I/O-Optimal Distribution Sweeping on Private-Cache Chip Multiprocessors

Deepak Ajwani
Centre for Unified Computing
University College Cork
Cork, Ireland
d.ajwani@cs.ucc.ie

Nodari Sitchinava
MADALGO
Department of Computer Science
University of Aarhus
Aarhus, Denmark
nodari@madalgo.au.dk

Norbert Zeh
Faculty of Computer Science
Dalhousie University
Halifax, Canada
nzeh@cs.dal.ca

***Abstract*—The parallel external memory (PEM) model has been used as a basis for the design and analysis of a wide range of algorithms for private-cache multi-core architectures. As a tool for developing geometric algorithms in this model, a parallel version of the I/O-efficient distribution sweeping framework was introduced recently, and a number of algorithms for problems on axis-aligned objects were obtained using this framework. The obtained algorithms were efficient but not optimal. In this paper, we improve the framework to obtain algorithms with the optimal I/O complexity of $O(\mathrm{sort}_P(N) + K/PB)$ for a number of problems on axis-aligned objects; P denotes the number of cores/processors, B denotes the number of elements that fit in a cache line, N and K denote the sizes of the input and output, respectively, and $\mathrm{sort}_P(N)$ denotes the I/O complexity of sorting N items using P processors in the PEM model.**

To obtain the above improvement, we present a new one-dimensional batched range counting algorithm on a sorted list of ranges and points that achieves an I/O complexity of $O((N + K)/PB)$, where K is the sum of the counts of all the ranges. The key to achieving efficient load balancing among the processors in this algorithm is a new method to count the output without enumerating it, which might be of independent interest.

***Keywords*-parallel external memory, PEM, multicore algorithms, computational geometry, parallel distribution sweeping**

I. INTRODUCTION

Multicore processors are becoming increasingly mainstream. The average desktop computer today contains two to four cores, but Intel announced a 48-core prototype recently [1] and the number is projected to reach hundreds of cores in the near future [2]–[4]. Thus, there is a need for techniques for designing parallel algorithms that can fully utilize these processors [5]. While parallel algorithms have been studied intensively in the past, using very fine-grained models, such as the PRAM model, or rather coarse-grained models, such as the BSP model, none of these approaches seem perfectly suited for multicore processors. The reason is the memory hierarchy of these processors. Even a single core can process data faster than it can be retrieved from main memory. To hide the latency of accessing main memory, modern multicore processors equip each core with a private low-latency cache that can be accessed quickly; this architecture became commonly known as private-cache chip multiprocessor (CMP). To benefit from these caches, algorithms need to be designed so that most of the time they access data in cache, just as in sequential I/O-efficient algorithms. In this paper, we design cache-efficient parallel algorithms for solving a number of fundamental geometric problems on axis-aligned objects on a private-cache CMP. We design our algorithms in the parallel external memory (PEM) model. The remainder of this section reviews this model, discusses previous work, and discusses our new contributions in more detail.

A. Model of Computation and Previous Work

The *parallel external memory* (PEM) model by Arge et al. [6] (see Figure 1) is a parallel extension of the *external memory* model by Aggarwal and Vitter [7]. It consists of P processors, each with a cache of size M. These caches are private to the processors, that is, each processor can access only its own cache. In addition, all processors have access to a shared memory of conceptually unlimited size, which simultaneously serves as a storage area for data that does not fit in cache and as the *only* means of exchanging information between processors. The processors independently manipulate the data in their private caches. In

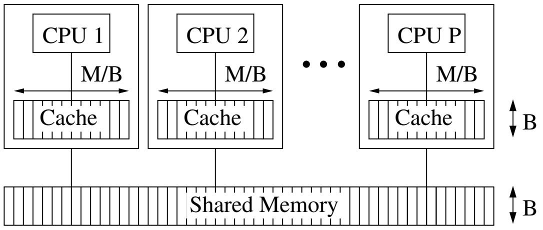

Figure 1. The PEM model

The research was carried out while the first author was working in MADALGO, Center for Massive Data Algorithmics, a center of the Danish National Research Foundation.

The third author was supported in part by the Natural Sciences and Engineering Research Council of Canada and the Canada Research Chairs programme.

order to manipulate a data item currently not in its cache, a processor must first load the data item from shared memory. To communicate with each other, the processors write data from their private caches back to shared memory. This data can then be read by other processors. Formally, data is transferred between the shared memory and the caches by means of *input-output* (I/O) operations. Each such operation transfers one block of B consecutive data elements between each processor's cache and shared memory. Different processors can access different blocks of shared memory in the same I/O operation. Thus, a single I/O operation can transfer up to P blocks between shared memory and the caches, one block per processor. The measure of performance of an algorithm in the PEM model is the number of such (parallel) I/O operations it performs. Thus, when considering only one processor, the PEM model becomes identical to the sequential I/O model. An issue that does not arise in the sequential case is how to resolve conflicts between different processors trying to access the same shared memory block in the same I/O operation. Just as in the PRAM model, we can specify whether to allow or disallow such concurrent read or write I/Os or to allow one but not the other. In this paper, we allow any number of processors to concurrently read the same block but disallow concurrent writes to the same block. This is equivalent to the CREW (concurrent-read-exclusive-write) regime of the PRAM model.

The PEM model is the simplest model of current multicore architectures, focusing on the challenges of combining parallelism with the requirement for spatial locality for efficient use of caches. A number of other, more complicated models of multicore architectures have been proposed in the literature. In [8], Bender et al. studied concurrent searching and updating of cache-oblivious B-trees by multiple processors. In [9]–[14] several different multicore models were considered and cache- and processor-oblivious algorithms were presented for fundamental combinatorial, graph, and matrix-based problems. It remains to be seen which of the different models provides the best trade-off between accurately predicting the real performance of algorithms on modern memory hierarchies and simplicity as an aid for designing algorithms.

In the PEM model, a number of problems have been studied so far. In their paper introducing the model [6], Arge et al. studied a number of fundamental combinatorial problems, such as computing prefix sums and sorting. They showed that the cost of sorting N elements in the PEM model is $\mathrm{sort}_P(N) = \mathrm{O}\big(\frac{N}{PB}\log_{M/B}\frac{N}{B}\big)$ I/Os, provided $P \leq N/B^2$ and $M = B^{O(1)}$. In [15], solutions to a number of fundamental graph problems, such as computing the connected components or a minimum spanning tree, were presented. Most recently, Ajwani et al. [16] presented solutions to a number of geometric problems, including convex hull computation and a number of problems on axis-aligned objects, such as orthogonal line segment intersection.

