Parallel Range, Segment and Rectangle Queries with Augmented Maps

Yihan Sun
Carnegie Mellon University
yihans@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Abstract

The support of range, segment and rectangle queries are fundamental problems in computational geometry, and have extensive applications in many domains. Despite significant theoretical work on these problems, efficient implementations can be complicated, and most implementations do not have useful theoretical bounds. In this paper, we focus on simple and efficient parallel algorithms and implementations for range, segment and rectangle queries, which have worst-case bounds in theory and good performance in practice, both sequentially and in parallel. We propose to use a framework based on the abstract data type *augmented map*, to model the problems. Based on the augmented map interface, we develop both multi-level tree structures and sweepline algorithms supporting range, segment and rectangle queries in two dimensions. For the sweepline algorithms, we also propose a parallel paradigm and show corresponding cost bounds. Theoretically, the construction algorithms of all of our data structures are work-efficient and highly parallelized.

We have implemented all the data structures described in the paper, ten in all, using a parallel augmented map library. Based on the library, each data structure only requires about 100 lines of C++ code. We test their performance on large data sets (up to 10^8 elements) and a machine with 72-cores (144 hyperthreads). The parallel construction achieves 32-68x speedup, and the speedup numbers for queries are up to 126-fold. Sequentially, each of our implementations outperforms the CGAL library by at least 2x in both construction and queries. Our sequential implementation has approximately the same construction time as the R-tree in the Boost library, but has significantly better query performance (1.6-1200x). We believe this paper provides the most comprehensive experimental study of data structures on range, segment and rectangle queries, both in parallel and sequential setting.

1 Introduction

Range, segment and rectangle queries are fundamental problems in computational geometry, with extensive applications in many domains. In this paper, we focus on 2D Euclidean space. The range query problem is to maintain a set of points, and to answer queries regarding the points contained in a query rectangle. The segment query problem is to maintain a set of non-intersecting segments, and to answer questions regarding all segments intersected with a query vertical line. The rectangle stabbing query (also referred to as the *enclosure stabbing query*) problem is to maintain a set of rectangles, and to answer questions regarding rectangles containing a query point. For all problems, we discuss queries of both listing all queried elements (the *list-all* query), and returning the count of queried elements (the *counting* query). Some other queries, can be implemented by variants (e.g., the weighted sum of all queried elements) or combinations (e.g. rectangle-rectangle intersection queries) of the queries in this paper. Efficient solutions to these problems are mostly based on variants of range trees [19], segment trees [16], sweepline algorithms [59], or combinations of them.

In addition to the large body of work on sequential algorithms and data structures [18, 20, 29, 32, 37, 38], there have also been many theoretical results on parallel algorithms and structures for such queries [4, 11, 13, 40]. However, efficient implementations of these structures can be complicated. We know of few parallel implementations of these theoretically efficient query structures, primarily due to delicate design of algorithmic details required by the structures. The parallel implementations we know of [28, 43, 46, 50] do not have useful theoretical bounds. Our goal is to develop theoretically efficient algorithms which can be implemented with ease and also run fast in practice, especially *in parallel*.

The algorithms and implementations in this paper use many ideas from the sequential and parallel algorithms mentioned above, but using a framework based on *augmented maps* [63]—an abstract data type (ADT). The augmented map is an abstract data type (ADT) based on ordered maps of key-value pairs, augmented with an abstract "sum" called the *augmented value* (see Section 3). We use augmented maps to help develop efficient and concise implementations. To do this, there are two major steps: 1) modeling problems using augmented maps, and 2) implement augmented maps using efficient data structures. Indeed, in this paper, as shown in Section 6, 7 and 8, we model the range, segment and rectangle query problems all as two-level map structures: an outer level map augmented with an inner map structure. We show that the augment map abstraction is extendable to a

wide range of problems, and develop five structures on top of the augmented map interface corresponding to different problems and queries.

As for implementing augmented maps, we employ two data structures: the augmented tree structures as studied in previous work, and *prefix structures*, which are proposed by this paper. We propose simple and parallel implementations for the prefix structures and analyze cost bounds. Interestingly, the algorithms based on the prefix structures resemble the standard sweepline algorithms. Therefore, our algorithm also parallelizes a family of sweepline algorithms that are efficient both in theory and practice. As a result, both augmented trees and prefix structures provide efficient parallel implementations for augmented maps, and each has its own merits in different settings.

By combining the five two-level map structures with the two underlying data structures as the outer map (the inner maps are always implemented by augmented trees), we develop a total of ten different data structures for range, segment and rectangle queries. Among the ten data structures, five of them are multi-level trees including *RangeTree* (for range query), *SegTree* (for segment query), *RecTree* (for rectangle query), and another two for fast counting queries *SegTree** (segment counting) and *RecTree** (rectangle counting). The other five are the corresponding sweepline algorithms.

All the data structures in this paper are efficient in theory. We summarize the theoretical costs in Table 1. The construction bounds are all optimal in work (lower bounded by sorting), and the query time is almost-linear in the output size. We did not use fractional cascading [31], so some of our query bounds are not optimal. However, we note that they are sub-optimal by at most a $\log n$ factor.

All the data structures in this paper are also fast in practice. We implement all of them making use of a parallel augmented map library *PAM* [63], which supports augmented maps using augmented trees. We compare our implementations to C++ libraries CGAL [64] and Boost [1]. We achieve a 33-to-68-fold self-speedup in construction on 72 cores (144 hyperthreads), and 60-to-126-fold speedup in queries. Our sequential construction is more than 2x faster than CGAL, and is comparable to Boost. Our query time outperforms both CGAL and Boost by 1.6-1400x. We also provide a thorough comparison among the new algorithms in this paper, leading to many interesting findings.

Beyond being fast, our implementation is also concise and simple. Using the augmented map abstraction greatly simplifies engineering and reduces the coding effort, which is indicated by the required lines of code—on top of PAM, each application only requires about 100 lines of C++ code even for the parallel version. For the same functionality, both CGAL and Boost use hundreds of lines of code for each sequential implementation. We note that PAM implements general-purpose augmented maps, and does not directly provide anything special for computational geometry. Due to page limitation, some content are in the full version of this paper [62]. Our code is available at `https://github.com/cmuparlay/PAM`.

		Build		Query	
		Work	**Depth**	**List-all**	**Count**
Range	**Swp.**	$n \log n$	n^{ϵ}	$\log n + k \log\left(\frac{n}{k} + 1\right)$	$\log n$
Query	**Tree**	$n \log n$	$\log^3 n$	$\log^2 n + k$	$\log^2 n$
Seg	**Swp.**	$n \log n$	n^{ϵ}	$\log n + k$	$\log n$
Query	**Tree**	$n \log n$	$\log^3 n$	$\log^2 n + k$	$\log^2 n$
Rec	**Swp.**	$n \log n$	n^{ϵ}	$\log n + k \log\left(\frac{n}{k} + 1\right)$	$\log n$
Query	**Tree**	$n \log n$	$\log^3 n$	$\log^2 n + k \log\left(\frac{n}{k} + 1\right)$	$\log^2 n$

Table 1: **Theoretical costs of all problems in this paper (asymptotical in Big-O notation)** - n is the input size, k the output size. $\epsilon < 1$ can be any given constant. "Swp." means the sweepline algorithms, "Tree" the two-level trees. We note that not all query bounds are optimal, but they are off optimal by at most a $\log n$ factor.

2 Related Work

Many data structures are designed for solving range, segment and rectangle queries, such as range trees [18], segment trees [20], kd-trees [17], R-trees [15, 55, 58], priority trees [49], and many others [24, 34, 35, 53, 57, 65]. They are then applied to a variety of other problems [3, 5, 6, 8, 9, 19, 21, 22, 27, 41, 44, 54].

There have been fundamental sequential data structures for such queries. The classic range tree [18] has construction time $O(n \log n)$ and query time $O(k + \log^2 n)$ for input size n and output size k. Using fractional cascading [31], the query time can be $O(k + \log n)$. We did not employ it for the simplicity and extensibility in engineering, and we believe the performance is still efficient and competitive. The terminology "segment tree" [20, 29] refers to different data structures in the literature, but the common insight is to use the tree structure to partition the interval hierarchically. Previous solutions for rectangle queries usually use combinations of range trees, segment trees, interval trees, and priority trees [20, 32, 37, 38]. Some other prior work focused on developing sequential sweepline algorithms for range queries [7, 10], segment intersecting [30, 51] and rectangle queries [48]. There are also sequential I/O-efficient algorithms for computational geometry [8, 9, 33] and sequential libraries support such queries [1, 64].

