Research Repository UCD

Title	Geometric Algorithms for Private-Cache Chip Multiprocessors
Authors(s)	Ajwani, Deepak, Sitchinava, Nodari, Zeh, Norbert
Publication date	2010-09-08
Publication information	Ajwani, Deepak, Nodari Sitchinava, and Norbert Zeh. “Geometric Algorithms for Private-Cache Chip Multiprocessors.” Springer, September 8, 2010. https://doi.org/10.1007/978-3-642-15781-3_7.
Conference details	The 18th Annual European Symposium (ESA 2010), Liverpool, United Kingdom, 6-8 September 2010
Series	Lecture Notes in Computer Science (LNCS. volume 6347)
Publisher	Springer
Item record/more information	http://hdl.handle.net/10197/9902
Publisher's statement	The final publication is available at www.springerlink.com.
Publisher's version (DOI)	10.1007/978-3-642-15781-3_7

Downloaded 2026-05-01 06:04:38

The UCD community has made this article openly available. Please share how this access benefits you. Your story matters! (@ucd_oa)

© Some rights reserved. For more information

Geometric Algorithms for Private-Cache Chip Multiprocessors (Extended Abstract)

Deepak Ajwani[1], Nodari Sitchinava[1], and Norbert Zeh[2,⋆]

[1] MADALGO[⋆⋆], Department of Computer Science, University of Aarhus, Denmark
{ajwani,nodari}@cs.au.dk
[2] Faculty of Computer Science, Dalhousie University, Halifax, Canada
nzeh@cs.dal.ca

Abstract. We study techniques for obtaining efficient algorithms for geometric problems on private-cache chip multiprocessors. We show how to obtain optimal algorithms for interval stabbing counting, 1-D range counting, weighted 2-D dominance counting, and for computing 3-D maxima, 2-D lower envelopes, and 2-D convex hulls. These results are obtained by analyzing adaptations of either the PEM merge sort algorithm or PRAM algorithms. For the second group of problems—orthogonal line segment intersection reporting, batched range reporting, and related problems—more effort is required. What distinguishes these problems from the ones in the previous group is the variable output size, which requires I/O-efficient load balancing strategies based on the contribution of the individual input elements to the output size. To obtain nearly optimal algorithms for these problems, we introduce a parallel distribution sweeping technique inspired by its sequential counterpart.

1 Introduction

With recent advances in multicore processor technologies, parallel processing at the chip level is becoming increasingly mainstream. Current multicore chips have 2, 4 or 6 cores, but Intel recently announced a 48-core chip [20], and the trend to increasing numbers of cores per chip continues. This creates a need for algorithmic techniques to harness the power of increasing chip-level parallelism [17]. A number of papers have made progress towards addressing this need [2,3,9,11–13].

Ignoring the presence of a memory hierarchy, current multicore chips resemble a PRAM, with all processors having access to a shared memory and communicating with each other exclusively through shared memory accesses. However, each processor (core) has a low-latency *private cache* inaccessible to other processors. In order to take full advantage of such architectures, now commonly known as private-cache chip multiprocessors (CMP's), algorithms have to

⋆ Supported in part by the Natural Sciences and Engineering Research Council of Canada and the Canada Research Chairs programme.

⋆⋆ MADALGO is the Center for Massive Data Algorithmics, a center of the Danish National Research Foundation.

be designed with a focus on minimizing the number of accesses to shared memory. In this paper, we study techniques to address this problem for a number of geometric problems, specifically for 2-D dominance counting, 3-D maxima, 2-D lower envelope, 2-D convex hull, orthogonal line segment intersection reporting, batched 2-D orthogonal range reporting, and related problems.

For these problems, optimal PRAM [5, 7, 14, 18] and sequential I/O-efficient algorithms [10, 19] are known, and some of these problems have also been studied in coarse-grained parallel models [15, 16]. The previous parallel algorithms and the I/O-efficient sequential algorithms achieve exactly one of our goals—parallelism or I/O efficiency—while the algorithms in this paper achieve both.

1.1 Model of Computation and Previous Work

Our algorithms are designed in the *parallel external memory* (PEM) model of [2]; see Figure 1. This model considers a machine with P processors, each with a *private cache* of size M. Processors communicate with each other through access to a *shared memory* of conceptually unlimited size. Each processor can use only data in its private cache for computation. The caches and the shared memory are divided into *blocks* of size B. Data is transferred between the caches and shared memory using *parallel input-output* (I/O) operations. During each such operation, each processor can transfer one block between shared memory and its private cache. The cost of an algorithm is the number of I/Os it performs. As in the PRAM model, different assumptions can be made about how to handle multiple processors reading or writing the same block in shared memory during one I/O operation. Throughout this paper, we allow concurrent reading of the same block by multiple processors but disallow concurrent block writes; in this respect, the model is similar to a CREW PRAM. The cost of sorting in the PEM model is $\text{sort}_P(N) = O\left(\frac{N}{PB} \log_{M/B} \frac{N}{B}\right)$ [2], provided $P \leq N/B^2$ and $M = B^{O(1)}$.

