I/O-optimal Algorithms for Orthogonal Problems for Private-Cache Chip Multiprocessors

Deepak Ajwani
MADALGO
Department of Computer Science
University of Aarhus
Aarhus, Denmark
ajwani@madalgo.au.dk

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 the private-cache multi-core architectures. Recently a parallel version of the distribution sweeping framework was introduced to efficiently solve a number of orthogonal geometric problems in the PEM model. In this paper we improve the framework to the optimal $O(sort_P(N) + K/PB)$ I/Os, where P is the number of cores/processors, B is the number of elements that fit into a cache-line, N and K are the sizes of the input and output, respectively, and $sort_P(N)$ denotes the I/O complexity of sorting N items on a P-processor PEM model.**

We achieve this with a new one-dimensional batched range counting algorithm on a sorted list of ranges and points that achieves $O((N + K)/PB)$ I/O complexity, where K is the sum of counts of all the ranges. The key to achieving efficient load balancing among the processors for this problem 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

Multicores are becoming a norm among the commodity hardware. The computers of an average user today contain two to four cores. But recently Intel announced a 48-core prototype [1] and the number is projected to reach hundreds of cores in the near future [2]–[4]. Thus, there is a need for algorithmic techniques to fully take advantage of the parallelism associated with such a large number of processing cores [5].

To hide the latency of accessing main memory modern multicores implement low-latency caches which are private for each processor. This architecture became commonly know as private-cache chip multiprocessor (CMP). To fully take advantage of such architecture, there is a need for algorithms that minimize access to shared memory and consequently, optimize the usage of private caches.

MADALGO is the 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.

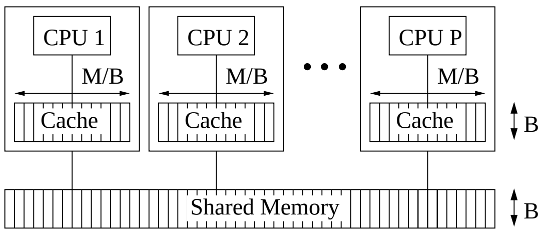

Figure 1. The PEM model

In this paper we improve the results of a recent paper [6] and provide techniques to achieve optimal cache utilization for output-sensitive geometric problems in the parallel external memory model – a private-cache CMP model.

A. Model of Computation and Previous Work

In this paper we study geometric algorithms in the *parallel external memory* (PEM) model of Arge et al. [7]. The model is a parallel extension of the *external memory* model of Aggarwal and Vitter [8] (see Figure 1). It consists of P processors, each with a private cache of size M for each processor. To perform any operation on data, a processor must first load the data into its cache. The caches are private to the processors, that is, each processor can access only its own cache. The inter-processor communication is conducted via processors writing to and reading from a *shared memory*. The data is transferred between the memory and caches in blocks of B elements. During one *input-output* (I/O) operation, each processor can transfer a single such block of elements between shared memory and its cache. Thus, during a single parallel I/O operation up to P blocks can be transferred between the shared memory and the P caches. One of the metrics of the PEM model is *parallel I/O complexity*, which counts the number of parallel I/O operations performed during execution of an algorithm. Just as in the PRAM model, different assumptions can be made when multiple processors attempt to read or write from the same block. In this paper we allow any number of processors concurrently reading the same block, but disallow concurrent writes.

The PEM model is the simplest model of current multicore architecutures, focusing on the challenges of combining parallelism with the requirement for spatial locality for efficient use of caches. A number of problems have been studied in the PEM model. Arge et al. [7] study a number of fundamental combinatorial problems such as prefix sum operation and sorting. In another paper [9], solutions to fundamental problems on graphs are presented.

A number of other results have been obtained in the more complicated cache-oblivious and resource obliviuos multicore models. In [10], Bender et al. study concurrent searching and updating of cache-oblivious B-trees by multiple processors. In [11]–[16] several different multicore models are considered and cache- and processor-oblivious algorithms are presented for fundamental combinatorial, graph, and matrix-based problems.

Most recently, Ajwani et al. [6] study the distribution sweeping technique [17] in the PEM model. They present a parallel version of the technique that has been very successful at solving geometric problems I/O-efficiently in the sequential external memory model [17]. Using the parallel distribution sweeping the authors show how to solve a number of two-dimensional orthogonal geometric problems, such as orthogonal line segment intersection reporting, orthogonal range reporting and reporting all pairs of overlapping rectangles in the PEM model. What distinguishes their paper from other results in multicore models is that their solutions to the problems are output-sensitive, which means that the running time and the I/O complexity depend on the size of the output reported. The difficulty with the output sensitivity in the parallel setting lies in partitioning the input among the processors so that each processor reads roughly equal fractions of the input, while producing equal fractions of the output. The authors present techniques to compute the *size* of the output and balance the work based on this information. However, this step for computing the size of the output seems to be the bottleneck for achieving the optimal I/O complexity for the geometric problems they considered. They describe two different solutions, one that is I/O-optimal in the size of the input and the other that is optimal in the size of the output, but neither achieves optimality in both parameters.

While output-sensitive algorithms haven't been studied before [6], there is related work in the sequential external memory [17] and cache-oblivous [18], [19] models, as well as in the PRAM model [20], [21]. However, the sequential external memory and cache-oblivious solutions lack the requirement to load balance the work among processors, while the PRAM solutions rely on the fine-grained access to the shared memory.

B. New Results

In this paper we improve the results of Ajwani et al. [6] and present a solution to the line segment intersection reporting problem that exhibits $\mathrm{O}(\mathrm{sort}_P(N)+K/PB)$ parallel I/O complexity, which is asymptotically optimal in the sizes of both the input and the output. Our main results are stated in Theorems 3 and 4.