In the parallel setting, there exist many theoretical results [2, 4, 11, 12, 14, 39, 40]. Atallah et al. [14] proposed the *array-of-trees* using persistent binary search trees (BST) to store all intermediate trees in a sweepline algorithm by storing all versions of a tree node in a super-node. Our method also uses persistent BSTs, but uses path-copying to maintain a set of trees independently instead of in one skeleton. Recently, Afshani et al. [2] implemented the array-of-trees for the 1D total visibility-index problem. Atallah et al. [11]

discussed a cascading divide-and-conquer scheme for solving computational geometry problems in parallel. Goodrich et al. [40] proposed a framework to parallelize several sweepline-based algorithms. There has been previous work focusing on parallelizing segment-tree-like data structures [4, 13], and our segment tree algorithm is inspired by them. There are also theoretical I/O efficient algorithms in parallel [6, 60]. We know of no implementations or experimental evaluations of these theoretically efficient algorithms on range, segment and rectangle queries. There are also parallel implementation-based works such as parallel R-trees [46], parallel sweepline algorithms [50], and algorithms focusing on distributed systems [67] and GPUs [66]. No theoretical guarantees are provided in these papers.

3 Preliminaries

Notation. We call a key-value pair in a map an *entry* denoted as $e = (k, v)$. We use $k(e)$ and $v(e)$ to extract the key and the value from an entry. Let $\langle P \rangle$ be a sequence of elements of type P. For a tree node u, we use $k(u)$, $l(u)$ and $r(u)$ to extract its key, left child and right child respectively.

In two dimensions, let X, Y and $D = X \times Y$ be the types of x- and y-coordinates and the type of points, where X and Y are two sets with total ordering defined by $<_X$ and $<_Y$ respectively. For a point $p \in D$ in two dimensions, we use $x(p) \in X$ and $y(p) \in Y$ to extract its x- and y-coordinates, and use a pair $(x(p), y(p))$ to denote p. For simplicity, we assume all input coordinates are unique. Duplicates can be resolved by slight variations of algorithms in this paper.

Parallel Cost Model. To analyze asymptotic costs of a parallel algorithm we use *work* W and *depth* D (or *span* S), where work is the total number of operations and depth is the length of the critical path. The parallel time T can be bounded as $T \leq \frac{W}{P} + D$ assuming a PRAM model [45] with P processors and a greedy scheduler [25, 26, 42]. We assume concurrent reads and exclusive writes (CREW).

Persistence. A *persistent* data structure [36] is a data structure that preserves previous versions of itself. When being modified, it always creates a new updated structure. For BSTs, persistence can be achieved by path-copying [56]. In path-copying, only the affected path related to the update is copied, so the asymptotical cost for an update remains unchanged. In this paper, we assume all underlying tree structures are persistent. In experiments, we use a library PAM supporting persistence [63].

Sweepline Algorithms. A sweepline algorithm (or plane sweep algorithm) is an algorithmic paradigm that uses a conceptual sweep line to process elements in order [59]. It uses a virtual line sweeping across the plane, which stops at some points (e.g., the endpoints of segments) to make updates. We refer to the points as the *event points* $p_i \in P$. They are processed in a total order defined by $\prec$: $P \times P \mapsto$ **BOOL**. Here we assume the events are known ahead of time. As the algorithm processes the points, a data structure T is maintained and updated at each event point to track the status at that point. Sarnak and Tarjan [56] first noticed that by being persistent, one can keep the intermediate structures $t_i \in T$ at all event points for later queries. In this paper, we adopt the same methodology, but parallelize it. We call t_i the *prefix structure* at the point i.

Function	*Work*	*Depth*
INSERT(m, e), **DELETE**(m, k)	$\log n$	$\log n$
INTERSECT(m_1, m_2) **DIFFERENCE**(m_1, m_2) **UNION**(m_1, m_2, σ)	$n_1 \log\left(\frac{n_1}{n_2} + 1\right)$	$\log n_1 \log n_2$
BUILD(s)	$n \log n$	$\log n$
UPTO(m, k), **RANGE**(m, k_1, k_2)	$\log n$	$\log n$
ALEFT(m, k), **ARANGE**(m, k_1, k_2)	$\log n$	$\log n$

Table 2: **Core functions on the augmented map interface** - $k, k_1, k_2 \in K$. m, m_1, m_2 are maps, $n = |m|$, $n_i = |m_i|$. s is a sequence. All bounds are in big-O notation. The bounds assume the augmenting functions f, g have constant cost.

Typically in sweepline algorithms, on encountering an event point p_i we compute t_i from the previous structure t_{i-1} and the new point p_i using an *update function* $h : T \times P \mapsto T$, i.e., $t_i = h(t_{i-1}, p_i)$. The initial structure is t_0. A sweepline algorithm can therefore be defined as the five tuple:

$$\boldsymbol{S} = \mathbf{SW}(P, \prec, T, t_0, h) \tag{3.1}$$

It defines a function that takes a set of points $p_i \in P$ and returns a mapping from each point to a prefix structure $t_i \in T$.

The Augmented Map. The *augmented map* [63] is an abstract data type (ADT) that associates an ordered *map* (a set of key-value pairs) with a "map-reduce" operation for keeping track of the abstract sum over entries (referred to as the *augmented value* of the map). More formally, an augmented map is an ordered map M where keys belong to some ordered set K (with total ordering defined by relation $<_K$) and values to a set V, that is associated with two functions: the *base function* $g : K \times V \mapsto A$ that maps a key-value pair to an augmented value (from a set A) and the *combine function* $f : A \times A \mapsto A$ that combines (reduces) two augmented values. We require f to be associative and have an identity I (i.e., set A with f and I forms a monoid). An augmented map can therefore be defined as the seven tuple:

$$a = \mathbf{AM}(K, <_K, V, A, g, f, I) \tag{3.2}$$

Then the augmented value of a map $M = ((k_1, v_1), \ldots, (k_n, v_n))$, denoted as $\mathcal{A}(M)$, is defined as $\mathcal{A}(M) = f(g(k_1, v_1), g(k_2, v_2), \ldots, g(k_n, v_n))$, where the definition of the binary function f is extended as:

$$f(\emptyset) = I, f(a_1) = a_1,$$
$$f(a_1, a_2, \ldots, a_n) = f(f(a_1, a_2, \ldots a_{n-1}), a_n)$$

A list of common functions on augmented maps used in this paper, and their parameters, are shown in Table 2 (the first column). Functions that are useful in this paper include:

the **ARANGE** function which returns the augmented value of all entries within a key range, and **ALEFT**(k) which returns the augmented value of all entries up to a key k. More details can be found in our previous paper [63].

Augmented Maps Using Augmented Trees. An efficient implementation of augmented maps is to use augmented balanced binary search trees [63]. Entries are stored in tree nodes and sorted by keys. Each node also maintains the *augmented value* of the subtree rooted at it. Using join-based algorithms [23, 63] on trees, the augmented map interface can be supported in an efficient and highly-parallelized manner, and the costs are listed in Table 2 assume f and g have a constant cost. All functions listed in Table 2 have optimal work and polylog depth. In the experiment we use the parallel library PAM [63] which implements augmented maps using augmented trees. The cost of functions in the library matches the bounds in Table 2.

4 A Parallel Sweepline Paradigm

We present a parallel algorithm to build the prefix structures, assuming the update function h can be applied "associatively" to the prefix structures. We assume $h(t, p) \equiv f_h(t, g_h(p))$ for some associative function $f_h : T \times T \mapsto T$ and $g_h : P \mapsto T$. Similarly as in an augmented map, we call f_h and g_h the *base* and *combine* function of the corresponding update function, respectively. Because of the associativity of h_p, to repeatedly update a sequence of points $\langle p_i \rangle$ onto a "prefix sum" t using h is equivalent to combining the "partial sum" of all points $\langle p_i \rangle$ to t using the combine function f_h. Thus, to build all the prefix structures in parallel, our approach is to evenly split the input sequence of points into b blocks, calculate the partial sum of each block, and refine the prefix structures in each block using the update function h. For simplicity we assume n is an integral multiple of b and $n = b \times r$. We define a *fold function* $\rho : \langle P \rangle \mapsto T$ that converts a sequence of points into a prefix structure, which gives the "partial sums" of the blocks. We note that the fold function can be directly computed using g_h and f_h, but in many applications, a much simpler algorithm can be used. The parallel sweepline paradigm can therefore be defined as the six tuple:

$$\mathbf{S}' = \mathbf{PS}(\boldsymbol{P}; \prec; \boldsymbol{T}; \boldsymbol{t_0}; \boldsymbol{h}; \boldsymbol{\rho}; \boldsymbol{f_h}) \tag{4.3}$$

We do not include g_h in the formalization since it is never used separately. Our parallel algorithm to build the prefix structures is as follows (also see Algorithm 1):

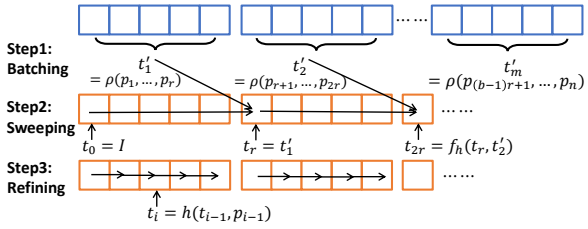
1. **Batching**. Assume all input points have been sorted by $\prec$. We evenly split them into b blocks and then in parallel generate b partial sums $t'_i \in T$ using ρ, each corresponding to one of the b blocks.
2. **Sweeping**. These partial sums t'_i are combined in turn sequentially by the combine function f_h to get the first prefix structure $t_0, t_r, t_{2r} \ldots$ in each block using $t_{i\times r} = f(t_{(i-1)\times r}, t'_i)$.