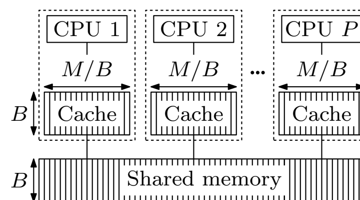

Fig. 1. PEM model

The PEM model provides the simplest possible abstraction of current multicore chips, focusing on the fundamental I/O issues that need to be addressed when designing algorithms for these architectures, similar to the I/O model [1] in the sequential setting. The hope is that the developed techniques are also applicable to more complicated multicore models. For the PEM graph algorithms developed in [3], this has certainly been the case already [13].

A number of other results have been obtained in more complicated multicore models. In [8], Bender et al. discussed how to support concurrent searching and updating of cache-oblivious B-trees by multiple processors. In [9, 11, 12], different multicore models are considered and cache- and processor-oblivious divide-and-conquer and dynamic programming algorithms are presented whose performance is within a constant factor of optimal for the studied problems.

An important difference between the work presented in this paper and the previous results mentioned above is that the algorithms in this paper are output-

sensitive. This creates a challenge in allocating input elements to processors so that all processors produce roughly equal fractions of the output. To the best of our knowledge, output-sensitive computations have not been considered before in any of the multicore models mentioned above. However, there exists related work in the sequential I/O [1, 19] and cache-oblivious models [4, 10], and in the PRAM [14, 18] model. The PRAM solutions rely on very fine-grained access to shared memory, while the cache-efficient solutions seem inherently sequential.

1.2 New Results

In this paper, we focus on techniques to solve fundamental computational geometry problems in the PEM model. The main contribution of the paper is a parallelization of the *distribution sweeping* paradigm [19], which has proven very successful as a basis for solving geometric problems in the sequential I/O model. Using this technique, we obtain solutions for reporting orthogonal line segment intersections, batched range searching, and related problems.

The above problems can be solved using $\Theta(\text{sort}_P(N) + K/PB)$ I/Os in the sequential I/O model ($P = 1$) and in the CREW PRAM model ($M, B = \text{O}(1)$). Here, K denotes the output size. Thus, it seems reasonable to expect that a similar I/O bound can be achieved in the PEM model. We don't achieve this goal in this paper but present two algorithms that come close to it: one performing $\text{O}(\text{sort}_P(N + K))$ I/Os, the other $\text{O}(\text{sort}_P(N) \log_d P + K/PB)$, for $d := \min(\sqrt{N/P}, M/B)$. The main challenge in obtaining these solutions is to balance the output reporting across processors, as different input elements may make different contributions to the output size. Our solutions are obtained using two different solutions to this balancing problem.

Our solutions are based on $\text{O}(\text{sort}_P(N))$ I/O algorithms for the counting versions of these problems. We also obtain optimal $\text{O}(\text{sort}_P(N))$ I/O solutions for computing the lower envelope of a set of non-intersecting line segments in the plane, the maxima of a 3-D point set, and the convex hull of a 2-D point set.

2 Tools

In this section, we define primitives we use repeatedly in our algorithms. Unless stated otherwise, we assume $P \le \min(N/B^2, N/(B \log N))$ and $M = B^{\text{O}(1)}$.

Prefix sum and compaction. Given an array $A[1 \mathinner{..} N]$, the *prefix sum* problem is to compute an array $S[1 \mathinner{..} N]$ such that $S[i] = \sum_{j=1}^{i} A[j]$. Given a second boolean array $M[1 \mathinner{..} N]$, the *compaction* problem is to rearrange the array A such that all elements $A[i]$ with $M[i] = \text{true}$ are together at the beginning of A and the relative order of elements with same value of $M[i]$ remains unchanged. PEM algorithms for these problems with I/O complexity $\text{O}(N/PB + \log P)$ are presented in [2] (also see [21]). Since we assume $P \le N/(B \log N)$, the I/O complexity of both operations reduces to $\text{O}(N/PB)$.

Global load balancing. Let $A_1, A_2, \ldots, A_r$ be a collection of arrays with $r \le P$ 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 $\mathrm{O}(1)$ 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 using $\mathrm{O}(1)$ prefix sum and compaction operations and, thus, takes $\mathrm{O}(N/PB)$ I/Os. Details of this operation will appear in the full paper.

Transpose and compact. Given P arrays $A_1, A_2, \ldots, A_P$ of total size N and such that each array A_i is segmented into d sub-arrays $A_{i,1}, A_{i,2}, \ldots, A_{i,d}$, a *transpose and compact* operation generates d arrays $A'_1, A'_2, \ldots, A'_d$, where A'_j is the concatenation of arrays $A_{1,j}, A_{2,j}, \ldots, A_{P,j}$. The segmentation is assumed to be given as a $P \times d$ matrix M stored in row-major order and such that $M[i, j]$ is the size of array $A_{i,j}$. A transpose and compact operation can be implemented using $\mathrm{O}(N/PB + d)$ I/Os as follows.