To achieve this bound we develop a new technique to solve the one-dimensional batched range counting problem on a sorted input of ranges and points, which might be of independent interest. Our algorithm (presented in Section IV) exhibits $\mathrm{O}((N+K)/PB)$ I/O complexity, where K is the sum of all the range counts. This is an improvement over previous best known result of $\mathrm{O}(\mathrm{sort}_P(N)+K/PB)$. While the equivalent bound of $\mathrm{O}((N+K)/B)$ in the sequential external memory model is not a very exciting result for a counting problem because the count can be performed by enumerating the whole output, the uniqueness of our result in the PEM model lies in the fact that K/P, the size of the output that each processor may access, might be smaller than the number of points falling within some range. Thus, to achieve the $\mathrm{O}((N+K)/PB)$ I/O complexity, the processor must be able to count without enumerating the whole output.

Throughout this paper, we will assume that the geometric objects contain only a constant number of endpoints and each endpoint stores information about the other endpoints in one of its fields. We will also assume, unless stated otherwise, that $P \le \min\{N/B^2, N/(B\log N)\}$ and $M = B^{\mathrm{O}(1)}$. The assumptions $P \le N/B^2$ and $M = B^{\mathrm{O}(1)}$ were stated in [7] and are required to prove the sorting bounds, while $P \le N/(B\log N)$ is used in [6] to prove bounds for the distribution sweeping algorithms.

II. Tools and Notations

In this section, we review the primitives that we use repeatedly throughout this paper. These primitives have been initially described in [7] and [6] or are simple extensions thereof.

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 $M[i]$ = true consecutively at the beginning of A without changing their relative order. PEM algorithms for these problems with I/O complexity $\mathrm{O}(N/PB + \log P) = \mathrm{O}(N/PB)$ are presented in [7] (also see [22]).

Multi-way distribution. Given an array of N elements with each element annotated with one of d properties, the *d-way distribution problem* asks to generate d contiguous arrays, each consisting of elements containing the same property. A distribution is *stable* if the relative order of elements in the resulting arrays is the same as in the input array. The stable d-way distribution problem can be solved in the PEM model in $\mathrm{O}(N/PB)$ I/Os for any constant d.

The solution is a simple application of parallel scan and compaction: each processor scans an equal fraction of the input and distributes its elements among d different lists. Finally, for each of d properties, the processors run a compaction operation on the generated arrays of that property. Since d is a constant, the compaction operations run in $\mathrm{O}(N/PB + \log P) = \mathrm{O}(N/PB)$ I/Os.

Sorting. Arge et al [7] show that an array of N elements can be sorted in the PEM model in $\mathrm{O}(\frac{N}{PB} \log_{M/B} \frac{N}{B})$ I/Os. We will use $\mathrm{sort}_P(N)$ to denote the I/O complexity of sorting N elements.

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 $\mathrm{O}(N/PB)$ I/Os. The details of the algorithm can be found in [6] but for the sake of completeness we reproduce them in Appendix A.

III. Parallel Distribution Sweeping Framework

In this section we review the parallel distribution sweeping framework of Ajwani et al. [6].

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 (e.g., horizontal segment endpoints and vertical segments). The lowest level of recursion divides the plane into P slabs, each containing $\Theta(N/P)$ input elements. Viewing the recursion as a rooted tree defines leaf invocations and children of a non-leaf invocation. An invocation on slab σ at the k^{th} recursive level is denoted as I_σ^k. All invocations at the same recursive level are processed in parallel.

Each invocation I_σ^k receives as input a y-sorted list Y_σ^k containing horizontal segments and vertical segment endpoints. The root invocation $I_{\mathbb{R}^2}^0$ contains all horizontal segments and vertical segment endpoints of the input and $Y_{\mathbb{R}^2}^0$ is generated by sorting the input by the y-coordinate. 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. In processing I_σ^k, we report all intersections between the vertical segments in σ and the portions of horizontal segments spanning some child slab

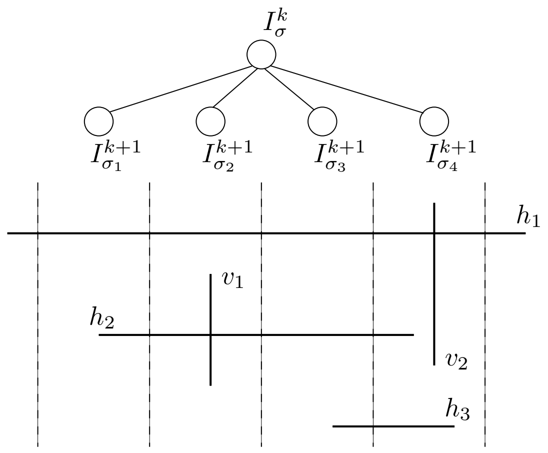

Figure 2. Invocation I_σ^k of the distribution sweeping framework. Given the above segments, the lists generated at invocation I_σ^k look as follows: $Y_{\sigma_1}^{k+1} = \{h_2\}$, $Y_{\sigma_2}^{k+1} = \{v_1\}$, $Y_{\sigma_3}^{k+1} = \{h_3\}$, $Y_{\sigma_4}^{k+1} = \{v_2, h_2, h_3\}$; $R_{\sigma_2}^k = \{v_1, h_2\}$, $R_{\sigma_4}^k = \{v_2\}$. 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 belongs to Y_σ^{k-1} and does not belong to Y_σ^k, therefore, is not an input to invocation I_σ^k. Also note that although h_2 spans σ_3, it does not participate in an intersection in σ_3 and, therefore, it does not appear in $S_{\sigma_3}^k$ and, consequently, in $R_{\sigma_3}^k$.