Algorithm 1: The construction of the prefix structure.

Input: A list p storing n input points in order, the update function h, the fold function ρ, the combine function f_h, the empty prefix structure t_0, and the number of blocks b. Assume $r = n/b$ is an integer.

Output: A series of prefix-combine trees t_i.

[Step 1.] parallel-for $i \leftarrow 0$ *to* $b-1$ **do**
  $t'_i = \rho(p_{i\times r}, \ldots, p_{(i+1)\times r-1})$
[Step 2.] for $i \leftarrow 1$ *to* $b-1$ **do** $t_{i\times r} = f_h(t_{(i-1)\times r}, t'_{i-1})$
[Step 3.] parallel-for $i \leftarrow 0$ *to* $b-1$ **do**
  $s = i \times r, e = s + r - 1$
  for $j \leftarrow s$ *to* e **do** $t_j = h(t_{j-1}, p_j)$
return $\{p_i \mapsto t_i\}$



3. **Refining**. All other prefix structures are built based on $t_0, t_r, t_{2r}, \ldots$ (built in the second step) in the corresponding blocks. All the b blocks can be built in parallel. In each block, the prefix structure t_i is computed sequentially in turn by applying h on p_i and t_{i-1}.

Algorithm 1 is straight-forward, yielding a simple implementation. Our experiments show that it also has good performance in parallel. However, we note that the parallelism is non-trivial. The challenge is in the sweeping step where f_h is applied sequentially for b times. Without new insights, this step can be as expensive as the original sequential sweepline algorithm, which is the reason that similar attempts previously did not achieve useful depth bounds [33, 50]. We observed that in many instantiations of this framework (especially those in this paper), a parallel combine function can be applied, which effectively guarantees the work-efficiency and parallelism of this algorithm. We formalize the typical setting as follows.

In a common setting of sweepline algorithms, each prefix structure keeps an ordered set tracking some elements related to a subset of the processed event points, having size $O(i)$ at point i. The function h updates one element to the set at a time, which costs $O(\log n)$ on a prefix structure of size n. The corresponding combine function f_h is some set functions (e.g., a **UNION** for a sequence of insertions). Using trees to maintain the sets, previous work shows [23] that the set function can be done in $O(n_2 \log(n_1/n_2 + 1))$ work and $O(\log n_1 \log n_2)$ depth for two set of size n_1 and $n_2 \leq n_1$. This algorithm is implemented in the PAM library (as shown in Table 2). Creating the partial sum of a block of r points

	The update function ($h(t,p)$)	The fold function ($\rho(s)$)	The combine function ($f(t,t')$)
Work	$\log \lvert t\rvert$	$\lvert s\rvert \log \lvert s\rvert$	$\lvert t'\rvert \log(\frac{\lvert t\rvert}{\lvert t'\rvert}+1)$
Depth	$\log \lvert t\rvert$	$\log \lvert s\rvert$	$\log \lvert t\rvert \log \lvert t'\rvert$
Output	$\lvert t\rvert$	$\lvert s\rvert$	$\lvert t'\rvert + \lvert t\rvert$

Table 3: **A typical setting of the function costs in a sweepline paradigm**. Bounds are in Big-O notation.

costs $O(r \log r)$ work and $O(\log r)$ depth, building a prefix structure of size at most r. We summarize the setting in Table 3, and the corresponding bounds of the sweepline algorithm in Theorem 4.1.

THEOREM 4.1. *A sweepline paradigm S as in Equation 5.5 can be built in parallel using its corresponding parallel paradigm S' (Equation 4.3). If the bounds as shown in Table 3 hold, then Algorithm 1 can construct all prefix structures in work $O(n \log n)$ and depth $O(\sqrt{n}\log^{1.5} n)$, where n is the number of event points.*

Proof. The cost of the algorithm is analyzed as follows:

1. **Batching**. This step builds b prefix structures, each of size at most n/b, so it takes work $O(b \cdot \frac{n}{b}\log\frac{n}{b}) = O(n\log\frac{n}{b})$ and depth $O(\log\frac{n}{b})$.
2. **Sweeping**. This step invokes b times of the combine function f_h sequentially, but the combine function works in parallel. The size of t'_i is no more than $O(r)$. Considering the given work and depth bounds of the combine function, the total work of this step is bounded by: $O\Big(\sum_{i=1}^{b} r\log(\frac{ir}{r}+1)\Big) = O(n\log b)$. The depth is: $O\Big(\sum_{i=1}^{b}\log r\log ir\Big) = O(b\log\frac{n}{b}\log n)$.
3. **Refining**. This step computes all the other prefix structures using h. The total work is: $O(\sum_{i=1}^{n}\log i) = O(n\log n)$. We process each block in parallel, so the depth is $O(\frac{n}{b}\log n)$.

In total, it costs work $O(n\log n)$ and depth $O\big(\big(b\log\frac{n}{b}+\frac{n}{b}\big)\log n\big)$. When $b = \Theta(\sqrt{n/\log n})$, the depth is $O(\sqrt{n}\log^{1.5} n)$.

This parallel algorithm is also easy to implement. Our code is available online at `https://github.com/cmuparlay/PAM/blob/master/common/sweep.h`. Theoretically, by repeatedly applying this process to each block in the last step, we can further reduce the depth to $O(n^{-\epsilon})$ for any constant $\epsilon > 0$. We state the following corollary and show the proof in Appendix D.

COROLLARY 4.1. *A sweepline paradigm S as in Equation 5.5 can be parallelized using its corresponding parallel paradigm S' (Equation 4.3). If the bounds as shown in Table 3 hold, then we can construct all prefix structures in work $O(\frac{1}{\epsilon}n\log n)$ and depth $\tilde{O}(n^{\epsilon})$ for arbitrary small $\epsilon > 0$.*

5 Augmented Maps Using Trees and Prefix Structures

With all preliminaries and techniques proposed above, we now discuss efficient parallel implementations of augmented maps. One solution is to use *augmented trees*, as studied in previous work [63] (summarized in Section 3). In this section, we further show how to use the *prefix structures* to implement parallel augmented maps.

For an augmented map $m = \{e_1, \ldots, e_{|m|}\}$, the prefix structures store the augmented values of all prefixes up to each entry e_i, i.e., $\mathbf{ALEFT}(m, k(e_i))$. For example, if the augmented value is the sum of values, the prefix structures are prefix sums. This is equivalent to using a combination of function f and g as the update function. That is to say, an augmented map $m = \mathbf{AM}(K, \prec, V, A, f, g, I)$ can be implemented by a sweepline paradigm S as:

$$S = \mathbf{SW}(K \times V; \prec; A; t_0 \equiv I; h(t,p) \equiv f(t, g(p))) \tag{5.4}$$

Because the combine function f in augmented maps is always associative, S trivially fits the parallel sweepline paradigm as defined in Equation 4.3 as follows:

$$S' = \mathbf{PS}(K \times V; \prec; A; t_0 \equiv I; h(t,p) \equiv f(t, g(p)); \\ \rho(p_1, \ldots p_n) \equiv f(g(p_1), \ldots g(p_n); f_h \equiv f) \tag{5.5}$$

This means we can apply Algorithm 1 to construct such an augmented map in parallel.

Prefix structures are especially useful for queries related to $\mathbf{ALEFT}$, as is the case for many queries in this paper. When using the prefix structures to represent the outer map in range, segment and rectangle queries, the algorithms are equivalent to sweepline algorithms, and they all accord with the assumption on the function cost in Theorem 4.1.

Now we have presented parallel implementations of augmented maps using two data structures. In the rest of the paper, we show how we model the range, segment and rectangle queries by augmented maps. Then by plugging in different representations, we show solutions to the problems using both augmented trees and prefix structures.