To obtain solutions to orthogonal line segment intersection and batched orthogonal range reporting, Ajwani et al. introduced a parallel version of the distribution sweeping technique introduced in [17] as a tool for obtaining sequential I/O-efficient solutions to these problems. The main challenge in obtaining optimal solutions to these problems is that the optimal I/O complexity is $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$, where K is the size of the output. The sequential distribution sweeping technique achieves this complexity for the case $P = 1$ by combining the distribution paradigm with the plane sweep paradigm: each recursive call scans the data, and each data access can be charged to an input element or an output element. The same ideas extend to the sequential cache-oblivious model [18], [19]. Existing solutions in the PRAM model [20], [21] achieve the optimal I/O complexity for the case $B = 1$ using completely different techniques that ensure that all processors produce roughly equal portions of the output but rely on very fine-grained access to the shared memory. Achieving the optimal complexity for the case $P > 1$ and $B > 1$ requires novel ideas that combine load balancing with blockwise access to shared memory. Ajwani et al. [16] presented a technique for determining the contribution of each input element to the output *size* efficiently. Using this technique, the data could then be distributed across the processors to achieve good load balancing. They presented two different solutions, both suboptimal. The first one involved a counting step at each level of recursion in the distribution sweep. Since there are $\log_d P$ levels of recursion, where $d := \max(2, \min(\sqrt{N/P}, M/B))$, and the counting step required sorting the input elements, this resulted in an I/O complexity of $\mathrm{O}(\mathrm{sort}_P(N)\log_d P + K/PB)$. The second solution avoided these counting steps by deferring the reporting of intersections to the last level of recursion. This, however, came at the cost of an increase of the input size to the last level of recursion to $\mathrm{O}(N + K)$, which gives an I/O complexity of $\mathrm{O}(\mathrm{sort}_P(N + K))$ I/Os.

B. New Results

In this paper, we show how to implement the counting step at each level of recursion in the distribution sweeping framework of [16] without sorting. This reduces the cost of each level of recursion to $\mathrm{O}((N + K_k)/PB)$ I/Os, where K_k is the output size produced at this level of recursion, and thus leads to solutions to batched range reporting and orthogonal line segment intersection with an overall I/O complexity of $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ I/Os, which is optimal.

The key to achieving this is a new one-dimensional batched range counting algorithm, which takes $\mathrm{O}((N + K)/PB)$ I/Os if the input points and ranges are sorted (which is the case when using this algorithm as part of the distribution sweeping technique), where K is the sum of all the range counts. While the equivalent bound of $\mathrm{O}((N + K)/B)$ I/Os can be achieved trivially in the sequential external memory model by scanning the input and enumerating the whole output, achieving this bound in the PEM model is more challenging. The problem is that $(N + K)/P$, the number of elements a single processor is allowed to inspect in order to achieve the above I/O complexity, may be less than the number of points in a given range. Thus, to achieve the $\mathrm{O}((N+K)/PB)$ I/O complexity, we need to count the output without enumerating it.

Unless stated otherwise, we assume that $P \leq \min\{N/B^2, N/(B \log N)\}$ and $M = B^{\mathrm{O}(1)}$ throughout this paper. The assumptions that $P \leq N/B^2$ and $M = B^{\mathrm{O}(1)}$ are required for optimal sorting in the PEM model [6], while $P \leq N/(B \log N)$ is required by the distribution sweeping framework of [16].

The remainder of this paper is organized as follows. In Section II, we introduce some notation used throughout the paper and discuss some primitives we use repeatedly. In Section III, we present our new 1-d range counting algorithm. In Section IV, we review the distribution sweeping framework of Ajwani et al. [16]. In Sections V and VI, finally, we put everything together to obtain optimal algorithms for orthogonal line segment intersection and related problems. We give some concluding remarks in Section VII.

II. Tools and Notation

In this section, we review a number of primitives we use repeatedly throughout this paper. These primitives were originally discussed in [6] and [16].

A. Prefix Sum and Compaction

Given an array $A[1\,..\,N]$, the *prefix sum* problem is to compute an array $S[1\,..\,N]$ such that $S[i] = \sum_{j=1}^{i} A[j]$. Given a second Boolean array $M[1\,..\,N]$, the *compaction* problem is to arrange all elements $A[i]$ such that $M[i] = \text{true}$ consecutively at the beginning of A without changing their relative order. PEM algorithms for these problems with I/O complexity $\mathrm{O}(N/PB + \log P)$ were presented in [6] (also see [22]).

B. Sorting

Arge et al [6] showed that an array of N elements can be sorted using $\mathrm{sort}_P(N) := \mathrm{O}\big(\frac{N}{PB} \log_{M/B} \frac{N}{B}\big)$ I/Os.

C. Global Load Balancing

Let $A_1, A_2, \ldots, A_r$ be a collection of arrays with $r = \mathrm{O}(P)$ and $\sum_{j=1}^{r} |A_j| = N$, and assume each element x has a positive weight w_x. Let $w_{\max} := \max_x w_x$, $W_j := \sum_{x \in A_j} w_x$ and $W := \sum_{j=1}^{r} W_j$. A *global load balancing* operation assigns *contiguous* subarrays of $A_1, A_2, \ldots, A_r$ to processors so that only a *constant* number of subarrays are assigned to each processor and the total weight of the elements assigned to any processor is $\mathrm{O}(W/P + w_{\max})$. This operation can be implemented by running a constant number of prefix sum and compaction operations and, hence, takes $\mathrm{O}(N/PB + \log P)$ I/Os. The details of the algorithm can be found in [16] and, for the sake of completeness, are also provided in Appendix A.

III. Batched 1-D Range Counting

Given a set $\mathcal{P}$ of points on the real line and a set $\mathcal{Q}$ of intervals over the real line, the *batched 1-d range counting* problem asks to compute the number k_q of points in $\mathcal{P}$ contained in each interval $q \in \mathcal{Q}$. In this section, we show how to solve this problem efficiently, provided the points and intervals are given in sorted order.

Theorem 1. *The batched 1-d range counting problem can be solved using $\mathrm{O}((N + K)/PB + \log P)$ I/Os, where $K = \sum_{q \in \mathcal{Q}} k_q$, provided the input is given as a sorted list of points and interval endpoints and $P \leq \min(N/(B \log N), N/B^2)$. For $P \leq \min(N/(B \log^2 N), N/B^2)$, the I/O complexity of the algorithm becomes $\mathrm{O}((N + K)/PB)$.*

The following corollary is an immediate consequence of Theorem 1, as we can produce the sorted input list required by Theorem 1 by replacing each interval with its two endpoints and sorting the resulting list of points and interval endpoints.

Corollary 1. *The batched 1-d range counting problem can be solved using $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ I/Os, where $K = \sum_{q \in \mathcal{Q}} k_q$, provided $P \leq \min(N/(B \log N), N/B^2)$.*

We use $\mathcal{U}$ to denote the sorted input list of points and interval endpoints. For an interval $q \in \mathcal{Q}$, we use q_l and q_r to denote its left and right endpoints, respectively. Our goal is to annotate every left endpoint q_l in $\mathcal{U}$ with the number k_q of points in $\mathcal{P}$ contained in q. For an interval endpoint q', let $\mathrm{ix}(q')$ denote the number of points in $\mathcal{P}$ that precede q' in $\mathcal{U}$. We call $\mathrm{ix}(q')$ the *index* of q' in $\mathcal{P}$. Since the elements in $\mathcal{U}$ are sorted, it is easy to see that $k_q = \mathrm{ix}(q_r) - \mathrm{ix}(q_l)$. Thus, our task is to compute $\mathrm{ix}(q_r)$ and $\mathrm{ix}(q_l)$, for every interval $q \in Q$, and store their difference with q_l.

A. Computing the Indices of Left Endpoints

The index of *every* interval endpoint can be computed using a prefix sum computation on $\mathcal{U}$. First we assign weight 1 to every point $p \in \mathcal{P}$ and weight 0 to every interval endpoint. The index of an interval endpoint $q' \in \mathcal{U}$ is the prefix sum of q' over these weights. Since a prefix sum computation takes $\mathrm{O}(N/PB + \log P)$ I/Os (see Section II), we can compute the indices of all interval endpoints using $\mathrm{O}(N/PB + \log P)$ I/Os. The batched range counting algorithm of [16] computes the indices of left and right interval endpoints in this manner and then sorts the list of interval endpoints to store the left and right endpoints of each interval consecutively, in order to computer the difference of their indices. Here we use this strategy only to compute the indices of all left endpoints using $\mathrm{O}(N/PB + \log P)$ I/Os.