σ_j (for some $1 \le j \le d$). Formally, let $E_{\sigma_j}^k$ denote 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 the solution constructs a y-sorted list $R_{\sigma_j}^k := S_{\sigma_j}^k \cup V_{\sigma_j}^k$, reports all intersections between segments in $R_{\sigma_j}^k$, and then recurses on each child invocation $I_{\sigma_j}^{k+1}$ with input $Y_{\sigma_j}^{k+1} := E_{\sigma_j}^k \cup V_{\sigma_j}^k$. An example of the lists generated at invocation I_σ^k is illustrated in Figure 2.

At every leaf invocation I_σ^k all intersections between the elements in Y_σ^k are found using sequential I/O-efficient techniques, while carefully balancing the work among processors. (For details, see [6].)

The suboptimality of the solution of Ajwani et al. [6] arises from two challenges, which in turn stem from the requirement to balance the load of reporting the intersections among the processors.

The first challenge arises from the fact that the reporting phase of the line segment intersection reporting algorithm in [6] requires that no vertical segment participates in more than $K' := \max\{N/B^2, K/(P \log_d P)\}$ intersections. Call a segment *heavy* if it participates in more than K' intersections, and *light* otherwise. The authors address this requirement by splitting each heavy segment into a set of light ones right before reporting intersections at each recursive level. Their solution requires $\mathrm{O}(\mathrm{sort}_P(N))$ I/Os at each recursive level, which over all recursive levels adds up to the suboptimal $\mathrm{O}(\mathrm{sort}_P(N) \log_d P)$ I/O complexity. In

Section VI we present a different approach to transforming heavy segments into the set of light ones. In particular, we divide heavy vertical segments into light ones as a preprocessing step to the whole intersection reporting algorithm. This preprocessing step takes $\mathrm{O}(\mathrm{sort}_P(N))$ I/Os which is within the bounds of an optimal solution for the line segment intersection reporting problem.

The second challenge arises from the requirement to balance the work of reporting intersections at each recursive level k, which in turn requires the knowledge of $K_k(v)$, the number of intersections that each vertical segment $v \in V^k_{\sigma_j}$ participates at that recursive level. Computing $K_k(v)$ for each recursive level is the other source of suboptimality in [6].

Recall that in invocation I^k_σ, the algorithm reports intersections among the segments in $R^k_{\sigma_j} = V^k_{\sigma_j} \cup S^k_{\sigma_j}$ for each of d child slabs σ_j. Also recall that the segments of $S^k_{\sigma_j}$ fully span the slab σ_j. Thus, if the y-coordinate of a horizontal segment $h \in S^k_{\sigma_j}$ falls within the range of the y-coordinates of the two endpoints of some vertical segment $v \in V^k_{\sigma_j}$, then v and h must intersect. Thus, the problem of computing the values $K_k(v)$ reduces to the batched one-dimensional range counting problem on the sorted lists $R^k_{\sigma_j} = S^k_{\sigma_j} \cup V^k_{\sigma_j}$ treating vertical segments of $V^k_{\sigma_j}$ as intervals and the y-coordinates of the horizontal segments of $S^k_{\sigma_j}$ as points.

Ajwani et al [6] present a solution to the range counting problem that exhibits $\mathrm{O}(\mathrm{sort}_P(N))$ I/O complexity. Using this counting solution at each level of recursion of the parallel distribution sweeping framework results in a suboptimal I/O complexity of $\mathrm{O}(\mathrm{sort}_P(N)\log_d P)$ for all counting steps. In Section IV we present an improved algorithm to solve the range counting problem. Our algorithm uses $\mathrm{O}((N+K)/PB)$ I/Os, where N is the size of the input to the range counting algorithm and K is the combined sum of the range counts. We use this algorithm for counting intersections $K_k(v)$ to achieve the optimal solution.

In the next section we present solution to the batched 1-D range counting problem. Next, in Section V we present our orthogonal line segment intersection reporting algorithm assuming that there are no heavy segments among the input, i.e. no segment participates in more than K' intersections. Finally, we address the problem of splitting heavy segments into light ones in Section VI.

IV. Batched 1-D Range Counting

Definition 1. *Let $\mathcal{Q}$ be a set of intervals and $\mathcal{P}$ a set of points on the real line. The* 1-D range counting *problem asks to compute for each $q \in \mathcal{Q}$ the value k_q, the number of points in $\mathcal{P}$ that fall within the interval q.*

Theorem 1. *If the input is given as an x-sorted list of points and interval endpoints of combined size N, the batched 1-D range counting problem can be solved using $\mathrm{O}((N + K)/PB + \log P \log_d N + d)$ I/Os in the PEM model, where $K = \sum_{q \in \mathcal{Q}} k_q$ and $d = \max\{2, \min\{\sqrt{N/P}, M/B\}\}$.*

To prove the theorem we present a PEM algorithm that solves the range counting problem within the stated bounds. Let $\mathcal{U} := \mathcal{Q} \cup \mathcal{P}$ be the x-sorted input list, $|\mathcal{U}| = N$. Our counting algorithm ignores the right endpoints of the intervals and represents each interval by its left endpoint. For ease of exposition we abuse the notation slightly and use q to denote both the left endpoint of the interval and the interval itself. When we need to refer to the right endpoint of the interval, we use $\overline{q}$. (Recall that information about $\overline{q}$ is stored with q.) Note, that from an input list that contains both endpoints of each interval, we can generate the list containing only the left endpoints in $\mathrm{O}(N/PB)$ I/Os using the 2-way distribution operation.

We start by presenting a brief overview of the range counting algorithm followed by the details of each step.

We annotate each endpoint $q \in \mathcal{Q}$ with $\mathrm{scc}(q)$, the number of points of $\mathcal{P}$ that precede q in $\mathcal{U}$. Then, the goal of the counting algorithm is for each endpoint $q \in \mathcal{Q}$, to compute and annotate q with $\mathrm{scc}(\overline{q})$, where $\overline{q}$ is the right endpoint of the interval q. The number of points that interval q contains then equals $\mathrm{scc}(\overline{q}) - \mathrm{scc}(q)$ and can be computed from information stored with q.

