma_{l(u)}$ is *internal*, *left-intersecting*, or *right-intersecting* depending on whether both its endpoints are in $\sigma_{l(u)}$, b intersects $l(\sigma_{l(u)})$ or b intersects $r(\sigma_{l(u)})$. Let $\rho(u)$ be the highest rank of all endpoints of processed internal blue segments, and

$y_l(u)$ the y-coordinate of the highest intersection between $l(\sigma_{l(u)})$ and processed left-intersecting blue segments; $y_r(u)$ is defined analogously for processed right-intersecting blue segments. By our previous discussion, r has an upward short-long intersection at u if and only if r has rank less than $\rho(u)$, intersects $l(\sigma_{l(u)})$ below y-coordinate $y_l(u)$ or intersects $r(\sigma_{l(u)})$ below $y_r(u)$; see Fig. 2(c).

Values $\rho(u)$, $y_l(u)$, and $y_r(u)$ are easily maintained as the FILL operation at node u processes blue segment endpoints. When processing a red segment r, it is easy to test whether it is long wrt. $\sigma_{l(u)}$ and its rank is less than $\rho(u)$, its intersection with $l(\sigma_{l(u)})$ has y-coordinate less than $y_l(u)$ or its intersection with $r(\sigma_{l(u)})$ has y-coordinate less than $y_r(u)$. If this is the case, r has at least one upward short-long intersection at u, and we append it to u's red list $R(u)$.

Reporting short-long intersections. Given the populated red lists, the second pass starts out with the input stream of each slab σ_i containing all blue segment endpoints in σ_i, sorted top-down (ie., by decreasing ranks). We merge these points so that every node u outputs a stream of blue segment endpoints in σ_u, sorted top-down. To report all short-long intersections at a node u, the FILL operation at node u keeps track of the *current position* in $R(u)$, which is the segment with minimum rank in $R(u)$ we have inspected during the current pass. Initially, this is the last segment in $R(u)$. Now when processing an endpoint $p \in \sigma_{l(u)}$ of a blue segment b, we first scan backwards in $R(u)$ from the current position to find the segment r with minimum rank in $R(u)$ whose rank is greater than that of p. Segment r becomes the new current position in $R(u)$. Segment r is the lowest segment in $R(u)$ that can have an upward intersection with b, and all segments having such intersections with b form a contiguous sequence in $R(u)$ starting with r. Therefore, we scan forward from r, reporting intersections between scanned segments and b until we find the first segment in $R(u)$ that does not have an upward short-long intersection with b; see Fig. 2(d).

Since every segment placed into $R(u)$ is involved in at least one intersection and all but $O(1)$ accesses to a segment in $R(u)$ can be charged to reported intersections, the scanning of red lists adds only $O(T_s/B)$ to the $O(\mathrm{Sort}(N))$ cost of the merger. The space usage of the algorithm can be reduced to $O(N + T_s)$ by running the pass populating red lists twice. The first time, we only count segments that would be placed into each list and then allocate a list of the appropriate size to each node. The second time, we place segments into the allocated lists. Using the same technique as in [6], the space can then be reduced further to $O(N)$. Details will appear in the full paper.

Lemma 1. *Short-long intersections can be reported using* $O(\mathrm{Sort}(N) + T_s/B)$ *memory transfers and linear space.*

5 Long-Long Intersections

In this section, we discuss how to find the long-long intersections at all merger nodes. Again, we focus on finding, at every node u, only long-long intersections

inside slab $\sigma_{l(u)}$. Similar to the short-long case, we first describe our procedure assuming we can allocate *two* lists of size N to each node. Later we discuss how to reduce the space usage to $O(N)$.

A simple solution using superlinear space. After some preprocessing discussed later in this section, long-long intersections can be found using one pass through the $\sqrt{N}$-merger. This time, the input stream corresponding to slab σ_i contains all segments whose right endpoints are inside σ_i and which intersect $l(\sigma_i)$. The segments are sorted by decreasing y-coordinates of their intersections with $l(\sigma_i)$. The goal of the merge process at a merger node u is to produce an output stream of all segments with right endpoints in σ_u and which intersect $l(\sigma_u)$. Again, these segments are to be output sorted by decreasing y-coordinates of their intersections with $l(\sigma_u)$. In the process of producing its output stream, each merger node u reports all long-long intersections inside $\sigma_{l(u)}$.