6 2D Range Query

Given a set of n points in the 2D plane, a *range query* asks some information of points within the intersection of a horizontal range (x_L, x_R) and vertical range (y_L, y_R).

The 2D range query can be answered using a two-level map structure *RangeMap*, each level corresponding to one dimension of the coordinates. The structure can answer both counting queries and list-all queries. The definition (the outer map R_M with inner map R_I) and an illustration are shown in Table 4 and Figure 1 (a). In particular, the key of the outer map is the coordinate of each point and the value is the count. The augmented value of such an outer map, which is the inner map, contains the same set of points, but are sorted by y-coordinates. Therefore, the base function of the outer map is just a singleton on the point and the combine function is

*** Range Query:**

(Inner Map)R_I= AM (K: D; $\prec$: $<_Y$; V: $\mathbb{Z}$; A: $\mathbb{Z}$; g: $(k,v) \mapsto 1$; f: $+_{\mathbb{Z}}$; I: 0)

-RangeMap R_M = AM (K: D; $\prec$: $<_X$; V: $\mathbb{Z}$; A: R_I; g: R_I.singleton; f: R_I.union; I: $\emptyset$)

- RangeSwp R_S = PS (P: D; $\prec$: $<_X$; T: R_I; t_0: $\emptyset$ h: R_I.insert ρ: R_I.build; f: R_I.union)

*** Segment Query:**

(Inner Map)S_I= AM (K: $D \times D$; $\prec$: $<_Y$; V: $\emptyset$; A: $\mathbb{Z}$; g: $(k,v) \mapsto 1$; f: $+_{\mathbb{Z}}$; I: 0)

- SegMap S_M = AM (K: X; $\prec$: $<_X$; V: $D \times D$; A: $S_I \times S_I$; g: g_{seg} f: f_{seg} I: $(\emptyset, \emptyset)$)

$$g_{\text{seg}}(k,(p_l,p_r)): \begin{cases} (\emptyset, S_I.\text{singleton}(p_l,p_r), \text{when } k = x(p_l)) \\ (S_I.\text{singleton}(p_l,p_r), \emptyset), \text{when } k = x(p_r) \end{cases}, f_{\text{seg}}: \text{See Equation 7.7;}$$

- SegSwp S_S = PS (P: $D \times D$; $\prec$: $<_X$; T: S_I; t_0: $\emptyset$; h: h_{seg}; ρ: ρ_{seg}; f: f_{seg})

$$h_{\text{seg}}(t,p) = \begin{cases} S_I.\text{insert}(t,p), \text{when } p \text{ is a left endpoint} \\ S_I.\text{delete}(t,p), \text{when } p \text{ is a right endpoint} \end{cases}, \rho_{\text{seg}}(\langle p_i \rangle) = \langle L, R \rangle \begin{cases} L \in S_I\text{: segments with } \textit{right} \text{ endpoint in } \langle p_i \rangle \\ R \in S_I\text{: segments with } \textit{left} \text{ endpoint in } \langle p_i \rangle \end{cases}$$

*** Rectangle Query:**

(Inner Map)G_I= AM (K: Y; $\prec$: $<_Y$; V: $D \times D$; A: Y; g: $(k,(p_l,p_r)) \mapsto y(p_r)$; f: $\max_Y$; I: $-\infty$)

- RecMap G_M = AM (K: X; $\prec$: $<_X$; V: $D \times D$; A: $G_I \times G_I$; g: g_{rec} f: f_{rec} I: $(\emptyset, \emptyset)$)

- RecSwp G_S = PS (P: $D \times D$; $\prec$: $<_X$; T: G_I; t_0: $\emptyset$; h: h_{rec}; ρ: ρ_{rec}; f: f_{seg})

g_{rec}, f_{rec}, h_{rec} and ρ_{rec} are defined similarly as g_{seg}, f_{seg}, h_{seg} and ρ_{seg}

Table 4: **Definitions of all structures** - Although this table seems complicated, we note that it *fully* defines all data structures in this paper using augmented maps. X and Y are types of x- and y-coordinates. $D = X \times Y$ is the type of a point.

UNION. The augmented value of the inner map counts the number of points in this (sub-)map. Then the construction of a sequence s of points can be done with the augmented map interface as: $r_M = R_M.\textbf{BUILD}(s)$.

To answer queries, we use two nested range searches (x_L, x_R) on the outer map and (y_L, y_R) on the corresponding inner map. The counting query can be represented using the augmented map interface as:

$$\textbf{RANGEQUERY}(r_M, x_L, x_R, y_L, y_R) =$$
$$R_I.\textbf{ARANGE}(R_M.\textbf{ARANGE}(r_M, x_L, x_R), y_L, y_R) \tag{6.6}$$

The list-all query can be answered similarly using $R_I.\textbf{RANGE}$ instead of $R_I.\textbf{ARANGE}$.

In this paper, we use augmented trees for inner maps. We discuss two implementations of the outer map: the augmented tree, which yields a range-tree-like data structure, and the prefix structures, which yields a sweepline algorithm.

6.1 2D Range Tree. If the outer map is supported using the augmented tree structure, the *RangeMap* becomes a range tree (*RangeTree*). In this case we do not explicitly build $R_M.\textbf{ARANGE}(r_M, x_L, x_R)$ in queries. Instead, as the standard range tree query algorithm, we search the x-range on the outer tree, and conduct the y-range queries on the related inner trees. This operation is supported by the function **APROJECT** in the augmented map interface and the PAM library. Such a tree structure can be constructed within work $O(n \log n)$ and depth $O(\log^3 n)$ (theoretically the depth can be easily reduced to $O(\log^2 n)$, but in the experiments we use the $O(\log^3 n)$ version to make fewer copies of data). It answers the counting query in $O(\log^2 n)$ time, and report all queried points in $O(k + \log^2 n)$ time for output size k. A similar discussion of range trees is shown in [63]. In this paper, we further discuss efficient updates on *RangeTree* using the augmented map interface in Appendix E.

6.2 The Sweepline Algorithm. We now present a parallel sweepline algorithm *RangeSwp* for 2D range queries using our parallel sweepline paradigm, which can answer counting queries efficiently. We use the prefix structures to represent the outer map. Then each prefix structure is an inner map tracking all points up to the current point sorted by y-coordinates. The combine function of the outer map is **UNION**, so the update function h can be an insertion and the fold function ρ builds an inner map from a list of points. The definition of this sweepline paradigm R_S is shown in Table 4.

Since the inner maps are implemented by augmented trees, the theoretical bound of the functions (**INSERT**, **BUILD**, and **UNION**) are consistent with the assumptions in Theorem 4.1. Thus the theoretical cost of *RangeSwp* follows from Theorem 4.1. This data structure takes $O(n \log n)$ space because of persistence. We note that previous work [36] showed a more space-efficient version (linear space), but the goal in this paper is to show a generic paradigm that can easily implement many different problems without much extra effort.

Answering Queries. Computing $\textbf{ARANGE}(r_M, x_L, x_R)$ explicitly in Equation 6.6 on *RangeSwp* can be costly. We note that it can be computed by taking a **DIFFERENCE** on the prefix structure t_R at x_R and the prefix structure t_L at x_L (each can be found by a binary search). If only the count is required, a more efficient query can be applied. We can compute the number of points in the range (y_L, y_R) in t_L and

t_R respectively, using **ARANGE**, and the difference of them is the answer. Two binary searches cost $O(\log n)$, and the range search on y-coordinates costs $O(\log n)$. Thus the total cost of a single query is $O(\log n)$.

Extension to Report All Points. This sweepline algorithm can be inefficient in list-all queries. Here we propose a variant for list-all queries in Appendix C. The cost of one query is $O(\log n + k\log(\frac{n}{k}+1))$ for output size k. Comparing with *RangeTree*, which costs $O(k+\log^2 n)$ per query, this algorithm is asymptotically more efficient when $k < \log n$.

7 2D Segment Query

Given a set of non-intersecting 2D segments, and a vertical segment S_q, a segment query asks for some information about the segments that cross S_q. We define a segment as its two endpoints (l_i, r_i) where $l_i, r_i \in D, x(l_i) \le x(r_i)$, and say it *starts from* $x(l_i)$ and *ends at* $x(r_i)$.