We copy M into a matrix M' and round every entry in M' up to the next multiple of B. We add a 0th column to M and a 0th row to M', all of whose entries are 0, and compute row-wise prefix sums of M and column-wise prefix sums of M'. Let the resulting matrices be M^r and M^c, respectively. Array $A_{i,j}$ needs to be copied from position $M^r[i, j-1]$ in A_i to position $M^c[i-1, j]$ in A'_j. We assign portions of the arrays $A_1, A_2, \ldots, A_P$ to processors using a global load balancing operation so that no processor receives more than $\mathrm{O}(N/P + B) = \mathrm{O}(N/P)$ elements and the pieces assigned to processors, except the last piece of each array $A_{i,j}$, have sizes that are multiples of B. Each processor copies its assigned blocks of arrays $A_1, A_2, \ldots, A_P$ to arrays $A'_1, A'_2, \ldots, A'_d$. Finally, we use a compaction operation to remove the gaps introduced in arrays $A'_1, A'_2, \ldots, A'_d$ by the alignment of the sub-arrays $A_{i,j}$ at block boundaries.

Note that the size of the arrays $A'_1, A'_2, \ldots, A'_d$ with the sub-arrays $A_{i,j}$ padded to full blocks is at most $N + Pd(B-1)$. Thus, the prefix sum, compaction, and global load balancing operations involved in this procedure can be carried out using $\mathrm{O}(N/PB + d)$ I/Os. The row-wise and column-wise prefix sums on matrices M and M' can also be implemented in this bound. However, M' needs to be stored in column-major order for this operation. This is easily achieved by transposing M' using $\mathrm{O}(d)$ I/Os (as its size is only $(P+1) \times d$) and then transposing it back into row-major order after performing the prefix sum.

3 Counting Problems

Interval stabbing counting and 1-D range counting. Let I be a set of intervals, S a set of points on the real line, and $N := |I| + |S|$. The *interval stabbing counting* problem is to compute the number of intervals in I containing each point in S. The *1-D range counting* problem is to compute the number of points in S contained in each interval in I.

Theorem 1. *Interval stabbing counting and 1-D range counting can be solved using* $\mathrm{O}(\mathrm{sort}_P(N))$ *I/Os in the PEM model. If the input is given as an x-sorted list of points and interval endpoints, interval stabbing counting and 1-D range counting take* $\mathrm{O}(N/PB)$ *and* $\mathrm{O}(\mathrm{sort}_P(|I|) + |S|/PB)$ *I/Os, respectively.*

Proof. Given the x-sorted list of points and interval endpoints, the number of intervals containing a point $q \in S$ is the prefix sum of q after assigning a weight of 1 to every left interval endpoint, a weight of -1 to every right interval endpoint, and a weight of 0 to every point in S. Thus, the interval stabbing problem can be solved using a single prefix sum operation, which takes $\mathrm{O}(N/PB)$ I/Os.

The number of points contained in an interval in I is the difference of the prefix sums of its endpoints after assigning a weight of 1 to every point in S and a weight of 0 to every interval endpoint. This prefix sum operation takes $\mathrm{O}(N/PB)$ I/Os again. To compute the differences of the prefix sums of the endpoints of each interval, we extract the set of interval endpoints from the x-sorted list using a compaction operation and sort the resulting list to store the endpoints of each interval consecutively. This takes another $\mathrm{O}(\mathrm{sort}_P(|I|) + |S|/PB)$ I/Os, for a total of $\mathrm{O}(\mathrm{sort}_P(|I|) + |S|/PB)$ I/Os.

If the x-sorted list of points and interval endpoints is not given, it can be produced from I and S using $\mathrm{O}(\mathrm{sort}_P(N))$ I/Os, which dominates the total cost of the computation. ☐

2-D weighted dominance counting. Given two points $q_1 = (x_1, y_1)$ and $q_2 = (x_2, y_2)$ in the plane, we say that q_1 1-*dominates* q_2 if $y_1 \geq y_2$; q_1 2-*dominates* q_2 if, in addition, $x_1 \geq x_2$. The latter is the standard notion of 2-D dominance. In the *2-D weighted dominance counting* problem, we are given a set S of points, each with an associated weight $w(q)$, and our goal is to compute the total weight of all points in S 2-dominated by each point in S. Our algorithm in Section 4 for orthogonal line segment intersection reporting requires us to count the number of intersection of each segment. This problem and the problem of 2-D batched range counting reduce to 2-D weighted dominance counting by assigning appropriate weights to segment endpoints or points [6]. Thus, it suffices to present a solution to 2-D weighted dominance counting here.

Theorem 2. *2-D weighted dominance counting can be solved using* $\mathrm{O}(\mathrm{sort}_P(N))$ *I/Os in the PEM model, provided* $P \leq N/B^2$ *and* $M = B^{\mathrm{O}(1)}$.

Proof. We start by sorting the points in S by their x-coordinates and partitioning the plane into vertical slabs σ_i, each containing N/P points. Each processor p_i is assigned one slab σ_i and produces a y-sorted list $U(\sigma_i)$ of points in this slab, each annotated with labels $W^1_{\sigma_i}(q)$ and $W^2_{\sigma_i}(q)$, which are the total weights of the points within σ_i that q 1- and 2-dominates, respectively. After the initial sorting step to produce the slabs, which takes $\mathrm{O}(\mathrm{sort}_P(N))$ I/Os, the lists $U(\sigma_i)$ and the labelling of the points in these lists can be produced using $\mathrm{O}(\mathrm{sort}_1(N/P))$ I/Os using standard I/O-efficient techniques [19] independently on each processor.