Next we describe how to annotate every left endpoint q_l with the index $\mathrm{ix}(q_r)$ of its corresponding right endpoint *without* sorting the endpoints.

B. Computing the Indices of Right Endpoints

For the computation of right endpoint indices, we distinguish between *light* and *heavy* intervals in $\mathcal{Q}$. An interval is *light* if it contains less than dB points, and *heavy* otherwise. For the sake of simplicity, we also refer to the endpoints of light or heavy intervals as light or heavy, respectively. The basic idea of our solution is the following. For light intervals, there are not too many points in $\mathcal{P}$ between their left and right endpoints. Thus, after balancing these points between processors, each processor can use the trivial sequential range counting approach to determine the indices of the light right endpoints it is in charge of. In total, the indices of all light endpoints can be computed using $\mathrm{O}(N/PB + \log P)$ I/Os. The number of heavy right endpoints is at most K/dB. This allows us to allocate an equal number of them to each processor; each processor then performs a binary search on $\mathcal{P}$ to determine the index of each heavy right endpoint. By itself, this strategy is too costly, as it would cost $\mathrm{O}((K/dPB)\log N)$ I/Os to determine the indices of all heavy right endpoints. Instead, we first limit the search for the index $\mathrm{ix}(q_r)$ of each heavy right endpoint q_r to the interval between the $\mathrm{ix}(q_l)$th point and the $(\mathrm{ix}(q_l) + d^2B)$th point in $\mathcal{P}$. This strategy finds the indices of most heavy right endpoints and fails for at most K/d^2B heavy endpoints. For these endpoints, we widen the search interval, allowing the search to succeed for all but K/d^3B endpoints. We continue in this manner until all indices have been computed, and we show that the total cost of this parametric search is $\mathrm{O}(K/PB)$ I/Os for all heavy endpoints. By summing the costs of finding the indices of light and heavy right endpoints, we obtain the desired bound of $\mathrm{O}((N+K)/PB + \log P)$ I/Os. Next we discuss the two phases in detail. We start by extracting the sorted lists $\mathcal{P}$ and $\mathcal{Q}_l$ of points and left interval endpoints from $\mathcal{U}$. This takes $\mathrm{O}(N/PB + \log P)$ I/Os by applying two compaction operations to $\mathcal{U}$.

Computing indices of light right endpoints. Let $q_{1,l}, q_{2,l}, \ldots, q_{t,l}$ be the list of left endpoints as they are stored in $\mathcal{Q}_l$. The procedure for computing the indices of light right endpoints consists of two steps. In the first step, we partition $\mathcal{Q}_l$ into P contiguous sublists to be assigned to each processor. In the second step, each processor computes the indices of all light right endpoints corresponding to its assigned left endpoints.

To partition $\mathcal{Q}_l$ into sublists, we assign a weight $w_{q_{j,l}}$ to each such endpoint, which is defined as $w_{q_{j,l}} := \min(dB, \max(1, \mathrm{ix}(q_{j+1,l}) - \mathrm{ix}(q_{j,l})))$, for $1 \le j < t$, and $w_{q_{t,l}} := \min(dB, \max(1, |\mathcal{P}| - \mathrm{ix}(q_{t,l})))$. Then we use the global load balancing operation from Section II to partition $\mathcal{Q}_l$ into sublists according to these weights.

Now let $\mathcal{Q}_{l,i}$ be the list of left endpoints assigned to processor p_i. To determine the indices of all light right endpoints corresponding to left endpoints in $\mathcal{Q}_{l,i}$, processor p_i scans $\mathcal{Q}_{l,i}$. For every left endpoint $q_l \in \mathcal{Q}_{l,i}$, it loads the subarray $\mathcal{P}[\mathrm{ix}(q_l) + 1 \mathinner{..} \mathrm{ix}(q_l) + dB]$ into its cache. Note that this subarray may overlap the subarray of points loaded for q_l's predecessor in $\mathcal{Q}_{l,i}$. Then p_i loads only the portion of q_l's subarray not already in its cache and evicts the portion of the predecessor's subarray not needed by q_l. Since $d \le M/B$, dB points can be held in cache. Now, if q is a light interval, the subarray $\mathcal{P}[\mathrm{ix}(q_l) + 1 \mathinner{..} \mathrm{ix}(q_l) + dB]$ contains all points in q, as well as at least one point not in q. Thus, by inspecting this subarray in its cache, p_i can determine whether q is light or heavy and, in the former case, compute $\mathrm{ix}(q_r)$ and store it with q_l.

Lemma 1. *The indices of light right endpoints can be computed using* $\mathrm{O}(N/PB + \log P)$ *I/Os, provided* $P \le \min(N/(B\log N), N/B^2)$.

Proof: The weights of the left endpoints in $\mathcal{Q}_l$ can be computed by distributing these endpoints evenly over the P processors and having each processor scan its assigned list of points. Thus, this takes $\mathrm{O}(N/PB)$ I/Os. The global load balancing based on these weights then takes $\mathrm{O}(N/PB + \log P)$ I/Os, as discussed in Section II.

Let W be the total weight of all points in $\mathcal{Q}_l$, and let W_i be the weight assigned to processor p_i. Since the maximum weight of each element in $\mathcal{Q}_l$ is dB, the global load balancing operation ensures that $W_i \le W/P + dB$. We prove that $W \le N$ and that each processor performs $\mathrm{O}(W_i/B + d)$ I/Os to compute the indices of its assigned light right endpoints. This shows that the cost of the second step is $\mathrm{O}(N/PB + d) = \mathrm{O}(N/PB)$ I/Os. By adding the costs of the two steps, we obtain the lemma.

To bound $W = \sum_{j=1}^{t} w_{q_{j,l}}$, let $w'_{q_{j,l}} := 1 + \mathrm{ix}(q_{j+1,l}) - \mathrm{ix}(q_{j,l})$, for $1 \le j < t$, and $w'_{q_{t,l}} := 1 + |\mathcal{P}| - \mathrm{ix}(q_{t,l})$. Then $w_{q_{j,l}} \le w'_{q_{j,l}}$, for all $1 \le j \le t$, and hence $W \le W' := \sum_{j=1}^{t} w'_{q_{j,l}}$. Now observe that a point $p \in \mathcal{P}$ contributes to the weight $w'_{q_{j,l}}$ of a point $q_{j,l}$ only if $q_{j,l} \le p \le q_{j+1,l}$ or $j = t$ and $q_{j,l} \le p$. This immediately implies that every point $p \in \mathcal{P}$ contributes to the weight of at most one left interval endpoint and, hence, that $W \le W' \le |\mathcal{Q}| + |\mathcal{P}| \le N$.