We partition $\mathcal{U}$ into x-sorted lists $\mathcal{Q}$ and $\mathcal{P}$ using 2-way distribution. Next, we compute the values $\mathrm{scc}(\overline{q})$ in two phases. In the first phase we compute the values $\mathrm{scc}(\overline{q})$ for all intervals q that contain at most dB points. The second phase processes the remaining intervals in rounds. In each round, using binary search we compute the values $\mathrm{scc}(\overline{q})$ and eliminate the intervals that contain an exponentially increasing number of points. Using an amortization argument we show that the large number of points contained within each interval can "pay" for the random accesses required by the binary search.

Now we present the details of the algorithm.

Computing scc(q). Given the x-sorted list $\mathcal{U}$ of points and interval endpoints, we assign a weight 0 to the intervals and 1 to the points. The value of the prefix sum operation on these weights defines the correct values $\mathrm{scc}(q)$ for each interval endpoint q.

Partitioning $\mathcal{U}$ into $\mathcal{Q}$ and $\mathcal{P}$. Partitioning of $\mathcal{U}$ into $\mathcal{Q}$ and $\mathcal{P}$ is simply a 2-way distribution, where the properties are defined by whether the element is an interval endpoint or a point of $\mathcal{P}$.

Phase 1: Processing intervals that contain up to dB points. For each interval $q_j = \mathcal{Q}[j]$ we assign weight $w_{q_j} := \min\{dB, \max\{1, \mathrm{scc}(q_{j+1}) - \mathrm{scc}(q_j)\}\}$. If q_j is the last endpoint in $\mathcal{Q}$, the weight for q_j is defined as $w_{q_j} := \min\{dB, \max\{1, |\mathcal{Q}| - \mathrm{scc}(q_j)\}\}$. Intuitively, the weight w_{q_j} is equal to the number of points of $\mathcal{P}$ between the two interval left endpoints $q_j = \mathcal{Q}[j]$ and $q_{j+1} = \mathcal{Q}[j+1]$, up to a maximum of dB. We use these weights as the input to the *global load balancing* operation to assign intervals of $\mathcal{Q}$ to processors. During the analysis we will show that this choice of weights ensures that the number of intervals assigned to each processor and the number of points of $\mathcal{P}$ that it reads during this phase is at most $(N+K)/P + dB$.

Let $\mathcal{Q}_i$ be the set of intervals assigned to processor p_i. Then p_i scans $\mathcal{Q}_i$ and for each $q \in \mathcal{Q}_i$ counts the number of points that have x-coordinate smaller than $\overline{q}$. Processor p_i accomplishes this by loading points $\mathcal{P}[\text{scc}(q)]$ through $\mathcal{P}[\text{scc}(q)+dB-1]$ into its cache and searching for the x-coordinate of $\overline{q}$ among them. Note, that $\text{scc}(\overline{q})$ equals to the index in $\mathcal{P}$ of the first point that lies to the right of $\overline{q}$. Thus, if p_i finds the x-coordinate of $\overline{q}$ among the dB points, p_i stores $\text{scc}(\overline{q})$ with q. Else, the interval q contains more than dB points and, therefore, q is marked to be processed in the second phase. At the end of the phase, using the 2-way distribution, all marked intervals are copied into a contiguous array $\mathcal{Q}'$ to be used as input for the second phase. When copying each q to $\mathcal{Q}'$, p_i stores the backpointer j – the index of q in list $\mathcal{Q}$ – with the copy of q in $\mathcal{Q}'$. This backpointer will be used in the second phase to quickly find q in the original list $\mathcal{Q}$ to store $\text{scc}(\overline{q})$.

Phase 2: Processing intervals that contain more than dB points. This step proceeds in $\log_d(N/B)-1$ rounds. We maintain the invariant that each interval in the input to each round $r = 1, 2, \ldots$ contains more than $d^r B$ points. (Note, that the first phase ensures that the invariant is satisfied at the beginning of the first round.) Then it follows that in each round r there is a total of at most $\frac{K}{d^r B}$ intervals remaining to be processed. We evenly allocate these intervals among the processors, using global load balancing operation and unit weights for each interval. Each processor p_i is then responsible for a set of intervals $\mathcal{Q}'_i$ of size at most $\frac{K}{d^r PB}$. For each endpoint $q \in \mathcal{Q}'_i$, processor p_i searches for the x-coordinate of $\overline{q}$ using binary search among the points $\mathcal{P}[\text{scc}(q)+d^r B]$ through $\mathcal{P}[\text{scc}(q)+d^{r+1}B-1]$. If the x-coordinate of $\overline{q}$ falls within the range, we store $\text{scc}(\overline{q})$ with q in the original list $\mathcal{Q}$ (using the backpointer j to quickly find it). Else, q contains more than $d^{r+1}B$ points and is marked to be processed in the next round. At the end of the round, all unmarked intervals are deleted and all lists $\mathcal{Q}'$ are compacted to occupy contiguous space to be used as the input for the next round. Note, that any element remains for the next round only if $\text{scc}(\overline{q})$ lies beyond index $\text{scc}(q)+d^{r+1}B$, i.e. q contains more than $d^{r+1}B$ points, therefore, satisfying the invariant for the next round. Since each interval can contain at most N points, after $\log_d(N/B)-1$ rounds there will be no intervals in $\mathcal{Q}'$ remaining to be processed, i.e. counts for all ranges have been computed.

A. Analysis of batched 1-D range counting

In this section we analyze the I/O complexity of the algorithm.

Lemma 1. *The batched 1-D range counting algorithm takes* $O(N/PB + K/PB + d + \log P \log_d N)$ *I/Os, where* $d = \max\{2, \min\{\sqrt{N/P}, M/B\}\}$.