We merge these lists using the d-way cascading merge procedure of PEM merge sort [2], which takes $\mathrm{O}(\mathrm{sort}_P(N))$ I/Os and can be viewed as a d-ary tree with leaves $\sigma_1, \sigma_2, \ldots, \sigma_P$ and $\log_d P$ levels. At each tree node v, the procedure computes a y-sorted list $U(v)$, which is the merge of the y-sorted lists $U(\sigma_i)$ associated with the leaves of the subtree with root v. Next we observe that we can augment the merge procedure at each node v to compute weights $W^1_v(q)$ and $W^2_v(q)$, which are the total weights of the points in $U(v)$ 1- and 2-dominated by q,

respectively. For the root r of the merge tree, we have $U(r) = S$, and $W_r^2(q)$ is the total weight of the points dominated by q, for each $q \in U(r)$.

So consider a node v with children $w_1, w_2, \ldots, w_d$. The cascading merge produces list $U(v)$ in rounds, in each round merging finer samples of the lists $U(w_1), U(w_2), \ldots, U(w_d)$ than in the previous round. In the round that produces the full list $U(v)$ from full lists $U(w_1), U(w_2), \ldots, U(w_d)$, the processor placing a point $q \in U(w_i)$ into $U(v)$ also accesses the predecessor $\text{prd}_{w_j}(q)$ of q in list $U(w_j)$, for all $1 \leq j \leq d$, which is the point in $U(w_j)$ with maximum y-coordinate no greater than q's. Now it suffices to observe that $W_v^1(q)$ and $W_v^2(q)$ can be computed as $W_v^1(q) = \sum_{j=1}^{d} W_{w_j}^1(\text{prd}_{w_j}(q))$ and $W_v^2(q) = W_{w_i}^2(q) + \sum_{j=1}^{i-1} W_{w_j}^1(\text{prd}_{w_j}(q))$. This does not increase the cost of the merge step, and the total I/O complexity of the algorithm is $\mathrm{O}(\text{sort}_P(N))$. □

4 Parallel Distribution Sweeping

We discuss our *parallel distribution sweeping* framework using orthogonal line segment intersection reporting as an example. *Batched orthogonal range reporting* and *rectangle intersection reporting* can be solved in the same complexity using adaptations of the procedure in this section. Details of these adaptations will be given in the full version of the paper.

The distribution sweeping technique recursively divides the plane into vertical slabs, starting with the entire plane as one slab and in each recursive step dividing the given slab into d child slabs, for an appropriately chosen parameter d. This division is chosen so that each slab at a given level of recursion contains roughly the same number of objects (e.g., segment endpoints and vertical segments). In the sequential setting [19], $d = M/B$, and the recursion stops when the input problem fits in memory. In the parallel setting, we set $d := \min\{\sqrt{N/P}, M/B\}$,[3] and the lowest level of recursion divides the plane into P slabs, each containing about N/P input elements. Viewing the recursion as a rooted tree, we talk about leaf invocations and children of a non-leaf invocation. We refer to an invocation on slab σ at the kth recursive level as I_σ^k.

We describe two variants of parallel distribution sweeping. In both variants, each invocation I_σ^k receives as input a y-sorted list Y_σ^k containing horizontal segments and vertical segment endpoints, and the root invocation $I_{\mathbb{R}^2}^0$ contains all horizontal segments and vertical segment endpoints in the input. 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, $E_{\sigma_j}^k$ 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. The first distribution sweeping variant constructs $Y_{\sigma_j}^{k+1}$ as the merge of lists $E_{\sigma_j}^k$, $S_{\sigma_j}^k$, and $V_{\sigma_j}^k$ and recurses on each child invocation $I_{\sigma_j}^{k+1}$ with this input.

[3] The choice of d comes from the d-way PEM mergesort of [2] and ensures that $d = \mathrm{O}(N/PB)$.

The second variant constructs a y-sorted list $R^k_{\sigma_j} := S^k_{\sigma_j} \cup V^k_{\sigma_j}$, for each child slab σ_j, reports all intersections between segments in $R^k_{\sigma_j}$, and then recurses on each child invocation $I^{k+1}_{\sigma_j}$ with input $Y^{k+1}_{\sigma_j} := E^k_{\sigma_j} \cup V^k_{\sigma_j}$; see Figure 2. In both variants, every leaf invocation I^k_σ finds all intersections between the elements in Y^k_σ using sequential I/O-efficient techniques, even though some effort is required to balance the work among processors.

The first variant, with I/O complexity $O(\text{sort}_P(N+K))$, defers the reporting of intersections to the leaf invocations and ensures that the input to every leaf I^k_σ invocation is exactly the list of vertical segment endpoints in σ and of all horizontal segments with an endpoint or an intersection in σ. The second variant achieves an I/O complexity of $O(\text{sort}_P(N)\log_d P + K/PB)$ and is similar to the sequential distribution sweeping technique in that each non-leaf invocation I^k_σ finds all intersections between vertical segments in each child slab σ_j and horizontal segments spanning this slab and then recurses on each slab σ_j to find intersections between segments with at least one endpoint in this slab.

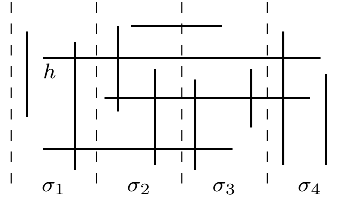

Fig. 2. When deferring intersection reporting to the leaves, we have $h \in Y^{k+1}_{\sigma_j}$, for $j \in \{1,2,4\}$. When reporting intersections immediately, we have $h \in Y^{k+1}_{\sigma_j}$, for $j \in \{1,4\}$ and $h \in R^k_{\sigma_2}$.