To bound the number of I/Os performed by each processor p_i, observe that, for two consecutive points $q_{j,l}$ and $q_{j+1,l}$ in $\mathcal{Q}_{l,i}$, $\mathrm{ix}(q_{j,l}) \le \mathrm{ix}(q_{j+1,l})$. This implies that each processor scans $\mathcal{Q}_{l,i}$ and a portion of $\mathcal{P}$, possibly skipping elements in $\mathcal{P}$ if $\mathrm{ix}(q_{j+1,l}) > \mathrm{ix}(q_{j,l}) + dB$, for some j. The cost of scanning $\mathcal{Q}_{l,i}$ is bounded by $\mathrm{O}(W_i/B)$ because each element of $\mathcal{Q}_{l,i}$ has weight at least 1. Now observe that processor p_i loads dB points from $\mathcal{P}$ into its cache, for the first endpoint in $\mathcal{Q}_{l,i}$. For every subsequent point $q_{j,l} \in \mathcal{Q}_{l,i}$, the number of new points loaded into the cache is bounded by $w_{q_{j,l}}$. Hence, the total number of points processor p_i reads from $\mathcal{P}$ is at most $W_i + dB$. These points are read sequentially, except when skipping over points because $\mathrm{ix}(q_{j+1,l}) > \mathrm{ix}(q_{j,l}) + dB$. The cost of sequentially reading $W_i + dB$ elements is $\mathrm{O}(W_i/B + d)$ I/Os. Skipping over points incurs a random disk access. However, in this case dB new points are read from $\mathcal{P}$, which implies that the cost of these random disk accesses is also bounded by $\mathrm{O}(W_i/B)$ I/Os. ■

Computing the indices of heavy right endpoints. To start the computation of the indices of heavy right endpoints, we apply a compaction operation to $\mathcal{Q}_l$ to obtain the list $\mathcal{Q}_l^1$ of heavy left endpoints. We store with each endpoint $q_l \in \mathcal{Q}_l^1$ its position in $\mathcal{Q}_l$, in order to be able to copy the index of the corresponding right endpoint back to $\mathcal{Q}_l$ once it has been computed. Since, apart from computing the indices of light right endpoints, the previous phase also identified all heavy left endpoints, this compaction operation takes $\mathrm{O}(N/PB + \log P)$ I/Os, as discussed in Section II.

Now the computation of the indices of heavy right endpoints proceeds in iterations. The hth iteration takes a list $\mathcal{Q}_l^h$ of left endpoints as input. This list contains exactly those left endpoints q_l for which $\mathrm{ix}(q_r) - \mathrm{ix}(q_l) \geq d^h B$. Its output is the list of indices of all heavy right endpoints q_r with $\mathrm{ix}(q_r) - \mathrm{ix}(q_l) < d^{h+1}B$, as well as the list $\mathcal{Q}_l^{h+1}$ that forms the input to the next iteration.

In the hth iteration, we distribute the elements of $\mathcal{Q}_l^h$ evenly across the P processors. Let $\mathcal{Q}_{l,i}^h$ be the list allocated to processor p_i. Then processor p_i inspects each left endpoint $q_l \in \mathcal{Q}_{l,i}^h$ in turn and performs a binary search on $\mathcal{P}[\mathrm{ix}(q_l) \mathinner{.\,.} \mathrm{ix}(q_l) + d^{h+1}B]$ to either determine $\mathrm{ix}(q_r)$ or decide that $\mathrm{ix}(q_r) \geq \mathrm{ix}(q_l) + d^{h+1}B$. In the former case, it writes $\mathrm{ix}(q_r)$ to the original position of q_l in $\mathcal{Q}_l$ (which is stored with q_l in$\mathcal{Q}_{l,i}^h$). In the latter case, it marks q_l for processing in the next iteration.

Once all processors have processed their allocated left endpoints in $\mathcal{Q}_l^h$ in this manner, we apply a compaction operation to $\mathcal{Q}_l^h$ to extract the list $\mathcal{Q}_l^{h+1}$ for the next iteration. This iterative procedure stops as soon as $\mathcal{Q}_l^{h+1}$ is empty.

Lemma 2. *The indices of heavy right endpoints can be computed using $\mathrm{O}((N + K)/PB + \log P)$ I/Os, provided $P \leq \min(N/(B \log N), N/B^2)$.*

Proof: The compaction performed in preparation for the first iteration takes $\mathrm{O}(N/PB + \log P)$ I/Os because $\mathcal{Q}_l$ contains at most N elements. Next we bound the cost of the iterations by $\mathrm{O}((N + K)/PB)$, which proves the lemma.

Consider the hth iteration. Each interval q such that $q_l \in \mathcal{Q}_l^h$ contains at least $d^h B$ points. Hence, the size of $\mathcal{Q}_l^h$ is bounded by $K/(d^h B)$, and each processor operates on $K/(d^h PB)$ elements. For each element, the processor performs a binary search of a subarray of $\mathcal{P}$ of size $d^{h+1}B$, which costs $\mathrm{O}(\log(d^{h+1}B) - \log B) = \mathrm{O}((h + 1) \log d)$ I/Os, as the last $\log B$ binary search steps remain within two consecutive blocks of $\mathcal{P}$. Thus, the I/O complexity of all binary searches in the hth iteration is $\mathrm{O}\big(\frac{K(h+1)\log d}{d^h PB}\big)$. The compaction cost in the hth iteration is bounded by $\mathrm{O}(K/(d^h PB) + \log P)$, as discussed in Section II.

Next we observe that the computation of right endpoint indices terminates after at most $\log_d(N/B)$ iterations because, for $h = \log_d(N/B)$, we have $d^h B = N$ and, hence, the hth iteration succeeds in determining the indices of all right endpoints corresponding to left endpoints left in $\mathcal{Q}_l^h$. Thus, omitting big-Oh notation for brevity, the cost of all iterations needed to compute the indices of heavy right endpoints is bounded by

$$\begin{aligned}
\sum_{h=1}^{\log_d(N/B)} & \left(\frac{K(h+1)\log d}{d^h PB} + \log P\right) \\
& \leq \frac{K \log d}{PB} \left(\sum_{h=1}^{\log_d(N/B)} \frac{h+1}{d^h}\right) + \log P \log_d \frac{N}{B} \\
& \leq \frac{K \log d}{PB} \cdot \frac{2d-1}{(d-1)^2} + \log P \log_d \frac{N}{B} \\
& = \frac{K \log d}{PB} \cdot \mathrm{O}\left(\frac{1}{d}\right) + \log P \log_d \frac{N}{B} \\
& = \mathrm{O}\left(\frac{K}{PB} + \log P \log_d \frac{N}{B}\right),
\end{aligned}$$

which is bounded by $\mathrm{O}((N+K)/PB)$, given the constraints on P. ■

Theorem 1 follows from Lemmas 1 and 2.

C. Multiple Instances of Batched 1-d Range Counting

When applying batched 1-d range counting in the context of the distribution sweeping framework, we need to solve several instances of batched 1-d range counting simultaneously at each level of recursion. The following result generalizes Theorem 1 to solving up to P instances simultaneously.