This merge process in itself poses a challenge compared to the short-long case, as segments in $S(r(u))$ that intersect both $r(\sigma_{l(u)})$ and $l(\sigma_{l(u)})$ may have to be placed into $S(u)$ in a different order from the one in which they arrive in $S(r(u))$; see Fig. 3(a). Thus, we need to allow segments to "pass each other", which we accomplish using two buffers $B(u)$ and $R(u)$ of size N associated with each node u in the merger. Buffer $B(u)$ is used to temporarily hold blue segments that need to be overtaken by red segments at u; these segments are sorted by the y-coordinates of their intersections with $l(\sigma_u)$. Buffer $R(u)$ serves the same purpose for red segments. Initially, $B(u)$ and $R(u)$ are empty.

To implement the merge process, we also need a "look-ahead" mechanism that allows each node u to identify the next long segment of each color to be retrieved from $S(r(u))$ without actually retrieving it. We discuss below how to provide such a mechanism. Again, the need for such a mechanism arises because long red and blue segments may change their order between $S(r(u))$ and $S(u)$. If the topmost segment b in $S(r(u))$ is long and blue, we can decide whether it is the next segment to be placed into $S(u)$ only if we know whether the next long red segment r intersects $l(\sigma_u)$ above b; but there may be an arbitrary number of blue and short red segments between b and r in $S(r(u))$, and we cannot afford to scan ahead until we find r in $S(r(u))$. Look-ahead provides us with r without the need to scan through $S(r(u))$.

A FILL operation at node u now reduces to repeatedly identifying the next segment s to be placed into $S(u)$. This segment is currently in $S(l(u))$, $S(r(u))$, $R(u)$ or $B(u)$ and is the one with the highest intersection with $l(\sigma_u)$ among the segments remaining in these streams. Thus, if s belongs to $S(l(u))$, it must be the next segment s' in $S(l(u))$ because the segments in $S(l(u))$ are sorted by their intersections with $l(\sigma_{l(u)}) = l(\sigma_u)$. If s belongs to $S(r(u))$, $R(u)$ or $B(u)$, it must be the next long red segment r or the next long blue segment b to be placed into $S(u)$. Note that our look-ahead mechanism provides us with r and b. To decide which of s', r, and b is the next segment s to be placed into $S(u)$, it suffices to compare their intersections with $l(\sigma_u)$.

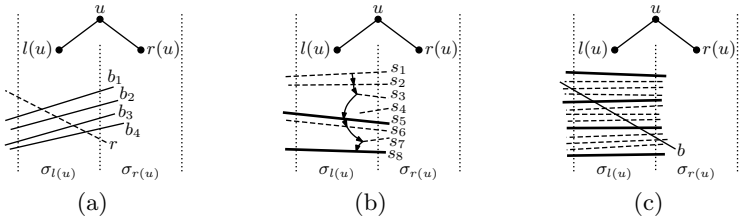

Fig. 3. (a) Segments b_1, b_2, b_3, b_4 arrive before r in $S(r(u))$ but need to be placed into $S(u)$ after r. Thus, r must be able to overtake them at u. (b) Implementation of look-ahead. Bold solid segments are in $R_t(u)$, dashed ones are not. Arrows indicate how every long segment finds the next long segment. (c) Approximate counting using sampling. The bold segments are in the sample, the dashed ones are not.

In order to place s into $S(u)$, we need to locate it in $S(l(u))$, $S(r(u))$, $B(u)$ or $R(u)$, remove it, and output it into $S(u)$. If $s \in S(l(u))$, $B(u)$ or $R(u)$, this is easy because s is the next segment in $S(l(u))$ or the first segment in $B(u)$ or $R(u)$. So assume that s is long, wlog. red, and stored in $S(r(u))$. Then we retrieve segments from $S(r(u))$ until we retrieve s. Since the segments in $S(r(u))$ are sorted by their intersections with $l(\sigma_{r(u)})$ and red segments do not intersect, there cannot be any long red segment in $S(r(u))$ that is retrieved before s. Thus, all segments retrieved from $S(r(u))$ before s are blue or short. Short segments can be discarded because they cannot be involved in any long-long intersections at u or any of its ancestors. Long blue segments are appended to $B(u)$ in the order they are retrieved, which is easily seen to maintain the segments in $B(u)$ sorted by their intersections with $l(\sigma_{l(u)})$.