First we discuss how to produce the lists $Y^{k+1}_{\sigma_j}$ (for both variants) and $R^k_{\sigma_j}$ at non-leaf invocations, as this step is common to both solutions. Then we discuss each of the two distribution sweeping variants in detail.

4.1 Generating Lists $Y^{k+1}_{\sigma_j}$ and $R^k_{\sigma_j}$ for Non-Leaf Invocations

We process all invocations I^k_σ at the kth recursive level in parallel. Let $N_k := \sum_\sigma |Y^k_\sigma|$ and $P_\sigma := \lceil P|Y^k_\sigma|/N_k \rceil$. Since $N_k = \Omega(N)$, N_k can be computed using $O(N_k/PB)$ I/Os using a prefix sum operation.

Within each vertical slab σ, we define P_σ horizontal slabs, each containing $|Y^k_\sigma|/P_\sigma = N_k/P$ elements of Y^k_σ. The P_σ horizontal slabs and d vertical child slabs σ_j define a $P_\sigma \times d$ grid. We refer to the cell in row i and column j as C_{ij}. Our first step is to compute the number of vertical segments intersecting the horizontal boundaries between adjacent grid cells. Then we use this information to count, for each horizontal segment $h \in Y^k_\sigma$, the number of grid cells that h spans and where it has at least one intersection. Finally, we generate y-sorted lists Y_{ij} and R_{ij}, for each grid cell C_{ij}, which are the portions of $Y^{k+1}_{\sigma_j}$ and $R^k_{\sigma_j}$ containing elements from the ith horizontal slab. The lists $Y^{k+1}_{\sigma_j}$ and $R^k_{\sigma_j}$ are then obtained from the lists Y_{ij} and R_{ij}, respectively, using transpose and compact operations. Next we discuss these steps in detail.

1. Intersection counts for horizontal grid cell boundaries. Using global load balancing, we allocate $O(N_k/P)$ elements of each list Y^k_σ to a processor. This partition of Y^k_σ defines the P_σ horizontal slabs in σ's grid. The processor associated with the ith horizontal slab sequentially scans its assigned

portion of Y_σ and generates y-sorted lists V_{ij} of vertical segment endpoints in each cell C_{ij}. It also adds an entry representing the top boundary of the cell C_{ij} as the first element in each list V_{ij}. Using a transpose and compact operation, we obtain y-sorted lists V'_{σ_j} of vertical segment endpoints and cell boundaries in each of the d child slabs σ_j. Observing that $N_k = \Omega(N)$ and $d = O(N/PB)$ [2], the intersection counts for all cell boundaries in σ_j can now be computed using $O(N_k/PB)$ I/Os by treating these cell boundaries as stabbing queries over V'_{σ_j}. The total I/O complexity of this step is therefore $O(N_k/PB)$.

2. Counting cells with intersections for each horizontal segment. Each processor performs a vertical sweep of the portion of Y_σ^k assigned to it in Step 1. For each vertical slab σ_j, it keeps track of the number of vertical segments in σ_j that span the current y-coordinate, starting with the intersection count of the top boundary of C_{ij} and updating the count whenever the sweep passes a top or bottom endpoint of a vertical segment. When the sweep passes a horizontal segment h, this segment has an intersection in a cell C_{ij} spanned by h if and only if the count for slab σ_j is non-zero. By testing this condition for each cell, we can determine t'_h, the number of slabs σ_j spanned by h and where h has an intersection. We assign weights $w_h := 1 + t'_h$ and $w_q := 1$ to each horizontal segment h and vertical segment endpoint q. The I/O complexity of this step is $O(N_k/PB)$ I/Os because each processor scans N_k/P elements in this step and keeps $d \leq M$ counters in memory.

3. Generating child lists. Using a global load balancing operation with the weights computed in Step 2, we reallocate the elements in Y_σ^k to processors so that the elements assigned to each processor have total weight W_k/P, where $W_k = \sum_\sigma \sum_{e \in Y_\sigma^k} w_e$. This partitioning of Y_σ^k induces new horizontal slabs in σ's grid. We repeat Step 1 to count the number of vertical segments intersecting each horizontal cell boundary and repeat the sweep from Step 2, this time copying every horizontal segment with an endpoint in C_{ij} to Y_{ij} and, depending on the distribution sweeping variant, adding every horizontal segment spanning σ_j and with an intersection in σ_j to Y_{ij} or R_{ij}, and every vertical segment endpoint in σ_j to Y_{ij} and R_{ij}. Finally, we obtain the lists $Y_{\sigma_j}^{k+1}$ and $R_{\sigma_j}^k$ using a transpose and compact operation. The I/O complexity of this step is $O(\frac{W_k}{PB}) = O(\frac{N_k + L_k}{PB})$ I/Os, where $L_k = \sum_h t'_h$ with the sum taken over all horizontal segments $h \in Y_\sigma^k$.

By summing the costs of these three steps, we obtain the following lemma.