Theorem 2. *Let $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_r$ be sorted lists of points and segment endpoints, each representing the input of a batched 1-d range counting instance, and let $N := \sum_{i=1}^r |\mathcal{U}_i|$. If $r \leq P$ and $P \leq \min(N/(B \log N), N/B^2)$, the r range counting instances represented by $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_r$ can be solved simultaneously using $\mathrm{O}((N + K)/PB + \log P)$ I/Os, where $K := \sum_{i=1}^r \sum_{q \in \mathcal{Q}_i} k_q$ and $\mathcal{Q}_i$ denotes the set of intervals in the range counting instance represented by $\mathcal{U}_i$. For $P \leq \min(N/(B \log^2 N), N/B^2)$, the I/O complexity of the algorithm becomes $\mathrm{O}((N + K)/PB)$.*

Proof: The batched range counting algorithm relies on a prefix sum computation and compaction operations on each list $\mathcal{U}_i$ and on using global load balancing to allocate segments to processors. The prefix sum computation on all lists $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_r$ can be carried out by applying a single segmented prefix sum operation to the concatenation of these lists. (A segmented prefix sum operation does not sum across the boundary between consecutive lists and can

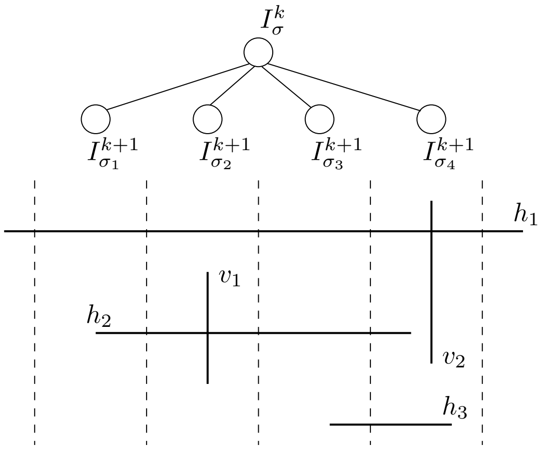

Figure 2. Invocation I_σ^k of the distribution sweeping framework. Given the above segments, the lists generated at invocation I_σ^k look as follows: $Y_{\sigma_1}^{k+1} = \{h_2\}$, $Y_{\sigma_2}^{k+1} = \{v_1\}$, $Y_{\sigma_3}^{k+1} = \{h_3\}$, $Y_{\sigma_4}^{k+1} = \{v_2, h_2, h_3\}$; $R_{\sigma_2}^k = \{v_1, h_2\}$, $R_{\sigma_4}^k = \{v_2\}$. The intermediate sets look as follows: $E_{\sigma_1}^k = \{h_2\}, E_{\sigma_3}^k = \{h_3\}, E_{\sigma_4}^k = \{h_2, h_3\}$; $S_{\sigma_2}^k = \{h_2\}$; $V_{\sigma_2}^k = \{v_1\}$, $V_{\sigma_4}^k = \{v_2\}$. Note that the intersection between h_1 and v_2 is reported at the *parent* invocation of I_σ^k: since h_1 spans the whole slab $\sigma = \cup_{j=1}^4 \sigma_j$, it does not belong to Y_σ^k. Also note that, although h_2 spans σ_3, it does not participate in an intersection in σ_3 and therefore does not appear in $S_{\sigma_3}^k$ nor in $R_{\sigma_3}^k$.

be implemented in the same complexity as a regular prefix sum operation.)

The global load balancing operation also relies only on prefix sum operations, which can be replaced with their segmented version, in order to allocate portions of multiple lists $\mathcal{U}_1, \mathcal{U}_2, \ldots, \mathcal{U}_r$ to the processors. In addition, it requires that we allocate portions of no more than P lists to processors, which is guaranteed by the condition $r \le P$.

In summary, the cost of the batched 1-d range counting procedure is not affected by operating on multiple instances of total size N simultaneously, as long as the total number of instances does not exceed P. ■

IV. Parallel Distribution Sweeping Framework

In this section we review the parallel distribution sweeping framework by Ajwani et al. [16], using orthogonal line segment intersection as illustrating example. Parallel distribution sweeping recursively divides the plane into vertical slabs, starting with the entire plane as one slab and in each recursive step dividing a given slab into $d := \max(2, \min(\sqrt{N/P}, M/B))$ child slabs; refer to Figure 2. This division is chosen so that each slab at a given level of recursion contains roughly the same number of objects (horizontal segment endpoints and vertical segments). The lowest level of recursion divides the plane into P slabs, each containing $\Theta(N/P)$ objects. By viewing the recursion as a rooted tree, we can naturally define *leaf invocations*, *non-leaf invocations*, and *children of non-leaf invocations*. We denote an invocation on a slab σ at the kth level of recursion by I_σ^k. We process all invocations at the same level of recursion in parallel.

Each invocation I_σ^k receives a y-sorted list Y_σ^k as input; this list contains all vertical segment endpoints in σ, as well as all horizontal segments with one endpoint in σ. For the root invocation $I_{\mathbb{R}^2}^0$, $Y_{\mathbb{R}^2}^0$ is a y-sorted list of all vertical segment endpoints and horizontal segments. We generate this list before starting the framework by replacing each vertical segment with its two endpoints and sorting the resulting list of horizontal segments and vertical segment endpoints.

For a non-leaf invocation I_σ^k, let $I_{\sigma_1}^{k+1}, I_{\sigma_2}^{k+1}, \ldots, I_{\sigma_d}^{k+1}$ denote its child invocations. The task of this invocation is to produce the input lists $Y_{\sigma_1}^{k+1}, Y_{\sigma_2}^{k+1}, \ldots, Y_{\sigma_d}^{k+1}$ of its child invocations and to report for each child slab σ_i the intersections between vertical segments in σ_i and horizontal segments completely spanning σ_i. To achieve this, invocation I_σ^k generates a number of lists from Y_σ^k. Let $E_{\sigma_j}^k$ be the y-sorted list of horizontal segments in Y_σ^k with an endpoint in σ_j, $S_{\sigma_j}^k$ the y-sorted list of horizontal segments in Y_σ^k spanning σ_j and with an intersection in σ_j, and $V_{\sigma_j}^k$ the y-sorted list of vertical segment endpoints in Y_σ^k contained in σ_j. For each child slab σ_j, we construct two y-sorted lists $R_{\sigma_j}^k := S_{\sigma_j}^k \cup V_{\sigma_j}^k$ and $Y_{\sigma_j}^{k+1} := E_{\sigma_j}^k \cup V_{\sigma_j}^k$. Then, for each child slab σ_j, we report all intersections between elements of $R_{\sigma_j}^k$ and recursively pass the list $Y_{\sigma_j}^{k+1}$ to the child invocation $I_{\sigma_j}^{k+1}$. The different lists are illustrated in Figure 2.

The task of a leaf invocation I_σ^k is to report all intersections between the elements of Y_σ^k without recursing further. This is done using sequential I/O-efficient techniques after allocating portions of the input lists of all leaf invocations to processors so that each processor is responsible for reporting roughly the same number of intersections.

In [16], Ajwani et al. discussed how to produce the lists $R_{\sigma_j}^k$ and $Y_{\sigma_j}^{k+1}$ at each non-leaf invocation using $\mathrm{O}(\mathrm{sort}_P(N) + K/DB)$ I/Os in total for all invocations. The reason why they did not achieve this I/O complexity for the whole algorithm was the need to balance the load of reporting intersections over all processors at each level of recursion. To achieve this, it was necessary (a) to ensure that no vertical segment participates in more than $\max(N/P, K/(P \log_d P))$ intersections at each level of recursion and (b) to count the number of intersections a vertical segment participates in at each level of recursion.

To ensure that no segment is involved in too many intersections at each level of recursion, Ajwani et al. presented an approach of splitting vertical segments with too many intersections at each level of recursion immediately before reporting intersections at this level. The cost of this splitting step was $\mathrm{O}(\mathrm{sort}_P(N))$ I/Os. Since this splitting step is performed at each of $\log_d P$ levels of recursion, this increased the I/O complexity to $\mathrm{O}(\mathrm{sort}_P(N) \log_d P + K/PB)$.