In this paper, we introduce a two-level map structure *SegMap* addressing this problem (shown in Table 4 and Figure 1 (b)). The keys of the outer map are the x-coordinates of all endpoints of the input segments, and the values are the corresponding segments. Each (sub-)outer map corresponds to an interval on the x-axis (from the leftmost to the rightmost key in the sub-map), noted as the *x-range* of this map. The augmented value of an outer map is a pair of inner maps: $L(\cdot)$ (the *left open set*) which stores all input segments starting outside of its x-range and ending inside (i.e., only the right endpoint is in its x-range), and symmetrically $R(\cdot)$ (the *right open set*) with all segments starting inside but ending outside. We call them the *open sets* of the corresponding interval. The open sets of an interval u can be computed by combining the open sets of its sub-intervals. In particular, suppose u is composed of two contiguous intervals u_l and u_r, then u's open sets can be computed by a function f_{seg} as:

$$\boldsymbol{f}_{\text{seg}}(\langle L(u_l), R(u_l)\rangle, \langle L(u_r), R(u_r)\rangle) =$$
$$\langle L(u_l) \cup (L(u_r) \backslash R(u_l)), R(u_r) \cup (R(u_l) \backslash L(u_r))\rangle \tag{7.7}$$

Intuitively, taking the right open set as an example, it stores all segments starting in u_r and going beyond, or those stretching out from u_l but *not* ending in u_r. This function is associative. We use f_{seg} as the combine function of the outer map.

The base function g_{seg} of the outer map (see Table 4) computes the augmented value of a single entry. For an entry $(x_k, (p_l, p_r))$, the interval it represents is the solid point at x_k. WLOG we assume $x_k = x(p_l)$ such that the key is the left endpoint. Then the only element in its open sets is the segment itself in its right open set. If $x_k > x_v$ it is symmetric.

We organize all segments in an inner map sorted by their y-coordinates and augmented by the count, such that in queries, the range search on y-coordinates can be done in the corresponding inner maps. We note that all segments in a certain inner tree must cross one common x-coordinate. For example, in the left open set of an interval i, all segments must cross the left border of i. Thus we can use the y-coordinates at this point to determine the ordering of all segments. Note that input segments are non-intersecting, so this ordering of two segments is consistent at any x-coordinate. The definition of such an inner map is in Table 4 (the inner map S_I). The construction of the two-level map *SegMap* (S_M) from a list of segments $B = \{(p_l, p_r)\}$ can be done as follows:

$$s_M = S_M.\textbf{BUILD}(B'), \text{ where}$$
$$B' = \{(x(p_l), (p_l, p_r)), (x(p_r), (p_l, p_r)) : (p_l, p_r) \in B\}$$

Assume the query segment is (p_s, p_e), where $x(p_s) = x(p_e) = x_q$ and $y(p_s) < y(p_e)$. The query will first find all segments that cross x_q, and then conduct a range query on $(y(p_s), y(p_e))$ on the y-coordinate among those segments. To find all segments that cross x_q, note that they are the segments starting before x_q but ending after x_q, which are exactly those in the right open set of the interval $(-\infty, x_q)$. This can be computed by the function **ALEFT**. The counting query can be done using the augmented map interface as:

$$\textbf{SEGQUERY}(s_M, p_s, p_e) = S_I.\textbf{ARANGE}$$
$$(\textbf{ROPEN}(S_M.\textbf{ALEFT}(s_M, x(p_t))), y(p_s), y(p_e))$$

where **ROPEN**$(\cdot)$ extracts the right open set from an open set pair. The list-all query can be answered similarly using S_I.**RANGE** instead of S_I.**ARANGE**.

We use augmented trees for inner maps. We discuss two implementations of the outer map: the augmented tree (which yields a segment-tree-like structure *SegTree*) and the prefix structures (which yields a sweepline algorithm *SegSwp*). We also present another two-level augmented map (*Segment* Map*) structure that can answer counting queries on axis-parallel segments in Appendix A.

7.1 The Segment Tree.

If the outer map is implemented by an augmented tree, the *SegMap* becomes very similar to a segment tree (noted as *SegTree*). Previous work has studied a similar data structure [4, 13, 29]. We note that their version can deal with more types of queries and problems, but we know of no implementation work of a similar data structure. Our goal is to show how to apply the simple augmented map framework to model the segment query problem, and show an efficient and concise parallel implementation of it.

In segment trees, each subtree represents an open interval, and the union of all intervals in the same level span the whole interval (see Figure 1 (b) as an example). The intervals are separated by the endpoints of the input segments, and the two children partition the interval of the parent. Our version is slightly different from the classic segment trees in that we also use internal nodes to represent a point on the axis. For example, a tree node u denoting an interval (l, r) have its left child representing $(l, k(u))$, right child for $(k(u), r)$, and the single node u itself, is the solid point at it key $k(u)$. For each tree node, the *SegTree* tracks the open sets of its subtree's interval, which is exactly the augmented value of the sub-

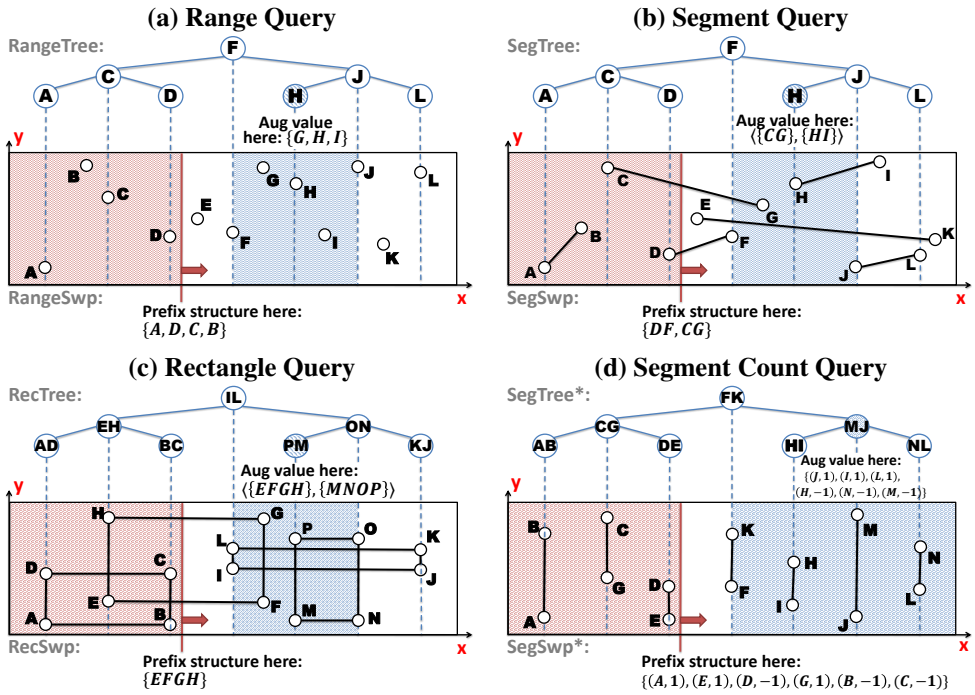


Figure 1: **An illustration of all data structures introduced in this paper** - The input data are shown in the middle rectangle. We show the tree structures on the top, and the sweepline algorithm on the bottom. All the inner trees (the augmented values or the prefix structures) are shown as sets (or a pair of sets) with elements listed in sorted order.

map rooted at u. The augmented value (the open sets) of a node can be generated by combining the open sets of its two children (and the entry in itself) using Equation 7.7.

Answering Queries more efficiently. Calling the **ALEFT** function on the outer tree of *SegTree* is costly, as it would require $O(\log n)$ function calls of **UNION** and **DIFFERENCE** on the way. Here we present a more efficient query algorithm making use of the tree structure, which is a variant of the algorithm in [13, 29]. Besides the open sets, in each node we store two helper sets (called the *difference sets*): the segments starting in its left half and going beyond the whole interval ($R(u_l) \backslash L(u_r)$ as in Equation 7.7), and vice versa ($L(u_r) \backslash R(u_l)$). We note that the calculation of the difference sets is not associative, but depends on the tree structure. These difference sets are the side-effect of computing the open sets. Hence in implementations we just keep them with no extra work. Suppose x_q is unique to all the input endpoints. The query algorithm first searches x_q outer tree. Let u be the current visited tree node. Then x_q falls in either the left or the right side of $k(u)$. WLOG, assume x_q goes right. Then all segments starting in the left half and going beyond the range of u should be reported because they must cover x_q. All such segments are in $R(l(u)) \backslash L(r(u))$, which is in u's difference sets. The range search on y-coordinates will be done on this difference sets tree structure. After that, the algorithm goes down to u's right child to continue the search recursively. The cost of returning all related segments is $O(k + \log^2 n)$ for output size k, and the cost of returning the count is $O(\log^2 n)$.

7.2 The Sweepline Algorithm. If prefix structures are used to represent the outer map, the algorithm becomes a sweepline algorithm *SegSwp* (shown as S_S in Table 4). We store at each endpoint p the augmented value of the prefix of all points up to p. Because the corresponding interval is a prefix, the left open set is always empty. For simplicity we only keep the right open set as the prefix structure, which is all "alive" segments up to the current event point (a segment (p_l, p_r) is alive at some point $x \in X$ iff $x(p_l) \leq x \leq x(p_r)$).