Lemma 1. *At the kth recursive level, the y-sorted lists $Y_{\sigma_j}^{k+1}$ and $R_{\sigma_j}^k$ can be generated using $O(\frac{N_k + L_k}{PB})$ I/Os, where $N_k = \sum_\sigma |Y_\sigma^k|$ and $L_k = \sum_h t'_h$ with the second sum taken over all horizontal segments in slab lists Y_σ^k.*

4.2 An $O(\mathrm{sort}_P(N + K))$ Solution

Our $O(\mathrm{sort}_P(N + K))$ I/O solution defers the reporting of intersections to the leaf invocations, ensuring that the input to each leaf invocation I_σ^k includes all segments with an endpoint in σ and all horizontal segments with an intersection in σ. We achieve this by setting $Y_{\sigma_j}^{k+1} := V_{\sigma_j}^k \cup E_{\sigma_j}^k \cup S_{\sigma_j}^k$, for each child slab σ_j of a non-leaf invocation I_σ^k. By Lemma 1, the input lists for level $k + 1$ can be

generated using $O(\frac{N_k+L_k}{PB}) = O(\frac{N+K}{PB})$ I/Os because $N_k \le N+K$ and $L_k \le K$. Since there are $\log_d P$ recursive levels, the cost of all non-leaf invocations is $O(\frac{N+K}{PB}\log_d P) = O(\mathrm{sort}_P(N+K))$ I/Os. At the leaf level, we balance the reporting of intersections among processors based on the number of intersections of each horizontal segment. The details are as follows.

1. Counting intersections. We partition each list Y_σ^k into y-sorted lists H_σ and V_σ of horizontal segments and vertical segment endpoints. This takes $O(N_k/PB)$ I/Os by copying each element of Y_σ^k into the corresponding position of H_σ or V_σ and compacting the two lists. Using global load balancing, we allocate $O(N_k/P) = O(\frac{N+K}{P})$ horizontal segments from $O(1)$ slabs to each processor. Applying sequential I/O-efficient orthogonal intersection counting [19] to its assigned horizontal segments and the vertical segments in the corresponding slabs, each processor computes t_h, the number of intersections of each of its horizontal segments h, and assigns weight $w_h := 1 + t_h$ to h. Since $|V_\sigma| = O(N/P)$, the cost of this step is $O(\mathrm{sort}_1(\frac{N+K}{P})) = O(\mathrm{sort}_P(N+K))$.

2. Reporting intersections. Using global load balancing with the weights computed in the previous step, we re-allocate horizontal segments to processors so that each processor is responsible for segments of total weight $W/P = (\sum_\sigma \sum_{h\in H_\sigma} w_h)/P = O(\frac{N+K}{P})$. Each processor runs a sequential I/O-efficient orthogonal line segment intersection reporting algorithm [19] on its horizontal segments and the vertical segments in the corresponding $O(1)$ slabs. This step takes $O(\mathrm{sort}_1(N/P + W/P)) = O(\mathrm{sort}_P(N+K))$ I/Os.

By summing the costs of all invocation, we obtain the following theorem.

Theorem 3. *In the PEM model, orthogonal line segment intersection reporting takes* $O(\mathrm{sort}_P(N+K))$ *I/Os, provided* $P \le \min\{\frac{N}{B\log N}, \frac{N}{B^2}\}$ *and* $M = B^{O(1)}$.

4.3 An $O(\mathrm{sort}_P(N)\log_d P + K/PB)$ Solution

In our $O(\mathrm{sort}_P(N)\log_d P + K/PB)$ solution, each invocation I_σ^k generates lists $Y_{\sigma_j}^{k+1} := V_{\sigma_j}^k \cup E_{\sigma_j}^k$ and $R_{\sigma_j}^k := V_{\sigma_j}^k \cup S_{\sigma_j}^k$, for each child slab σ_j of σ, and then reports all intersections between elements in $R_{\sigma_j}^k$ before recursing on each slab σ_j with input $Y_{\sigma_j}^{k+1}$. The leaf invocations are the same as in the $O(\mathrm{sort}_P(N+K))$ solution, and we process all invocations at each level of recursion simultaneously.

Generating all lists $Y_{\sigma_j}^{k+1}$ and $R_{\sigma_j}^k$ at the kth recursive level takes $O(\frac{N_k+L_k}{PB})$ I/Os; see Section 4.1. Since each list Y_σ^k contains only segments with an endpoint in σ, we have $N_k \le 2N$ and $\sum_k N_k = O(N\log_d P)$. Since we also have $\sum_k L_k \le K$, the cost of generating lists $Y_{\sigma_j}^{k+1}$ and $R_{\sigma_j}^k$ for all non-leaf invocations is $O((N/PB)\log_d P + K/PB)$, while the cost of all leaf invocations is $O(\mathrm{sort}_P(N) + K/PB)$ (each processor processes elements from only $O(1)$ slabs, and each slab contains only $O(N/P)$ vertical segments and horizontal segment endpoints). Next we discuss how to report all intersections between elements of the lists $R_{\sigma_j}^k$ at the kth recursive level using $O(\mathrm{sort}_P(N) + \frac{K_k + K/\log_d P}{PB})$ I/Os, where K_k is the number of intersections reported at the kth recursive level. This sums to a cost of $O(\mathrm{sort}_P(N)\log_d P + K/PB)$ I/Os for all non-leaf invocations and dominates the total cost of the algorithm. This proves the following result.