As we discuss next, counting the number of intersections of a vertical segment at each level of recursion amounts to solving up to P instances of batched range counting. Recall that in invocation I_σ^k, we report intersections among the segments in $R_{\sigma_j}^k = V_{\sigma_j}^k \cup S_{\sigma_j}^k$, for each of d child slabs σ_j of σ. Also recall that the segments in $S_{\sigma_j}^k$ fully span the slab σ_j. Thus, a vertical segment v in $R_{\sigma_j}^k$ and a horizontal segment h in $R_{\sigma_j}^k$ intersect if and only if h's y-coordinate is contained in the y-range of v, and the problem of computing the intersection counts for the vertical segments in $R_{\sigma_j}^k$ reduces to batched one-dimensional range counting on $R_{\sigma_j}^k$, treating vertical segments as intervals over the y-axis and horizontal segments as points on the y-axis. The total number of child slabs at each level of recursion is bounded by the number of leaf invocations, which is P. Thus, if the batched range counting algorithm supports solving up to P instances simultaneously, the batched range counting problems at each level of recursion can be solved in a single invocation of this algorithm. The batched range counting algorithm presented in [16] takes $\mathrm{O}(\mathrm{sort}_P(N))$ I/Os. Since this algorithm is invoked once per level of recursion, this adds another $\mathrm{O}(\mathrm{sort}_P(N) \log_d P)$ I/Os to the total I/O complexity of the algorithm.

Next we discuss how to achieve the optimal I/O complexity of $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ I/Os for the orthogonal line segment intersection problem. We present this solution in two parts. In Section V, we present an algorithm for splitting vertical segments with more than $K' := \max(N/P, K/(P \log_d P))$ intersections into segments with at most K' intersections. This algorithm has I/O complexity $\mathrm{O}(\mathrm{sort}_P(N))$ and produces a set of at most $\mathrm{O}(N)$ segments. In Section VI, we show how to achieve the optimal I/O complexity of $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ I/Os, provided no vertical segment is involved in more than K' intersections. To do this, we implement the batched range counting steps at each level of recursion using our batched range counting algorithm from Section III.

V. Splitting Segments With Many Intersections

Let us call a segment *heavy* if it participates in more than $K' := \max\{N/P, K/(P \log_d P)\}$ intersections, and *light* otherwise. In this section, we discuss how to split the heavy segments in a collection of N horizontal and vertical segments so that the resulting collection of segments contains only light segments and the total number of segments is $\mathrm{O}(N)$. We discuss how to split heavy horizontal segments here. Heavy vertical segments can be split by exchanging the roles of the coordinates.

We start by counting the number of intersections each horizontal segment is involved in. This takes $\mathrm{O}(\mathrm{sort}_P(N))$ I/Os using an algorithm of [16]. Using a compaction operation (see Section II), we extract the list H_l of heavy horizontal segments. Let V be the set of vertical segments.

The central part of our algorithm is to generate a list L_h of x-coordinates where to split each heavy segment $h \in H_l$. Our algorithm generates the elements of all these lists L_h simultaneously, in no particular order. To distinguish which list an x-coordinate x belongs to, we represent an x-coordinate in L_h as the pair (h, x). To generate the light subsegments of the segments in H_l, we sort the list L of all such pairs we generate primarily by the lists L_h they belong to and secondarily by their x-coordinates. Given this sorted list, the light subsegments of all heavy horizontal segments can then be generated using a single parallel scan. We argue below that the total number of pairs (h, x) generated for all heavy horizontal segments is $\mathrm{O}(N)$. Thus, once these pairs have been generated, the sorting and scanning steps needed to generate the subsegments take $\mathrm{O}(\mathrm{sort}_P(N))$ I/Os.

To split a segment $h \in H_l$, we intuitively add x-coordinates to L_h such that there are K' intersection points between each pair of consecutive x-coordinates. Doing this precisely is difficult. Instead, we allow these splitting coordinates to deviate by up to N/P intersection points from their ideal positions. This may increase the number of intersections per generated subsegment to $K' + N/P = \mathrm{O}(K')$, which is sufficient for our purposes.

The high-level procedure is as follows: We use the distribution sweeping framework, starting with a y-sorted list $Y_{\mathbb{R}^2}^0$ of vertical segment endpoints and horizontal segments as the input to the root invocation $I_{\mathbb{R}^2}^0$. For an invocation I_σ^k with input list Y_σ^k, we send all the vertical segment endpoints in Y_σ^k to the appropriate child lists $Y_{\sigma_j}^{k+1}$. We add a horizontal segment to $Y_{\sigma_j}^{k+1}$ if it has an endpoint in σ_j or it should be split at an x-coordinate inside σ_j. We determine the positions where to split a horizontal segment h as follows: Consider the first invocation I_σ^k such that h is completely contained in σ but spans at least one slab boundary between child slabs of σ. We define the leftmost such intersection between h and a slab boundary to be the *anchor* of h; see Figure 3. We split h at its anchor. The remaining split coordinates of h are chosen by "walking" left and right from the anchor and placing a split point each time we pass K' intersections points. These split coordinates are *ideal* in the sense that they would guarantee exactly K' intersection points per generated subsegment of h, except possibly for the two end pieces including the endpoints of h, which may have fewer than K' intersections. When this recursive process reaches the leaf level of recursion, the input list of a leaf invocation I_σ^k contains exactly those horizontal segments that have an endpoint in σ or should be split at an x-coordinate inside σ. Note that no segment in H_l can be completely contained in such a leaf slab σ because σ contains only N/P vertical segments and, hence, a segment contained in σ can have only $N/P \leq K'$ intersections. Thus, every segment in Y_σ^k intersects at least one of the boundaries of σ. We split all segments intersecting the left boundary of σ at this left boundary. (We do not need to split segments at the right boundary because this is the left boundary for the next slab to the right, and the segment will be split in the corresponding invocation.) Note that, by doing so, we do not necessarily split segments at their ideal split coordinates. However, since σ contains only N/P vertical segments, moving the ideal split point of a segment $h \in Y_\sigma^k$ to the left boundary of σ adds at most N/P intersection points to the subsegment of h to the right of this split point. Thus, each of the subsegments generated using these split coordinates intersects at most $K' + N/P$ vertical segments, as desired.

It remains to discuss how to decide for a non-leaf invocation I_σ^k whether a horizontal segment $h \in Y_\sigma^k$ has an ideal split point inside a child slab σ_j of σ, in order to add h to the input list $Y_{\sigma_j}^{k+1}$ of the corresponding child invocation $I_{\sigma_j}^{k+1}$. As long as a horizontal segment h has not reached an invocation yet where it intersects a slab boundary, nothing special needs to be done for this segment. Once we have fixed the anchor of h, we operate on the two subsegments h_l and h_r left and right of the anchor independently. Here we discuss how to handle the right subsegment h_r; the left subsegment can be handled symmetrically. As h_r moves towards the leaf invocation corresponding to the leaf slab containing its right endpoint, we maintain a weight $w(h_r)$ of h_r. When h_r is at an invocation I_σ^k, the weight of h_r is the number of intersections h_r has between its left endpoint (the anchor of h) and the left boundary of σ. For the invocation I_σ^k that determines the anchor of h, h_r is completely contained in σ. Hence, we initialize the weight of h_r to 0. Then, for any invocation I_σ^k that processes h_r, we consider all child slabs $\sigma_i, \sigma_{i+1}, \ldots, \sigma_j$ of σ which h_r spans completely. Let $\beta_i, \beta_{i+1}, \ldots, \beta_{j+1}$ be the slab boundaries defining these slabs, and let k_q be the number of intersections h_r has inside the slab σ_q, for $i \le q \le j$. We define the *rank* of a slab boundary β_q, for $i \le q \le j+1$, as $\mathrm{rank}_{h_r}(\beta_q) := w(h_r) + \sum_{q'=i}^{q-1} k_{q'}$. It is not hard to see that h_r has an ideal split coordinate in a child slab σ_q if $\lfloor \mathrm{rank}_{h_r}(\beta_q)/K' \rfloor < \lfloor \mathrm{rank}_{h_r}(\beta_{q+1})/K' \rfloor$. In this case, we add h_r to the list $Y_{\sigma_q}^{k+1}$. When adding h_r to such a list $Y_{\sigma_q}^{k+1}$, we set its weight in $Y_{\sigma_q}^{k+1}$ to $\mathrm{rank}_{h_r}(\beta_q)$.