Sequentially, this is a standard sweepline algorithm for segment queries—as the line sweeping through the plane, each left endpoint should trigger an insertion of its corresponding segment into the prefix structure while the right endpoints cause deletions. We note that this is also what happens when a single point is plugged in as u_r in Equation 7.7. We use our parallel sweepline paradigm to parallelize this process. In the batching step, we compute the augmented value of each block, which is the open sets of the corresponding interval. The left open set of an interval are segments with their right endpoints inside the interval, noted as L, and the right open set is those with left endpoints inside, noted as R. In the sweeping step, the prefix structure is updated by the combine function f_{seg}, but only on the right open set, which is equivalent to first taking a **UNION** with R and then a **DIFFERENCE** with L. Finally, in the refining step, each left endpoint triggers an insertion and each right endpoint causes a deletion. This algorithm fits the sweepline abstraction in Theorem 4.1, so the corresponding bound holds.

Answering Queries. The **ALEFT** function on the prefix structure is just a binary search of x_q in the sorted list of x-coordinates. In that prefix structure all segments are sorted by y-coordinates, and we search the query range of y-coordinates on that. In all, a query for reporting all intersecting segments

costs $O(\log n + k)$ (k is the output size), and a query on counting related segments costs $O(\log n)$.

8 2D Rectangle Query

Given a set of rectangles in two dimensions, a rectangle query asks for all rectangles containing a query point $p_q = (x_q, y_q)$. Each rectangle $C = (p_l, p_r)$, where $p_l, p_r \in D$, is represented as its left-top and right-bottom vertices. We say the interval $[x(p_l), x(p_r)]$ and $[y(p_l), y(p_r)]$ are the *x-interval* and *y-interval* of C, respectively.

The rectangle query can be answered by a two-level map structure *RecMap* (G_M in Table 4 and Figure 1 (c)), which is similar to the *SegMap* as introduced in Section 7. The outer map organizes all rectangles based on their x-intervals using a similar structure to the outer map of *SegMap*. The keys of the outer map are the x-coordinates of all endpoints of the input rectangles, and the values are the rectangles. The augmented value of a (sub-)outer map is also the *open sets* as defined in *SegMap*, which store the rectangles that partially overlap the x-range of this sub-map. The combine function is accordingly the same as the segment map.

Each inner map of the *RecMap* organizes the rectangles based on their y-intervals. All the y-intervals in an inner tree are organized in an *interval tree* (the term *interval tree* refers to different definitions in the literature. We use the definition in [34]). The interval tree is an augmented tree (map) structure storing a set of 1D intervals sorted by the left endpoints, and augmented by the maximum right endpoint in the map. Previous work [63] has studied implementing interval trees using the augmented map interface. It can report all intervals crossing a point in time $O(\log n + k \log(n/k+1))$ for input size n and output size k.

RecMap answers the enclosure query of point (x_q, y_q) using a similar algorithm to *SegMap*. The query algorithm first finds all rectangles crossing x_q by computing the right open set R in the outer map up to x_q using **ALEFT**, which is an interval tree. The algorithm then selects all rectangles in R crossing y_q by applying a list-all query on the interval tree.

Using interval trees as inner maps does not provide an efficient interface for counting queries. We use the same inner map as in *SegMap** for counting queries. The corresponding map structure (*RecMap**) is presented in Appendix B.

We use augmented trees for inner maps (the interval trees). We discuss two implementations of the outer map: the augmented tree (which yields a multi-level tree structure) and the prefix structures (which yields a sweepline algorithm).

8.1 The Multi-level Tree Structure. *RecMap* becomes a two-level tree structure if the outer map is supported by an augmented tree, which is similar to the segment tree, and we use the same trick of storing the difference sets in the tree nodes to accelerate queries. The cost of a list-all query is $O(k \log(n/k+1) + \log^2 n)$ for output size k.

8.2 The Sweepline Algorithm. If we use prefix structures to represent the outer map, the algorithm becomes a sweepline algorithm (G_S in Table 4). The skeleton of the sweepline algorithm is the same as *SegSwp*—the prefix structure at event point x stores all "live" rectangle at x. The combine function, fold function and update function are defined similar as in *SegSwp*, but onto inner maps as interval trees. This algorithm also fits the sweepline abstraction in Theorem 4.1, so the corresponding bound holds.

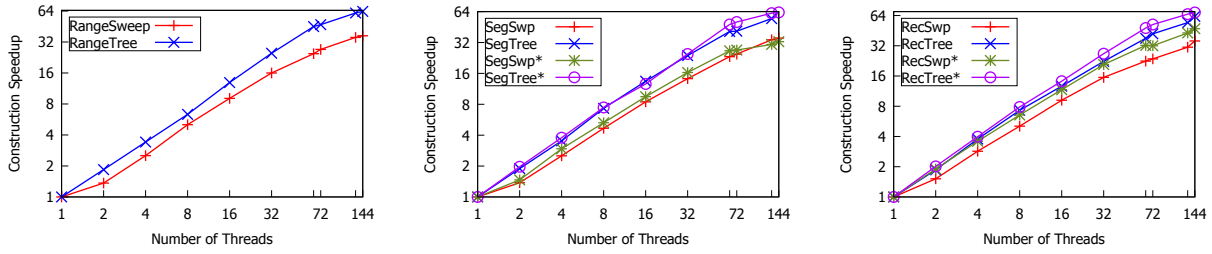
To answer the list-all query of point (x_q, y_q), the algorithm first finds the prefix structure t_q at x_q, and applies a list-all query on the interval tree t_q at point y_q. The cost is $O(\log n + k \log(n/k+1))$ per query for output size k.

9 Experiments

We implement all data structures mentioned in this paper in C++. We run our experiments on a 72-core Dell R930 with 4 x Intel(R) Xeon(R) E7-8867 v4 (18 cores, 2.4GHz and 45MB L3 cache) with 1TB memory. Each core is 2-way hyperthreaded giving 144 hyperthreads. Our code was compiled using the g++ 5.4.1 compiler which supports the Cilk Plus extensions. We compile with `-O2`.

All implementations of augmented trees used in the our algorithms are supported by the parallel augmented map library PAM [63], which supports persistence by path-copying. We implement the abstract parallel sweepline paradigm as described in Section 4. On top of them, each of our data structures only need about 100 lines of code. Our code is available online at `https://github.com/cmuparlay/PAM`. More details about PAM can be found online [61] and in our previous work [63].

For range queries, we test *RangeTree* and *RangeSwp* in Section 6. The tested *RangeTree* implementation is based on the range tree code in the previous paper [63]. For segment queries, we test *SegTree* and *SegSwp* in Section 7, as well as the counting versions *SegTree** and *SegSwp** in Appendix A. For rectangle queries, we test *RecTree* and *RecSwp* in Section 8, as well as the counting versions *RecTree** and *RecSwp** in Appendix B. We use integer coordinates. We test both construction time and query time, and both counting queries and list-all queries. In some of the problems, the construction of the data structure for counting and list-all queries can be slightly different, and we always report the version for counting queries. On list-all queries, since the cost is largely affected by the output size, we test a small and a large query window with average output size of less than 10 and about 10^6 respectively. We accumulate query time over 10^3 large-window queries and over 10^6 small window queries. For counting queries we accumulate 10^6 queries. For parallel queries we process all queries in parallel using a parallel for-loop. The sequential algorithms tested in this paper are directly running the parallel algorithms on one core. We use n for the input size, k the output size, p the number



(a). Range Query **(b). Segment Query** **(c). Rectangle Query**

Figure 2: The speedup of building various data structures for range, segment and rectangle queries ($n = 10^8$).

of threads. For the sweepline algorithms we set $b = p$, and do not apply the sweepline paradigm recursively to blocks.

We compare our sequential versions with two C++ libraries CGAL [64] and Boost [1]. CGAL provides a range tree [52] structure similar to ours, and the segment tree [52] in CGAL implements the 2D rectangle query. Boost provides an implementation of R-trees, which can be used to answer range, segment and rectangle queries. CGAL and Boost only support list-all queries. We parallelize the queries in Boost using OpenMP. CGAL uses some shared state in queries so the queries cannot be parallelized trivially. We did not find comparable parallel implementations in C++, so we compare our parallel query performance with Boost. We also compare the query and construction performance of our multi-level tree structures and sweepline algorithms with each other, both sequentially and in parallel.