Theorem 4. *In the PEM model, orthogonal line segment intersection reporting takes* $O(\text{sort}_P(N)\log_d P + \frac{K}{PB})$ *I/Os, if* $P \le \min\{\frac{N}{B\log N}, \frac{N}{B^2}\}$ *and* $M = B^{O(1)}$.

To achieve a cost of $O(\text{sort}_P(N) + \frac{K_k + K/\log_d P}{PB})$ I/Os per recursive level, we assume every vertical segment has at most $K' := \max\{N/P, K/(P\log_d P)\}$ intersections. Below we sketch how to eliminate this assumption by splitting vertical segments with more than K' intersections into subsegments with at most K' intersections as needed.

To report the intersections at the kth recursive level, we process all lists $R^k_{\sigma_j}$ in parallel. We do this in three steps. First we count the number of intersections of each vertical segment in such a list. Then we split each list $R^k_{\sigma_j}$ into y-sorted lists V_{σ_j} and H_{σ_j} containing the top endpoints of vertical segments and horizontal segments, respectively. Each endpoint in V_{σ_j} also stores the bottom endpoint and the number of intersections of the corresponding segment. In the third step, we allocate portions of the lists V_{σ_j} to processors, and each processor reports the intersections of its allocated vertical segments. The details are as follows.

1. Counting intersections. Counting the number of intersections for each vertical segment in $R^k_{\sigma_j}$ is equivalent to answering 1-D range counting queries over $R^k_{\sigma_j}$, as each horizontal segment in $R^k_{\sigma_j}$ completely spans σ_j. Thus, by applying Theorem 1 to all lists $R^k_{\sigma_j}$ simultaneously, this step takes $O(\text{sort}_P(N) + K_k/PB)$ I/Os because there are $O(N)$ vertical segments and at most K_k horizontal segments in all lists $R^k_{\sigma_j}$ at the kth recursive level.

2. Generating lists $\boldsymbol{H_{\sigma_j}}$ and $\boldsymbol{V_{\sigma_j}}$. Splitting $R^k_{\sigma_j}$ into lists H_{σ_j} and V_{σ_j} can be done as the splitting of Y^k_σ for leaf invocations. Before doing this, however, we annotate every vertical segment endpoint q with the index $\text{scc}(q)$ such that $H_{\sigma_j}[\text{scc}(q)]$ is the first horizontal segment below q in the list H_{σ_j}. This is done by assigning a weight of 0 to vertical segment endpoints and 1 to horizontal segments and computing prefix sums on these weights. Thus, the I/O complexity of this step is $O(\frac{N+K_k}{PB})$.

3. Reporting intersections. Let t_q be the number of intersections of the vertical segment with top endpoint q, and $w_q := 1 + t_q$. We allocate portions of the lists V_{σ_j} to processors by using global load balancing with these weights. Since every vertical segment has at most K' intersections, this assigns segment endpoints with total weight $O(\frac{N+K_k}{P} + K')$ to each processor. The cost of this assignment step is $O(\frac{N+K_k}{PB})$ I/Os.

Now each processor performs a sequential sweep of its assigned portion V' of a list V_{σ_j} and of a portion H' of H_{σ_j}, starting with position $\text{scc}(q)$, where q is the first point in V_{σ_j}. The elements in V' and H' are processed by decreasing y-coordinates. When processing a segment endpoint in V', its vertical segment is inserted into an active list A. When processing a segment h in H', we scan A to report all intersections between h and vertical segments in A and remove all vertical segments from A that do not intersect h. The sweep terminates when all points in V' have been processed and A is empty.

The I/O complexity per processor p_i is easily seen to be $O(r_i + (W_i + Z_i)/B)$, where $r_i = O(1)$ is the number of portions of lists V_{σ_j} assigned to p_i, W_i is

the total weight of the elements in these portions, and Z_i is the total number of scanned elements in the corresponding lists H'. Our goal is to show that $Z_i = \mathrm{O}(W_i + K')$, which bounds the cost of reporting intersections by $\mathrm{O}(1 + \frac{N+K_k}{PB} + \frac{K'}{B}) = \mathrm{O}(\frac{N+K_k}{PB} + \frac{K'}{B})$. To this end, we show that there are only $\mathrm{O}(K')$ horizontal segments scanned by p_i that do not intersect any vertical segments assigned to p_i. Consider the last segment h in a portion H' of a list H_{σ_j} scanned by p_i and which does not intersect a segment in the corresponding sublist V' of V_{σ_j} assigned to p_i. Since every horizontal segment in H_{σ_j} has at least one intersection at this recursive level, h must intersect some vertical segment v assigned to another processor. Observe that the top endpoint of v must precede V' in V_{σ_j}, which implies that v intersects all segments in H' scanned by p_i but without intersections with segments in V'. Since v has at most K' intersections, there can be at most K' such segments in H', and p_i scans portions of only $\mathrm{O}(1)$ lists H_{σ_j}.

By adding the costs of the different steps, we obtain a cost of $\mathrm{O}(\mathrm{sort}_P(N) + (K_k + K/\log_d P)/PB)$ I/Os per recursive level, as claimed in Theorem 4.