Proof:

Computing scc(q) and partitioning $\mathcal{U}$. These steps of the algorithm consists of a constant number of prefix sums, parallel scans and compaction operations. Thus, it takes $O(N/PB + \log P)$ I/Os.

Analysis of phase 1: Let processor p_i be assigned the set $\mathcal{Q}_i \subseteq \mathcal{Q}$ of intervals and let it read the set $\mathcal{P}_i \subseteq \mathcal{P}$ of points during the first phase of the counting algorithm. We show that the sizes $|\mathcal{Q}_i|$ and $|\mathcal{P}_i|$ are bounded by $O(N/P + dB)$. Since the elements of $\mathcal{Q}_i$ and $\mathcal{P}_i$ are read consecutively, and since each processor's cache can fit $M \geq dB$ elements, once a point is evicted from a processor's cache, it is never loaded again. Thus, the I/O complexity of reading the elements of $\mathcal{Q}_i$ and $\mathcal{P}_i$ will be bounded by $|\mathcal{Q}_i|/B + |\mathcal{P}_i|/B + dB/B = O(N/PB + d)$ I/Os.

The crucial part of phase 1 is the selection of the weights for the global load balancing operation. Let's take a closer look at these weights. As mentioned earlier, the weight of interval represented by the left endpoint $q_j \in \mathcal{Q}$ equals to the number of points between q_j and the next interval left endpoint q_{j+1} (up to a maximum of dB points). The total weight $W = \sum_{q_j \in \mathcal{Q}} w_{q_j} \leq \sum_{q_j \in \mathcal{Q}} 1 + \sum_{q_j \in \mathcal{Q}} (\text{scc}(q_{j+1}) - \text{scc}(q_j)) \leq |\mathcal{Q}| + |\mathcal{P}| = N$. Then the combined weight of elements of $\mathcal{Q}_i$ is $W_i = O(W/P + w_{\max}) = O(N/P + dB)$ by the global load balancing operation. Since each weight $w_{q_j} \geq 1$, it follows that $|\mathcal{Q}_i| \leq W_i = O(N/P + dB)$.

To bound $|\mathcal{P}_i|$, let $S_{q_j} \subseteq \mathcal{P}_i$ be the set of up to the first dB points contained within interval $q_j \in \mathcal{Q}_i$. Let $S'_{q_j} \subseteq S_{q_j}$ be the set of all points of S_{q_j} that lie between the left endpoints q_j and q_{j+1} in $\mathcal{U}$. The definition of the weights says that $w_{q_j} \geq |S'_{q_j}|$. Then the set of points of $\mathcal{P}_i$ that processor p_i reads is equal to

$$|\mathcal{P}_i| = \left| \bigcup_{q_j \in \mathcal{Q}_i} S_{q_j} \right| \leq \sum_{q_j \in \mathcal{Q}_i} |S'_{q_j}| + dB$$
$$\leq \sum_{q_j \in \mathcal{Q}_i} w_{q_j} + dB = O(W_i + dB) = O(N/P + dB)$$

Analysis of phase 2: Consider round r. Since according to the invariant, each interval q contains more than $d^r B$ points, the size of the input at the beginning of round r is $|\mathcal{Q}'| \leq K/(d^r B)$. Each processor p_i gets an equal fraction of these elements, i.e. $|\mathcal{Q}'_i| = O(K/(d^r PB))$ elements. The I/O complexity of the binary search for each $q \in \mathcal{Q}'_i$ is $O(\log \frac{d^{r+1}B}{B}) = O((r+1)\log d)$. Finally, the deletion of the processed elements and the compaction of the remaining elements takes $O(|\mathcal{Q}'_i|/PB + \log P)$ I/Os. Thus, the parallel I/O complexity of round r is $O(\frac{K(r+1)\log d}{d^r PB} + \log P)$ I/Os.

Combining this over all rounds, the total I/O complexity of the second phase is

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

Combining the I/O complexities of each part of the algorithm, concludes the proof of the lemma. ■

Corollary 1. *If $P \leq \min\{N/(B\log^2 N), N/B^2\}$, the batched 1-D range counting problem can be solved in the PEM model in $\mathrm{O}(N/PB + K/PB)$ I/Os, where K is the total sum of the range counts.*

Proof: If $P \leq N/(B\log^2 N)$ then $N/PB \geq \log^2 N \geq \log P \log_d(N/B)$. Also, since $P \leq N/B^2$, it follows that $\sqrt{N/P} \geq B$. Then $d \leq \sqrt{N/P} = \frac{N/P}{\sqrt{N/P}} \leq \frac{N/P}{B} = N/PB$. The corollary follows. ■

Corollary 2. *If $P \leq \min\{N/(B\log N), N/B^2\}$ and $M = B^{\mathrm{O}(1)}$, the batched 1-D range counting problem can be solved in the PEM model in $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ I/Os, where K is the total sum of range counts.*

Proof: If $P \leq N/(B\log N)$, then $\log(N/B) \leq N/PB$. And since $d \leq N/PB$ (see proof of Corollary 1), it follows that $\mathrm{O}(N/PB + K/PB + d + \log P \log_d(N/B)) = \mathrm{O}(N/PB + K/PB + \log(N/B)\log_d P) = \mathrm{O}(\frac{N}{PB}\log_d P + K/PB) = \mathrm{O}(\mathrm{sort}_P(N) + K/PB)$. The last equality has been proven in [7] for $d = \max\{2, \min\{\sqrt{N/P}, M/B\}\}$ and under the assumption that $M = B^{\mathrm{O}(1)}$. ■

B. Batched 1-D range counting on independent inputs

For the distribution sweeping framework, we need to solve range counting problem on a set of different slabs in parallel. The next theorem says that we can do it for up to $r = \mathrm{O}(P)$ independent instances.