So far we have talked only about outputting the segments at each node u in the correct order. To discuss how to report intersections, we say that a segment is placed into $S(u)$ *directly* if it is never placed into $R(u)$ or $B(u)$; otherwise, we say that it is *overtaken* by at least one segment. It is not hard to see that every long-long intersection at a node u involves a segment s placed directly into $S(u)$ and a segment that is overtaken by s; a segment s placed directly into $S(u)$ has long-long intersections with exactly those segments of the other color that are in $B(u)$ or $R(u)$ at the time when s is placed into $S(u)$. Thus, we can augment the merge process at u to report long-long intersections as follows. Immediately before placing a long red segment r directly into $S(u)$, we scan $B(u)$ to report all intersections between r and the segments in $B(u)$. When a long blue segment b is placed directly into $S(u)$, we scan $R(u)$ instead. Since only segments that are overtaken (and thus involved in at least one intersection) are placed into $R(u)$ and $B(u)$ and every scan of $R(u)$ and $B(u)$ reports one intersection per scanned segment, the manipulation of these buffers at all merger nodes adds only $O(T_s/B)$ memory transfers to the $O(\mathrm{Sort}(N))$ cost of the merger. Next we discuss how to implement the look-ahead mechanism using only $O(\mathrm{Sort}(N))$ additional memory transfers, which leads to an $O(\mathrm{Sort}(N) + T_s/B)$ cost for finding all long-long intersections.

Look-ahead. Consider the merge process reporting long-long intersections at a node u. Given look-ahead at u's children, it is easy to ensure that every segment in $S(l(u))$ or $S(r(u))$ knows the next segment s' of the same color in $S(l(u))$ or $S(r(u))$, respectively. When placing a long segment s from $S(r(u))$ into $S(u)$, however, we need to identify not the next segment of the same color as s in $S(r(u))$ but the next *long* such segment s''. If s' is long, then $s'' = s'$. Otherwise, we say that s' *terminates* at node u, as it is not placed into $S(u)$. In this case, s' comes between s and s'' in $S(r(u))$. Note also that every segment terminates at exactly one node in the merger.

To allow us to identify segment s'', we preprocess the merger and associate two lists $R_t(u)$ and $B_t(u)$ with every node u. List $R_t(u)$ (resp., $B_t(u)$) contains all those long red (resp., blue) segments in $S(r(u))$ that are immediately preceded by red (resp., blue) segments that terminate at u. Given these lists, a long segment s in $S(r(u))$ that is succeeded by a terminating segment of the same color in $S(r(u))$ can identify the next long segment of the same color by retrieving the next segment from $R_t(u)$ or $B_t(u)$, depending on its color; see Fig. 3(b). These lists are easily constructed in $O(\mathrm{Sort}(N))$ memory transfers by merging the blue and red segments independently; details will appear in the full paper. In order to ensure that each list uses only as much space as it needs—and, thus, that all look-ahead lists use only $O(N)$ space—we run each merge twice. The first pass counts the number of segments to be placed into each list, the second one populates the lists after allocating the required space to each list.

During the merge that reports long-long intersections, each list $R_t(u)$ or $B_t(u)$ is scanned exactly once, as the segments in these lists are retrieved in the order they are stored. Thus, scanning these lists uses $O(N/B)$ memory transfers.

Linear space via approximate counting of intersected segments. Finally, we discuss how to reduce the space usage of the merge that finds long-long intersections to $O(N + T_s)$. Using the same technique as in [6] again, the space usage can then be reduced further to $O(N)$. Details appear in the full paper.

To achieve this space reduction, we need to reduce the total size of the red and blue buffers $R(u)$ and $B(u)$ to $O(N+T_s)$. We observe that $R(u)$ and $B(u)$ never contain more than $c_b(u)$ and $c_r(u)$ segments, respectively, where $c_b(u)$ and $c_r(u)$ denote the maximum number of red (resp., blue) segments intersected by any long blue (resp., red) segment at u. Hence, it suffices to determine these values and allocate $c_b(u)$ space for $R(u)$ and $c_r(u)$ space for $B(u)$. Since these values summed over all nodes of the merger do not sum to more than T_s, this would ensure that the total space usage of all buffers $R(u)$ and $B(u)$ is at most T_s. However, it seems difficult to determine $c_b(u)$ and $c_r(u)$ exactly without already using buffers $R(u)$ and $B(u)$. Instead, we compute upper bounds $c'_b(u)$ and $c'_r(u)$ such that $c_b(u) \leq c'_b(u) \leq c_b(u) + \sqrt{N}$ and $c_r(u) \leq c'_r(u) \leq c_r(u) + \sqrt{N}$, which can be done in linear space. By allocating $c'_b(u)$ space for buffer $R(u)$ and $c'_r(u)$ space for buffer $B(u)$, each buffer is big enough and we waste only $O(\sqrt{N})$ space per merger node. Since there are $O(\sqrt{N})$ merger nodes, the total space used by all buffers is therefore $O(N + T_s)$.