Our algorithm relies on the assumption that every vertical segment has at most K' intersections in two places: balancing the reporting load among processors and bounding the number of elements in H_{σ_j}-lists scanned by each processor. After Step 1, the top endpoint q of each vertical segment in V_{σ_j} stores its intersection count t_q and the index $\mathrm{scc}(q)$ of the first segment in H_{σ_j} below q. For each endpoint q with $t_q > K'$, we generate $l_q := \lceil t_q/K' \rceil$ copies $q_1, q_2, \ldots, q_{l_q}$, each with an intersection count of K'—q_{l_q} has intersection count $t_q \bmod K'$—and successor index $\mathrm{scc}(q_i) := \mathrm{scc}(q) + (i-1)K'$. We sort the resulting augmented V_{σ_j}-list by the successor indices of its entries and modify the reporting step to remove a vertical segment from the active list when a number of intersections matching its intersection count have been reported. This is equivalent to splitting each vertical segment with more than K' intersections at the current recursive level into subsegments with at most K' intersections each. The full version of this paper will show that the number of elements in the V_{σ_j}-lists at each recursive level remains $\mathrm{O}(N)$ and that this does not alter the cost of the algorithm.

5 Additional Problems

Theorem 5. *The lower envelope of a set of non-intersecting 2-D line segments, the convex hull of a 2-D point set, and the maxima of a 3-D point set can be computed using* $\mathrm{O}(\mathrm{sort}_P(N))$ *I/Os, provided* $P \leq N/B^2$ *and* $M = B^{\mathrm{O}(1)}$.

Proof (Sketch). The lower envelope of a set of non-intersecting line segments and the maxima of a 3-D point set can be computed by merging point lists sorted along one of the coordinate axes and computing appropriate labels of the points in each list $U(v)$ from the labels of their predecessors in v's child lists using the same strategy as for 2-D weighted dominance counting [6]. The result on convex hull is obtained using a careful analysis (whose details will be presented in the full paper) of an adaptation of the CREW PRAM algorithm of [7]. □

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Communications of the ACM 31(9), 1116–1127 (1988)
2. Arge, L., Goodrich, M.T., Nelson, M.J., Sitchinava, N.: Fundamental parallel algorithms for private-cache chip multiprocessors. In: SPAA. pp. 197–206 (2008)
3. Arge, L., Goodrich, M.T., Sitchinava, N.: Parallel external memory graph algorithms. In: IPDPS (2010), to appear
4. Arge, L., Mølhave, T., Zeh, N.: Cache-oblivious red-blue line segment intersection. In: ESA. pp. 88–99 (2008)
5. Atallah, M.J., Cole, R., Goodrich, M.T.: Cascading divide-and-conquer: A technique for designing parallel algorithms. SIAM J. Comp. 18(3), 499–532 (1989)
6. Atallah, M.J., Goodrich, M.T.: Efficient plane sweeping in parallel. In: SoCG. pp. 216–225 (1986)
7. Atallah, M.J., Goodrich, M.T.: Parallel algorithms for some functions of two convex polygons. Algorithmica 3, 535–548 (1988)
8. Bender, M.A., Fineman, J.T., Gilbert, S., Kuszmaul, B.C.: Concurrent cache-oblivious B-trees. In: SPAA. pp. 228–237 (2005)
9. Blelloch, G.E., Chowdhury, R.A., Gibbons, P.B., Ramachandran, V., Chen, S., Kozuch, M.: Provably good multicore cache performance for divide-and-conquer algorithms. In: SODA. pp. 501–510 (2008)
10. Brodal, G.S., Fagerberg, R.: Cache oblivious distribution sweeping. In: Widmayer, P., Ruiz, F.T., Bueno, R.M., Hennessy, M., Eidenbenz, S., Conejo, R. (eds.) ICALP. LNCS, vol. 2380, pp. 426–438. Springer-Verlag (2002)
11. Chowdhury, R.A., Ramachandran, V.: The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. In: SPAA. pp. 71–80 (2007)
12. Chowdhury, R.A., Ramachandran, V.: Cache-efficient dynamic programming for multicores. In: SPAA. pp. 207–216 (2008)
13. Chowdhury, R.A., Silvestri, F., Blakeley, B., Ramachandran, V.: Oblivious algorithms for multicores and network of processors. In: IPDPS (2010), to appear
14. Datta, A.: Efficient parallel algorithms for geometric partitioning problems through parallel range searching. In: ICPP pp. 202–209 (1994)
15. Dehne, F., Fabri, A., Rau-Chaplin, A.: Scalable parallel geometric algorithms for coarse grained multicomputers. In: SoCG. pp. 298–307 (1993)
16. Fjällström, P.O.: Parallel algorithms for batched range searching on coarse-grained multicomputers. Linköping Electronic Articles in Computer and Information Science 2(3) (1997)
17. Gibbons, P.: Theory: Asleep at the switch to many-core. Workshop on Theory and Many-Cores (T&MC) (May 2009)
18. Goodrich, M.T.: Intersecting line segments in parallel with an output-sensitive number of processors. SIAM J. Comp. 20(4), 737–755 (1991)
19. Goodrich, M.T., Tsay, J.J., Vengroff, D.E., Vitter, J.S.: External-memory computational geometry. In: FOCS. pp. 714–723 (1993)
20. 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)
21. Sitchinava, N.: Parallel external memory model – a parallel model for multi-core architectures. Ph.D. thesis, University of California, Irvine (2009)