Theorem 2. *Let $A_1, A_2, \ldots, A_r$ be r independent instances of batched 1-D range counting problems. If each $A_i = \mathcal{Q}_i \cup \mathcal{P}_i$ is x-sorted, $r = \mathrm{O}(P)$ and $\sum_{i=1}^{r} |A_i| = N$, then all these instances can be solved in $\mathrm{O}((N+K)/PB + \log P \log_d N + d)$ I/Os in the PEM model, where $K = \sum_{i=1}^{r}\sum_{q\in\mathcal{Q}_i} k_q$ and $d = \max\{2, \min\{\sqrt{N/P}, M/B\}\}$.*

Proof: By replacing the parallel scan and prefix sum operations of the range counting algorithm with the corresponding segmented versions, we ensure that these operations still perform within $\mathrm{O}(N/PB + \log P)$ I/Os.

The requirement that $r = \mathrm{O}(P)$ is equivalent to the requirement of the global load balancing operation, thus we can use it to allocate elements of the input lists $A_1, \ldots, A_r$ to the processors.

The rest of the operations are performed by processors sequentially on the set of elements that each one is assigned. Multiple instances do not change the I/O complexity of the algorithm, as long as the elements allocated to a single processor do not come from more than a constant number of different instances. This last criteria is satisfied by the properties of the global load balancing operation. ■

Corollary 3. *If $P \leq \min\{N/(B\log^2 N), N/B^2\}$, the batched 1-D range counting problem on up to $\mathrm{O}(P)$ independent instance of combined size N and total range counts equal K can be solved in the PEM model in $\mathrm{O}(N/PB + K/PB)$ I/Os.*

Corollary 4. *If $P \leq \min\{N/(B\log N), N/B^2\}$, the batched 1-D range counting problem on up to $\mathrm{O}(P)$ independent instance of combined size N and total range counts equal K can be solved in the PEM model in $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ I/Os.*

The proofs for these corollaries are similar to the corresponding Corollaries 1 and 2.

V. Optimal Orthogonal Line Segment Intersection Reporting

In this section we assume that no vertical segment participates in more than $K' := \max\{N/P, K/(P\log_d P)\}$ intersections.

As mentioned in Section III, to solve the orthogonal line segment intersection reporting problem, we run the distribution sweeping framework from [6], and computing the number of intersections, $K_k(v)$, that each vertical segment v participates in at invocation I_σ^k, by using our range counting algorithm presented in Section IV. There are at most P slabs σ at any given recursive level of the parallel distribution sweeping framework. Thus, we can use Theorem 2 to efficiently compute the range counting problem on up to P independent instances. The only remaining task is to bound the sizes of all these instances of the range counting problems at the k^{th} level of recursion.

At the k^{th} recursive level, each vertical segment appears only in one invocation. Thus, the total number of vertical segments is at most N. A horizontal segment may appear in multiple lists $S_{\sigma_j}^k$. However, a horizontal segment appears in $S_{\sigma_j}^k$ if and only if it participates in at least one intersection. Let K_k denote the total number of intersection to be reported at the k^{th} recursive level. Then the total size of all lists $S^k_{\sigma_j}$ is at most K_k. The total size of all lists $R^k_{\sigma_j} = V^k_{\sigma_j} \cup S^k_{\sigma_j}$ at the k^{th} recursive level is, therefore, bounded by $\sum_{\sigma_j} |R^k_{\sigma_j}| = \sum_{\sigma_j} |V^k_{\sigma_j}| + \sum_{\sigma_j} |S^k_{\sigma_j}| \leq N + K_k$. The total sum of counts is also bounded by the total number of intersections to be reported at the k^{th} recursive level, that is, K_k. Therefore, using Corollary 3, the range counting problem, and, consequently, computing the values $K_k(v)$ for each vertical segment $v \in V^k_{\sigma_j}$ on the k^{th} level of recursion can be performed in $\mathrm{O}((N + K_k)/PB)$ I/Os. By summing this cost over all levels of recursion, we obtain the I/O complexity of $\sum_{k=1}^{\log_d P}(N/PB + K_k/PB) = \mathrm{O}(\frac{N}{PB} \log_d P + K/PB) = \mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ for all counting steps. This matches the cost of the remainder of the algorithm and, thus, leads to an optimal solution with $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ I/O complexity. The space complexity of the distribution sweeping framework of [6] is $\mathrm{O}(N + K)$ and remains unchanged.

This proves the following theorem.

Theorem 3. *In the PEM model, orthogonal line segment intersection reporting can be solved in* $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ *I/Os using* $\mathrm{O}(N + K)$ *space, provided* $P \leq \min\{\frac{N}{B \log^2 N}, \frac{N}{B^2}\}$.

A careful reader will notice that the maximum number of processors of this solution to the line segment intersection reporting problem decreased by a factor of $\mathrm{O}(\log N)$ compared to the one presented in [6]. Using Corollary 4 and the same argument as above, one can easily observe that the above algorithm achieves $\mathrm{O}(\mathrm{sort}_P(N) \log_d P + K/PB)$ I/O complexity for $P \leq \min\{\frac{N}{B \log N}, \frac{N}{B^2}\}$, matching the bounds of Ajwani et al. [6] and, consequently, providing an alternative solution.

However, the following theorem shows that we can still achieve the optimal $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ I/O complexity for the same range of processors, at the expense of using slightly more space. This trade-off between space and scalability was not possible in the previous solution.

Theorem 4. *In the PEM model, orthogonal line segment intersection reporting can be solved in* $\mathrm{O}(\mathrm{sort}_P(N) + K/PB)$ *I/Os and* $\mathrm{O}(N \log_d P + K)$ *space, provided* $P \leq \min\{\frac{N}{B \log N}, \frac{N}{B^2}\}$ *and for* $d = \max\{2, \min\{\sqrt{N/P}, M/B\}\}$.