In the rest of this section we show results for range, segment and rectangle queries and comparisons across all tested structures. We show that our implementations achieve good speedup (32-126x on 72 cores with 144 hyperthreads). The overall sequential performance (construction and query) of our implementations is comparable or outperforms existing implementations. We present more experimental results in the full version of this paper [62].

9.1 2D Range Queries. We test *RangeTree* and *RangeSwp* for both counting and list-all queries, sequentially and in parallel. We test 10^8 input points generated uniformly randomly. For counting queries, we generate endpoints of the query rectangle uniformly randomly. For list-all queries with large and small windows, we control the output size by adjusting the average length of the edge length of the query rectangle. We show the running time in Table 5. We show the scalability curve for construction in Figure 2 (a).

Sequential Construction. *RangeTree* and *RangeSwp* have similar performance and outperform CGAL (2x faster) and Boost (1.3-1.5x). Among all, *RangeTree* is the fastest in construction. We guess the reason of the faster construction of our *RangeTree* than CGAL is that their implementation makes copies the data twice (once in merging and once to create tree nodes) while ours only copies the data once.

Parallel Construction. *RangeTree* achieves a 63-fold speedup on $n = 10^8$ and $p = 144$. *RangeSwp* has relatively worse parallel performance, which is a 33-fold speedup, and 2.3x slower than *RangeTree*. This is likely because of its worse theoretical depth ($\tilde{O}(\sqrt{n})$ vs. $O(\log^2 n)$). As for *RangeTree*, not only the construction is highly-parallelized, but the combine function (**UNION**) is also parallel. Figure 2(a) shows that both *RangeTree* and *RangeSwp* scale up to 144 threads.

Query Performance. In counting queries, *RangeSwp* shows the best performance in both theory and practice. On list-all queries, *RangeSwp* is much slower than the other two range trees when the query window is large, but shows better performance for small windows. These results are consistent with their theoretical bounds. Boost's R-tree is 1.5-26x slower than our implementations, likely because of lack of worst-case theoretical guarantee in queries. Our speedup numbers for queries are above 65 because they are embarrassingly parallel, and speedup numbers of our implementations are slightly higher than Boost.

9.2 2D Segment Query. We test 5×10^7 input segment, using *SegSwp*, *SegTree*, *SegSwp** and *SegTree**. Note that for these structures on input size n (number of segments), $2n$ points are needed in the map. Thus we use input size of $n = 5 \times 10^7$ for comparison with the maps for range queries. The x-coordinate of each endpoint is generated uniformly randomly. To guarantee that the input segments do not intersect, we generate n non-overlapping intervals as the y-coordinates and assign each of them to a segment uniformly randomly. For *SegSwp** and *SegTree**, each segment is directly assigned a y-coordinate uniformly randomly. For counting queries, we generate endpoints of the query segment uniformly randomly. For list-all queries with large and small windows, we control the output size by adjusting the average length of the query segment. We show the running times in Table 5. We also show the parallel speedup for construction in Figure 2(b). We discuss the performance of *SegTree** and *SegSwp** in the full version of this paper [62].

Sequential Construction. Boost is 1.4x faster than *SegSwp* and 2.4x than *SegTree*. This is likely due to R-tree's simpler

n	Algorithm	Build, s			Counting Query, μs			List-all (small), μs			List-all (large), ms		
		Seq.	Par.	Spd.	Seq.	Par.	Spd.	Seq.	Par.	Spd.	Seq.	Par.	Spd.
10^8	**RangeSwp**	243.89	7.30	33.4	12.74	0.15	86.7	11.44	0.13	85.4	213.27	1.97	108.4
	RangeTree	200.59	3.16	63.5	61.01	0.75	81.1	17.07	0.21	80.5	44.72	0.69	65.2
	Boost	315.34	-	-	-	-	-	25.41	0.51	49.8	1174.40	22.42	52.4
	CGAL	525.94	-	-	-	-	-	153.54	-	-	110.94	-	-
5×10^7	**SegSwp**	254.49	7.20	35.3	6.78	0.09	75.3	6.18	0.08	77.2	255.72	2.65	96.5
	SegTree	440.33	6.79	64.8	50.31	0.70	71.9	39.02	0.48	81.7	123.26	1.99	61.9
	Boost	179.44	-	-	-	-	-	7421.30	113.09	65.6	998.20	23.21	43.0
5×10^7	**SegSwp***	233.19	7.16	32.6	7.44	0.11	67.6	-	-	-	-	-	-
	SegTree*	202.01	3.21	63.0	33.58	0.40	83.8	-	-	-	-	-	-
5×10^7	**RecSwp**	241.51	6.76	35.7	-	-	-	8.34	0.10	83.4	575.46	5.91	97.4
	RecTree	390.98	6.23	62.8	-	-	-	43.57	0.58	75.1	382.26	5.35	71.4
	Boost	183.65	-	-	-	-	-	52.22	0.94	55.6	706.50	11.10	63.6
5×10^6	**CGAL**[1]	398.44	-	-	-	-	-	90.02	-	-	4412.67	-	-
5×10^7	**RecSwp***	585.18	12.37	47.32	6.56	0.05	126.1	-	-	-	-	-	-
	RecTree*	778.28	11.34	68.63	39.75	0.35	113.6	-	-	-	-	-	-

Table 5: **The running time of all data structures** - "Seq.", "Par." and "Spd." refer to the sequential, parallel running time and the speedup. [1]: Result of CGAL is shown as on input size 5×10^6. On 5×10^7, CGAL did not finish in one hour.

structure. However, Boost is 4-1200x slower in queries than our implementations. *SegTree* is the slowest in construction because it stores four sets (the open sets and the difference sets) in each node, and calls two **DIFFERENCE** and two **UNION** functions in each combine function.

Parallel Construction. In parallel construction, *SegTree* is slightly faster than *SegSwp*. Considering that *SegTree* is 1.7x slower than *SegSwp* sequentially, the good parallelism comes from its good scalability (64x speedup). The lack of parallelism of *SegSwp* is for the same reason as *RangeSwp*.

Query Performance. In the counting query and list-all query on small window size, *SegSweep* is significantly faster than *SegTree* as would be expected from its better theoretical bound. As for list-all on large window size, although *SegTree* and *SegSwp* have similar theoretical cost (output size k dominates the cost), *SegTree* is faster than *SegSwp* both sequentially and in parallel. This might have to do with locality. In the sweepline algorithms, the tree nodes even in one prefix structure were created at different times because of path-copying, and thus are not contiguous in memory, leading to bad cache performance. Both *SegSwp* and *SegTree* have better query performance than Boost's R-tree (8.7-1400x faster in parallel). Also, the Boost R-tree does not take advantage of smaller query windows. Comparing the sequential query performance on large windows with small windows, on outputting about 10^6x less points, *SegTree* and *SegSwp* are 3000x and 40000x faster respectively, while Boost's R-tree is only 130x faster. Our implementations on small windows is not 10^6x as fast as on large windows because on small windows the $\log n$ or $\log^2 n$ term dominates the cost. This illustrates that the bad query performance of R-trees comes from lack of worst-case theoretical guarantee. The query speedup of our implementations is over 61.

9.3 2D Rectangle Query. We test rectangle queries using *RecSwp*, *RecTree*, *RecSwp** and *RecTree**, on $n = 5 \times 10^7$. The the query points are generated uniformly randomly. For counting queries, the endpoints of the input rectangles are generated uniformly randomly. For list-all queries with large and small windows, we control the output size by adjusting the average length of the edge length of the input rectangle. The running times are reported in Table 5, and the parallel speedup for construction are in Figure 2(c). We discuss the performance of *RecTree** and *RecSwp** in the full version of this paper [62].

Sequential Construction. Sequentially, *RecSwp*, *RecTree* and Boost R-tree have performance close to segment queries – Boost is faster in construction than the other two (1.6-2.1x), but is much slower in queries, and *RecTree* is slow in construction because of its more complicated structure. CGAL did not finish construction on $n = 5 \times 10^7$ in one hour, and thus we present the results on $n = 5 \times 10^6$. In this case, CGAL has a performance slightly worse than our implementations even though our input size is 10x larger.

Parallel Construction. The parallel performance is similar to the segment queries, in which *RecTree* is slightly faster than *RecSwp* because of good scalability (62x speedup).

Query Performance. In list-all queries on a small window size, *RecSwp* is significantly faster than other implementations due to its better theoretical bound. Boost is 1.2-9x slower than our implementations when query windows are small, and is 1.2-2x slower when query windows are large, both sequentially and in parallel. The query speedup of our implementations is over 71.