We discuss how to compute values $c'_b(u)$, as values $c'_r(u)$ can be computed similarly. To compute values $c'_b(u)$, we compute a $\sqrt{N}/2$-sample of the long red segments passing through each node u and determine for every long blue segment b how many segments in the sample it intersects. If this number is $h(b)$, then b intersects between $\sqrt{N}(h(b)-1)/2$ and $\sqrt{N}(h(b)+1)/2$ long red segments at node u. See Fig. 3(c). We choose $c'_b(u)$ to be the maximum of $\sqrt{N}(h(b)+1)/2$ taken over all long blue segments b at node u.

More precisely, we use two passes through the $\sqrt{N}$-merger after allocating a sample buffer $R_s(u)$ of size $2\sqrt{N}$ to each node. The first pass merges red segments by their intersections with left slab boundaries. At a node u, every $\sqrt{N}/2$'th long segment is placed into $R_s(u)$. The second pass merges blue segments by their intersections with left slab boundaries. Before this pass, we set $c'_b(u) = 0$ for every node u. During the merge, when we process a long blue segment b, we determine the number $h_l(b)$ of segments in $R_s(u)$ that intersect $l(\sigma_{l(u)})$ below b, as well as the number $h_r(b)$ of segments in $R_s(u)$ that intersect $r(\sigma_{l(u)})$ below r. Let $h(b) = |h_r(b) - h_l(b)|$. If $\sqrt{N}(h(b)+1)/2 > c'_b(u)$, we set $c'_b(u) = \sqrt{N}(h(b)+1)/2$.

Since we allocate only $O(\sqrt{N})$ space to each merger node during the approximate counting of intersections, the space usage of this step is linear. Moreover, we merge red and blue segments once, and it can be shown that the computation of values $h_r(b)$ and $h_l(b)$ for all blue segments b passing through node u requires two scans of list $R_s(u)$ in total. Hence, this adds $O(N/B)$ to the merge cost, and we obtain the following lemma, which completes the proof of Theorem 2.

Lemma 2. *Long-long intersections can be reported using* $O(\mathrm{Sort}(N) + T_s/B)$ *memory transfers and linear space.*

References

1. A. Aggarwal, J.S. Vitter. The Input/Output complexity of sorting and related problems. *Comm. ACM*, 31(9):1116–1127, 1988.
2. L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, M. G. C. Resende (eds.), *Handbook of Massive Data Sets*. Kluwer Academic Publishers, 2002.
3. L. Arge, G.S. Brodal, R. Fagerberg. Cache-oblivious data structures. In D. Mehta, S. Sahni (eds.), *Handbook on Data Structures and Applications*. CRC Press, 2005.
4. L. Arge, D.E. Vengroff, J.S. Vitter. External-memory algorithms for processing line segments in geographic information systems. *Algorithmica*, 47:1–25, 2007.
5. M.A. Bender, R. Cole, R. Raman. Exponential structures for cache-oblivious algorithms. In *Proc. ICALP*, pp. 195–207, 2002.
6. G.S. Brodal, R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. ICALP*, pp. 426–438, 2002.
7. Y.-J. Chiang, M.T. Goodrich, E.F. Grove, R. Tamassia, D.E. Vengroff, J.S. Vitter. External-memory graph algorithms. In *Proc. SODA*, pp. 139–149, 1995.
8. M. Frigo, C.E. Leiserson, H. Prokop, S. Ramachandran. Cache-oblivious algorithms. In *Proc. FOCS*, pp. 285–298, 1999.
9. M.T. Goodrich, J.-J. Tsay, D.E. Vengroff, J.S. Vitter. External-memory computational geometry. In *Proc. FOCS*, pp. 714–723, 1993.
10. J.S. Vitter. External memory algorithms and data structures: Dealing with MASSIVE data. *ACM Comp. Surveys*, 33(2):209–271, 2001.