Proof: The parallel distribution sweeping algorithm processes each level of recursion in parallel using all processors, one recursive level at a time. A recursive level doesn't depend on the reporting performed at other recursive levels. Thus, the input lists Y^k_σ and R^k_σ can be generated for all recursive levels at the beginning of the algorithm. Then we run the range counting algorithm as before, but this time on the lists of *all* recursive levels at once. The total number of these lists is still $\mathrm{O}(P)$, thus, we can use Theorem 2 to bound the I/O complexity of intersection counting.

Each vertical segment appears at most once at each recursive level, while each horizontal segment appears in the input of a slab if it participates in an intersection or has an endpoint in that slab. Thus, over $\mathrm{O}(\log_d P)$ recursive levels, the total size of all lists is at most $\mathrm{O}(N \log_d P + K)$. Thus, if $P \leq \min\{N/(B \log N), N/B^2\}$ then $\log N = \mathrm{O}(N/PB)$ and $d = \mathrm{O}(N/PB)$ and by Theorem 2 the I/O complexity of the counting step is bounded by

$$\begin{aligned}
& \mathrm{O}\left(\frac{N \log_d P + K}{PB} + \log P \log_d N + d\right) \\
= \; & \mathrm{O}\left(\mathrm{sort}_P(N) + K/PB + \log N \log_d P + d\right) \\
= \; & \mathrm{O}\left(\mathrm{sort}_P(N) + K/PB + \frac{N}{PB} \log_d P\right) \\
= \; & \mathrm{O}\left(\mathrm{sort}_P(N) + K/PB\right)
\end{aligned}$$

The space complexity is increased to $\mathrm{O}(N \log_d P + K)$ for storing all the lists.

■

In the next section we solve the remaining task, namely, preprocessing the input splitting heavy segments into a set of light ones.

VI. Dividing long segments

In this section we describe how to split heavy segments, i.e. the segments that participate in more than $K' := \max\{N/P, K/(P \log_d P)\}$ intersections into a set of light ones. To be consistent with our definitions of slabs for the distribution sweeping, we present a method of splitting the horizontal segments. The vertical segments can be processed analogously.

We start by counting the number of intersections each horizontal segment is involved in. An algorithm to perform this in $\mathrm{O}(\mathrm{sort}_P(N))$ I/Os has been presented in [6]. Using a 2-way distribution operation, which takes $\mathrm{O}(N/PB)$ I/Os, we extract the list H_L of heavy horizontal segments, and in the remainder of this section we consider only these horizontal segments. For each segment $h \in H_L$ we generate a list L_h of x-coordinates such that the segments induced by splitting h at these x-coordinates participate in $\mathrm{O}(K')$ intersections each. Thus, each pair of adjacent entries in each L_h defines a new segment that participates in at most $\mathrm{O}(K')$ intersections. Note, that the total number of generated segments is at most $K/K' \leq P \log_d P = \mathrm{O}(N)$, because $P \leq N/\log N$. Thus, the total size of generated lists L_h is bounded by $\mathrm{O}(N)$, i.e. the number of newly generated segments is still linear with the number of original segments. It is important to note that maintaining the the entries of each list L_h in order and in contiguous space during generation would add unnecessary complications to the algorithm. Instead, we generate these entries as tuples (h, x_i) and construct the lists L_h at the end of the algorithm by sorting the set of tuples lexicographically. Since, $\sum_{h \in H_L} |L_h| = \mathrm{O}(N)$, this sorting step, which is performed only once, does not affect the overall I/O complexity of the algorithm.

We generate the entries of L_h for all $h \in H_L$ by adapting the $\mathrm{O}(\mathrm{sort}_P(N+K))$ solution described in [6]. In particular, we defer the generation of entries of L_h to leaf invocations. Thus, at each non-leaf invocation I_σ^k we only generate the input lists $Y_{\sigma_j}^{k+1}$ for the child invocations $I_{\sigma_j}^{k+1}$. We will show that our approach for generating the child lists guarantees that at a leaf invocation I_σ^k we simply need to cut each horizontal segment $h \in Y_\sigma^k$ at the slab boundaries of σ. Since the combined size of lists L_h is linear, the combined size of the input lists Y_σ^k at each recursive level are also bounded by $\mathrm{O}(N)$, resulting in optimal $\mathrm{O}(\mathrm{sort}_P(N))$ solution over $\mathrm{O}(\log_d P)$ recursive levels.

Let us describe the algorithm for generating child lists $Y_{\sigma_j}^{k+1}$. Given a list Y_σ^k, any vertical segment that lies within slab σ_j is added to the list $Y_{\sigma_j}^{k+1}$. Consider a horizontal segment $h = (x_l, x_r, y) \in Y_\sigma^k$. Segment h is added to the child list $Y_{\sigma_j}^{k+1}$ if σ_j contains one of its endpoints. Now, assume h fully spans some slab σ_j. Let $x(\sigma_j^l)$ and $x(\sigma_j^r)$ denote the x-coordinates of the left and the right boundaries of σ_j. Consider the following subsegments of h: $h_1 = (x_l, x(\sigma_j^l), y)$ and $h_2 = (x_l, x(\sigma_j^r), y)$. Let $w(h_1)$ and $w(h_2)$ be the number of vertical segments that h_1 and h_2 intersect. Note, that if $\lfloor w(h_1)/K' \rfloor < \lfloor w(h_2)/K' \rfloor$, we would need to cut h within the slab σ_j, that is, add to L_h some x-coordinate which falls within the range of slab σ_j. Thus, we would like to copy h to $Y_{\sigma_j}^{k+1}$ if this condition is satisfied. Then, any horizontal segment belongs to the input list Y_σ^k of a *leaf* invocation I_σ^k, either because it has an endpoint within σ or because it should be cut within σ. However, each leaf invocation contains at most N/P vertical segments, i.e., each horizontal segment intersects at most N/P vertical segments within the leaf slab σ. Thus, it does not matter where we cut h within σ. So we can cut h at the boundaries of σ by creating the tuples $(h, x(\sigma^l))$ and $(h, x(\sigma^r))$.