The implementation of this procedure requires counting for every horizontal segment the number of intersections it has in each child slab it completely spans. In [16] it is shown how do this using $O(N_k/PB)$ I/Os per level of recursion, where N_k is the input size to all invocations at level k. Next we show that $N_k = O(N)$, for all k, which implies that the cost of this counting step and the cost of distributing segments to the child lists at each level of recursion is $O(N/PB)$ I/Os. By summing over all levels of recursion, we obtain that the total cost of generating the input lists for the leaf invocations is $O((N/PB)\log_d P) = O(\mathrm{sort}_P(N))$ I/Os. We have already argued that the cost of generating the light subsegments from the input lists of the leaf invocations takes $O(\mathrm{sort}_P(N))$ I/Os. Hence, the total I/O complexity of this procedure is $O(\mathrm{sort}_P(N))$, and the total number of segments we generate is bounded by the total input size of all leaf invocations, which is $O(N)$.

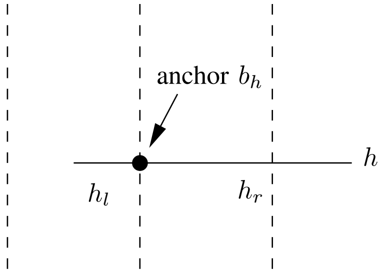

Figure 3. An example of anchoring segment h at b_h.

The bound on N_k follows if we can bound the total number of ideal split coordinates by $O(N)$ because a horizontal segment belongs to the input of an invocation I_σ^k only if it has an endpoint or an ideal split coordinate inside σ. However, since every two consecutive ideal split coordinates of a segment have exactly K' intersections between them, the total number of ideal split coordinates is at most $K/K' \le P \log_d P \le N$, where the latter inequality follows from the bound $P \le N/(B \log N)$.

Theorem 3. *A list of N horizontal and vertical segments can be replaced with a list of $O(N)$ horizontal and vertical segments with the same intersection points and such that each segment has at most $K' := \max(N/P, K/(P \log_d P))$ intersections. If $P \le \min(N/B^2, N/B \log N)$, the I/O complexity of the algorithm to do this is $O(\mathrm{sort}_P(N))$.*

VI. Optimal Orthogonal Line Segment Intersection Reporting

As discussed in Section IV, the orthogonal line segment intersection algorithm of [16] achieves the optimal I/O complexity of $O(\mathrm{sort}_P(N) + K/PB)$ I/Os, excluding the cost of splitting heavy segments and the cost of counting the number of intersections each vertical segment is involved in at each level of recursion.

In Section V, we have shown how to split heavy segments into light ones using $O(\mathrm{sort}_P(N))$ I/Os and without increasing the number of segments by more than a constant factor. Thus, to achieve the optimal I/O complexity, it suffices to show that we can count the number of intersections each vertical segment is involved in at each level of recursion using a total of at most $O(\mathrm{sort}_P(N) + K/PB)$ I/Os for all levels.

Let K_k be the total number of intersections to be reported at all invocations at the kth level of recursion, and let N_k be the total size of the lists $R_{\sigma_j}^k$ constructed at this level in order to report these intersections. Since a horizontal segment belongs to such a list $R^k_{\sigma_j}$ if and only if it spans σ_j and has an intersection with a vertical segment in σ_j, it follows that $N_k \leq N + K_k$. Also, since all vertical and horizontal segments appear at each level of recursion in the slab containing their endpoints, $N_k \geq N$. As argued in Section IV, counting the number of intersections each vertical segment is involved in at this level of recursion amounts to batched 1-d range counting on all the lists $R^k_{\sigma_j}$ at this level. Since there are at most P such lists $R^k_{\sigma_j}$ at any level of recursion, Theorem 2 shows that we can solve all these batched 1-d range counting problems simultaneously using $\mathrm{O}((N_k + K_k)/PB) = \mathrm{O}((N + K_k)/PB)$ I/Os, provided $P \leq \min(N/(B\log^2 N), N/B^2)$. Since $\sum_k K_k \leq K$ and there are $\log_d P$ levels of recursion, the total cost of all these batched range counting steps is therefore $\mathrm{O}((N\log_d P + K)/PB) = \mathrm{O}(\mathrm{sort}_P(N) + K/PB)$. This proves the following theorem.

Theorem 4. *Orthogonal line segment intersection reporting can be solved using* $\mathrm{O}(\mathrm{sort}_P(N)+K/PB)$ *I/Os and* $\mathrm{O}(N+K)$ *space, provided* $P \leq \min(N/(B\log^2 N), N/B^2)$.

Note that the permissible number of processors is no more than $N/(B\log^2 N)$, while the suboptimal solution in [16] worked for up to $\min(N/(B\log N), N/B^2)$ processors. We can achieve the optimal I/O complexity in Theorem 4 also for up to this number of processors, at the expense of using more space.

Theorem 5. *Orthogonal line segment intersection reporting can be solved using* $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ *I/Os and* $\mathrm{O}(N\log_d P + K)$ *space, where* $d := \max(2, \min(\sqrt{N/P}, M/B))$, *provided* $P \leq \min(N/(B\log N), N/B^2)$.

Proof: The only part of our orthogonal line segment intersection algorithm that relies on the assumption that $P \leq N/(B\log^2 N)$ is the batched range counting algorithm. For $N/(B\log^2 N) < P \leq \min(N/(B\log N), N/B^2)$, the cost of batched range counting becomes $\mathrm{O}((N+K)/PB + \log P)$ I/Os, which would increase the total cost of the batched range counting steps over all levels of recursion to $\mathrm{O}(\mathrm{sort}_P(N) + K/PB + \log P\log_d P)$.

To avoid this blow-up of the range counting cost, we first generate the lists $R^k_{\sigma_j}$ for *all* levels of recursion and then run *one* batched range counting step on all these lists, followed by reporting the intersections in all these lists. This is possible because the total number of these lists over all levels of recursion is still $\mathrm{O}(P)$. As we have argued above, the size of the lists $R^k_{\sigma_j}$ at one level of recursion is $\mathrm{O}(N + K_k)$. By summing this over all levels, we obtain a total size of these lists of $\mathrm{O}(N\log_d P + K)$, which gives the increased space bound in the theorem. The cost of running batched range counting on these lists is $\mathrm{O}((N\log_d P + K)/PB + \log P) = \mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ because $P \leq N/(B\log N)$ implies that $\log P \leq N/PB$. This gives the I/O complexity claimed in the theorem. ■

VII. Discussion

This paper improves the parallel distribution sweeping framework of [16] to obtain an optimal $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ I/Os orthogonal line segment intersection reporting algorithm. In order to achieve this, we had to address two challenges:

- Ensure that no vertical segment participates in more than $K' = \max\{N/P, K/(P\log_d P)\}$ intersections at each level of recursion.
- Count the number of intersections a vertical segment participates in at each level of recursion within the optimal I/O bound.