Unfortunately, the distribution sweeping framework cannot compute $w(h_1)$ and $w(h_2)$ for each horizontal segment until it reaches leaf invocations. Thus, we cannot make a decision about the membership of h in the child lists in invocation I_σ^k. However, as we show below, we can compute these values if both endpoints of h_1 and h_2 coincide with boundaries of some slabs (not necessarily at the same invocation level). To enforce this requirement we split each horizontal segment h into two segments at the slab boundary b_h that h crosses in the earliest possible invocation. We say the two segments are *anchored* at b_h. The rest of the algorithm proceeds treating the two segments as independent segments and counting intersection from the anchor point b_h, rather than the left endpoint of the segment.

Now we formally describe the details of the algorithm. We start by adding the left and right endpoint of each segment $h = (x_l, x_r, y)$ to L_h, that is, we generate tuples (h, x_l) and (h, x_r).

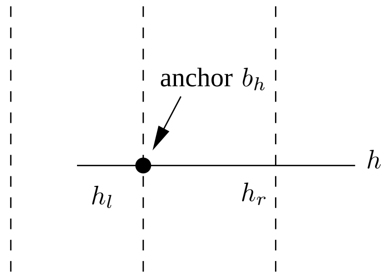

Figure 3. An example of anchoring segment h at b_h.

At each invocation I_σ^k consider a horizontal segment $h \in Y_\sigma^k$. If h is completely contained within a child slab σ_j of σ, we add h to $Y_{\sigma_j}^{k+1}$ without any further processing.

If h has not been anchored yet and intersects at least one slab boundary among the child slabs of σ, we add the leftmost such boundary b_h to L_h (by generating the tuple $(h, x(b_h))$) and split h into segments $h_l = (x_l, x(b_h), y)$ and $h_r = (x(b_h), x_r, y)$. We initialize the weight of each of the two segments at 0 and from now on maintain the two weights separately. Note that since b_h is the leftmost slab boundary that h intersects, h_l is completely contained within the child slab σ_j, whose right boundary is b_h. Hence, we add h_l to $Y_{\sigma_j}^{k+1}$. We process h_r in the same way as we process an anchored segment $h \in Y_\sigma^k$, as discussed next.

If h is an anchored segment, assume w.l.o.g. that it is left-anchored, that is, its left endpoint is b_h (the right-anchored segments are processed symmetrically). Let $w(h)$ denote the weight of h and let $\sigma_l, \sigma_{l+1}, \ldots, \sigma_r$ be the child slabs of σ completely spanned by h. For $l \leq j \leq r$, let w_j be the number of intersections h has in slab σ_j. (The original description of generating the lists $Y_{\sigma_j}^{k+1}$ in [6] computes the values w_j, but is only interested in whether $w_j > 0$ and, therefore, discards them. For our purposes here, we maintain these values.) For $l \leq j \leq r$, let $w'_j = w(h) + \sum_{k=l}^{j} w_k$ and define $w'_{l-1} = w(h)$. We add segment h to $Y_{\sigma_j}^{k+1}$, for $l \leq j \leq r$, if $\lfloor w'_{j-1}/K' \rfloor < \lfloor w'_j/K' \rfloor$. In this case, the copy of h in $Y_{\sigma_j}^{k+1}$ receives weight w'_{j-1}. If the right endpoint of h is interior to slab σ_{r+1}, we also add h to $Y_{\sigma_{r+1}}^{k+1}$, with weight w'_r.

Let us analyze the I/O complexity of generating the lists $Y_{\sigma_j}^{k+1}$. A segment h is added to list $Y_{\sigma_j}^k$ if and only if one of its endpoints is contained in σ_j or h must be split within σ. The last condition is equivalent to one of the entries of L_h lying within the range spanned by slab σ_j. Thus, the total size of the lists Y_σ^k at each invocation level k is bounded by $\sum_\sigma |Y_\sigma^k| = \mathrm{O}(N + \sum_{h \in H_L} |L_h|) = \mathrm{O}(N)$. Then, by Lemma 1 of [6], the I/O complexity of generating the lists Y_σ^k at each level of recursion is $\mathrm{O}(N/PB)$. Over $\log_d P$ levels of recursion, the total I/O complexity of generating

L_h adds up to $\mathrm{O}(\frac{N}{PB}\log_d P) = \mathrm{O}(\mathrm{sort}_P(N))$.

As mentioned before, at the leaf invocations I_σ^k we generate tuples $(h, x(\sigma^l))$ and $(h, x(\sigma^r))$, which requires a parallel scan of the lists Y_σ^k. Since the combined size of these lists is linear, this step takes $\mathrm{O}(N/PB)$ I/Os.

Finally, sorting the tuples to generate the lists L_h takes $\mathrm{O}(\mathrm{sort}_P(N))$ I/Os.

Combining the I/O complexities of each step and repeating the process for vertical segments proves the following theorem.

Theorem 5. *If $P \leq \min\{N/B^2, N/\log N\}$, a list of N horizontal and vertical segments can be preprocessed in the PEM model in $\mathrm{O}(\mathrm{sort}_P(N))$ I/Os, such that each segment that participates in more than $K' := \max\{N/P, K/(P\log_d P)\}$ intersections is divided into smaller segments and no segment in the output participates in more than K' intersections.*

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

[7] 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.

[8] 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.

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

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

[11] 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.

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

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

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

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

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

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

Without loss of generality, we also assume that every array A_i is aligned at block boundaries 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 $A_1, A_2, \ldots, A_r$, 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 $W/P + w_{\max}$. 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 $W/P + w_{\max}$.