We achieved the first goal by splitting the segments in a preprocessing step. To attain the second goal, we used our $\mathrm{O}((N+K)/PB)$ I/O 1-d batched range counting algorithm.

It remains open whether similar results can be obtained on hardware-oblivious models of private-cache chip multiprocessors. It would be particularly interesting to see if an I/O-optimal low-depth cache-oblivious distribution sweeping paradigm can be designed, along the lines of [14].

References

[1] Intel Corp., "Futuristic Intel chip could reshape how computers are built, consumers interact with their PCs and personal devices," Press Release: http://www.intel.com/pressroom/archive/releases/2009/20091202comp_sm.htm, Dec. 2009.

[2] D. Geer, "Chip makers turn to multicore processors," *IEEE Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[3] G. Lowney, "Why Intel is designing multi-core processors," available at https://conferences.umiacs.umd.edu/paa/lowney.pdf.

[4] J. Rattner, "Multi-core to the masses," *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pp. 3–3, 2005.

[5] P. Gibbons, "Theory: Asleep at the switch to many-core," Workshop on Theory and Many-Cores (T&MC), May 2009.

[6] L. Arge, M. T. Goodrich, M. J. Nelson, and N. Sitchinava, "Fundamental parallel algorithms for private-cache chip multiprocessors," in *SPAA*, 2008, pp. 197–206.

[7] A. Aggarwal and J. S. Vitter, "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.

[8] M. A. Bender, J. T. Fineman, S. Gilbert, and B. C. Kuszmaul, "Concurrent cache-oblivious B-trees," in *SPAA*, 2005, pp. 228–237.

[9] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, "Provably good multicore cache performance for divide-and-conquer algorithms," in *SODA*, 2008, pp. 501–510.

[10] R. A. Chowdhury and V. Ramachandran, "Cache-efficient dynamic programming for multicores," in *SPAA*, 2008, pp. 207–216.

[11] ——, "The cache-oblivious Gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation," in *SPAA*, 2007, pp. 71–80.

[12] R. Cole and V. Ramachandran, "Resource-oblivious sorting on multicores," in *ICALP*, 2010, pp. 226–237.

[13] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran, "Oblivious algorithms for multicores and network of processors," in *IPDPS*, 2010.

[14] G. Blelloch, P. Gibbons, and H. Simhadri, "Low depth cache-oblivious algorithms," in *SPAA*, 2010, pp. 189–199.

[15] L. Arge, M. T. Goodrich, and N. Sitchinava, "Parallel external memory graph algorithms," in *IPDPS*, 2010.

[16] D. Ajwani, N. Sitchinava, and N. Zeh, "Geometric algorithms for private-cache chip multiprocessors," in *ESA*, 2010.

[17] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter, "External-memory computational geometry," in *FOCS*, 1993, pp. 714–723.

[18] L. Arge, T. Mølhave, and N. Zeh, "Cache-oblivious red-blue line segment intersection," in *ESA*, 2008, pp. 88–99.

[19] G. S. Brodal and R. Fagerberg, "Cache oblivious distribution sweeping," in *ICALP*, ser. Lecture Notes in Computer Science, vol. 2380. Springer-Verlag, 2002, pp. 426–438.

[20] A. Datta, "Efficient parallel algorithms for geometric partitioning problems through parallel range searching," in *ICPP*, 1994, pp. 202–209.

[21] M. T. Goodrich, "Intersecting line segments in parallel with an output-sensitive number of processors," *SIAM J. Comp.*, vol. 20, no. 4, pp. 737–755, 1991.

[22] N. Sitchinava, "Parallel external memory model – a parallel model for multi-core architectures," Ph.D. dissertation, University of California, Irvine, 2009.

Appendix A.
Global Load Balancing

Let $A_1, A_2, \dots, A_r$ be arrays each of whose elements e has a positive weight w_e. Assume further that $r \le P$ and $\sum_{i=1}^{r} |A_i| = N$, and let $W_i := \sum_{e \in A_i} w_e$ be the total weight of the elements in array A_i, $W := \sum_{i=1}^{r} W_i$, and $w_{\text{max}} := \max_{1 \le i \le r} \max_{e \in A_i} w_e$. The *global load balancing* problem is to assign contiguous *chunks* of arrays $A_1, A_2, \dots, A_r$ to processors so that each processor receives $\mathrm{O}(1)$ chunks and the total weight of the elements assigned to each processor is $\mathrm{O}(W/P + w_{\text{max}})$. In Section II, we claimed that this operation can be implemented using $\mathrm{O}(N/PB + \log P)$ I/Os and gave a sketch of the algorithm. Here we provide the details.

Without loss of generality, we assume that every array A_i is aligned at a block boundary and its size is a multiple of B. If that is not the case, we can pad each array with dummy entries of weight 0 at the end and remove the padding after the completion of the load balancing procedure. Note that the padding does not asymptotically increase the total size of the arrays because the padding is at most $B - 1$ elements for each array, $r(B-1) \le P(B-1) \le N$ elements in total because $P \le N/B$.

First we apply a prefix sum operation to the weights of the elements in each array A_i. This can be implemented using a single "segmented" prefix sum operation applied to the concatenation A of arrays $A_1, A_2, \dots, A_r$, which does not sum across the boundary of two consecutive arrays A_i and A_{i+1}. Thus, this step takes $\mathrm{O}(N/PB + \log P)$ I/Os. Next we divide A into P chunks of size $\lceil N/P \rceil$ and assign one chunk to each processor. This can be done using simple index arithmetic on A. Each processor inspects every element e in its assigned chunk and marks it if either e is the first element of an array A_i or the prefix sums W_e and $W_{e'}$ of e and its predecessor e' in A_i satisfy $\lfloor PW_{e'}/W \rfloor < \lfloor PW_e/W \rfloor$. Next we apply a compaction operation to A to obtain the list of marked elements, each annotated with the array A_i it belongs to and its position in A_i. These marked elements are the start elements of the chunks we wanted to construct, and we assign two consecutive chunks to each processor. The I/O complexity of this procedure is easily seen to be $\mathrm{O}(N/PB + \log P)$, as it involves a prefix sum and a compaction operation, plus sequential processing of $\lceil N/PB \rceil$ blocks per processor and one access to two consecutive elements per processor in the array of marked elements. The constructed chunks have the desired properties:

- Since the first element of every array A_i is marked, every chunk contains elements from exactly one array A_i.
- The number of chunks is at most $2P$, that is, by assigning two chunks to each processor, we do assign all chunks to processors. To see this, observe that the number of marked elements per array A_i is at most $1 + \lfloor W_i P/W \rfloor$, which implies that the total number of marked elements, that is, the total number of chunks is at most $r + P \le 2P$.
- Every chunk has total weight at most $W/P + w_{\text{max}}$. To see this, consider a chunk with first element e and last element e', and let W_e and $W_{e'}$ denote their prefix sums. Then $\lfloor PW_e/W \rfloor = \lfloor PW_{e'}/W \rfloor$, that is, the total weight of the elements in the chunk, excluding e, is at most W/P. Since e has weight at most w_{max}, the total weight of the chunk is at most $W/P + w_{\text{max}}$.