9.4 Summary. *The sweepline algorithms usually perform better in sequential construction, but in parallel are slower than two-level trees.* This has to do with the better scalability of the two-level trees. With properly defined augmentation,

the construction of the two-level trees is automatically done by the augmented map constructor in PAM, which is highly-parallelized (polylog depth). For the sweepline algorithms, the parallelism comes from the blocking-and-batching process, with a $\tilde{O}(\sqrt{n})$ depth. Another reason is that more threads means more blocks for sweepline algorithms, introducing more overhead in batching and folding. Most of the implementations have close construction time. Sequentially *SegTree* and *RecTree* are much slower than the others, because they store more information in each node and have more complicated combine functions. The speedup of all sweepline algorithms are close at about 30-35x, and all two-level trees at about 62-68x.

In general, the sweepline algorithms are better in counting queries and small window queries but are slower in large window queries. In counting queries and small window queries (when the output size does not dominate the cost) the sweepline algorithms perform better because of the better theoretical bound. On large window queries (when the output size dominates the cost), the two-level tree structures performs better because of better locality.

Our implementations scale to 144 threads, achieve good speedup and show efficiency on large input size. Our implementation show good performance both sequentially and in parallel on input size as large as 10^8. On 72 cores with 144 hyperthreads, the construction speedup is 32-69x, and the query speedup is 65-126x.

Our implementations are always faster on queries than existing implementations with comparable or faster construction time, even sequentially. Our implementations outperform CGAL in both construction and queries by at least 2x. Overall, the Boost R-tree is about 1.5x slower to 2.5x faster than our algorithms in construction, but is always slower (1.6-1400x) in queries both sequentially and in parallel, likely because of lack of worst-case theoretical guarantee of R-trees.

9.5 Other Experiments. We also present the results and discussions on the performance of the data structures for counting queries, updating range trees and space (memory) consumption in the full version of this paper [62].

10 Conclusion

In this paper we developed implementations of a broad set of parallel algorithms for range, segment and rectangle queries that are very much faster and simpler than the previous implementations, and are also theoretically efficient. We did this by using the augmented map framework. Based on different representations (augmented trees and prefix structures), we designed both two-level trees and sweepline algorithms addressing range, segment and rectangle queries. We implemented all algorithms in this paper and tested the performance both sequentially and in parallel. Experiments show that our algorithms achieve good speedup, and have good performance even sequentially. The overall performance considering both construction and queries of our implementations outperforms existing sequential libraries such as CGAL and Boost.

Acknowlegement

This work was supported in part by NSF grants CCF-1408940, CCF-1533858, and CCF-1629444. We acknowledge Yan Gu for proof-reading the paper and the useful suggestions.

References

[1] Boost C++ Libraries. `http://www.boost.org/`, 2015.

[2] Peyman Afshani, Mark De Berg, Henri Casanova, Benjamin Karsin, Colin Lambrechts, Nodari Sitchinava, and Constantinos Tsirogiannis. An efficient algorithm for the 1d total visibility-index problem. In *Proc. Algorithm Engineering and Experiments (ALENEX)*, 2017.

[3] Pankaj K. Agarwal, Kyle Fox, Kamesh Munagala, and Abhinandan Nath. Parallel algorithms for constructing range and nearest-neighbor searching data structures. In *Proc. Symp. on Principles of Database Systems (PODS)*, 2016.

[4] Alok Aggarwal, Bernard Chazelle, Leo Guibas, Colm Ó'Dúnlaing, and Chee Yap. Parallel computational geometry. *Algorithmica*, 1988.

[5] Thomas D Ahle, Martin Aumüller, and Rasmus Pagh. Parameter-free locality sensitive hashing for spherical range reporting. In *Proc. the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2017.

[6] Deepak Ajwani, Nodari Sitchinava, and Norbert Zeh. Geometric algorithms for private-cache chip multiprocessors. In *European Symposium on Algorithms (ESA)*, 2010.

[7] Lars Arge, Gerth Stølting Brodal, Rolf Fagerberg, and Morten Laustsen. Cache-oblivious planar orthogonal range searching and counting. In *Proc. symposium on Computational geometry(SoCG)*, 2005.

[8] Lars Arge, Vasilis Samoladas, and Jeffrey Scott Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. Symp. on Principles of database systems (PODS)*, 1999.

[9] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM Journal on Computing*, 2003.

[10] Lars Arge and Norbert Zeh. Simple and semi-dynamic structures for cache-oblivious planar orthogonal range searching. In *Proc. Symp. on Computational Geometry (SoCG)*, 2006.

[11] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM J. Comput.*, 1989.

[12] Mikhail J Atallah and Michael T Goodrich. Efficient parallel solutions to some geometric problems. *Journal of Parallel and Distributed Computing*, 1986.

[13] Mikhail J. Atallah and Michael T. Goodrich. Efficient plane sweeping in parallel. In *Proc. Symposium on Computational Geometry (SoCG)*, 1986.

[14] Mikhail J Atallah, Michael T Goodrich, and S Rao Kosaraju. Parallel algorithms for evaluating sequences of set-manipulation operations. In *Aegean Workshop on Computing*, 1988.

[15] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Acm Sigmod Record*, 1990.

[16] J. L. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Trans. Comput.*, 1980.

[17] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 1975.

[18] Jon Louis Bentley. Decomposable searching problems. *Information processing letters*, 1979.

[19] Jon Louis Bentley and Jerome H Friedman. Data structures for range searching. *ACM Computing Surveys (CSUR)*, 1979.

[20] Jon Louis Bentley and Derick Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, 1980.

[21] Philip Bille, Inge Li Gørtz, and Søren Vind. Compressed data structures for range reporting. In *Proc. International Conference on Language and Automata Theory and Applications (LATA)*, 2015.

[22] Gabriele Blankenagel and Ralf Hartmut Güting. External segment trees. *Algorithmica*, 1994.

[23] Guy E Blelloch, Daniel Ferizovic, and Yihan Sun. Just join for parallel ordered sets. In *Proc. Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[24] Guy E Blelloch, Yan Gu, Yihan Sun, and Julian Shun. Parallel write-efficient algorithms and data structures for computational geometry. 2018.

[25] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. on Computing*, 1998.

[26] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 1974.

[27] Nieves R Brisaboa, Guillermo De Bernardo, Roberto Konow, Gonzalo Navarro, and Diego Seco. Aggregated 2d range queries on clustered points. *Information Systems*, 2016.

[28] Chee Yong Chan and Yannis E Ioannidis. Hierarchical cubes for range-sum queries. In *Proceedings of the 25th VLDB Conference*, 1999.

[29] Bernard Chazelle. Intersecting is easier than sorting. In *Proc. Symposium on Theory of Computing (STOC)*, 1984.

[30] Bernard Chazelle and Herbert Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM (JACM)*, 1992.

[31] Bernard Chazelle and Leonidas J Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1986.

[32] Siu Wing Cheng and Ravi Janardan. Efficient dynamic algorithms for some geometric intersection problems. *Information Processing Letters*, 1990.

[33] Yi-Jen Chiang. Experiments on the practical i/o efficiency of geometric algorithms: Distribution sweep versus plane sweep. *Computational Geometry*, 1998.

[34] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3rd edition)*. MIT Press, 2009.

[35] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry*. 2000.

[36] James R Driscoll, Neil Sarnak, Daniel Dominic Sleator, and Robert Endre Tarjan. Making data structures persistent. In *Proc. Symp. on Theory of computing (STOC)*, 1986.

[37] Herbert Edelsbrunner. A new approach to rectangle intersections part i. *International Journal of Computer Mathematics*, 1983.

[38] Herbert Edelsbrunner and Hermann A. Maurer. On the intersection of orthogonal objects. *Information Processing Letters*, 1981.

[39] Michael T Goodrich. Intersecting line segments in parallel with an output-sensitive number of processors. *SIAM Journal on Computing*, 1991.

[40] Michael T. Goodrich, Mujtaba R. Ghouse, and J Bright. Sweep methods for parallel computational geometry. *Algorithmica*, 1996.

[41] Michael T Goodrich and Darren Strash. Priority range trees. In *International Symposium on Algorithms and Computation (ISAAC)*, 2010.

[42] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 1969.

[43] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. Range queries in olap data cubes. In *Proc. International Conference on Management of Data (SIGMOD)*, 1997.

[44] Rico Jacob and Nodari Sitchinava. Lower bounds in the asymmetric external memory model. In *Proc. Symp. on Parallelism in Algorithms and Architectures (SPAA)*, 2017.

[45] Joseph JáJá. *An introduction to parallel algorithms*. Addison-Wesley, 1992.

[46] Ibrahim Kamel and Christos Faloutsos. *Parallel R-trees*. 1992.

[47] George S Lueker. A data structure for orthogonal range queries. In *Symp. Foundations of Computer Science*, 1978.

[48] Edward M McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. Technical report, 1980.

[49] Edward M. McCreight. Priority search trees. *SIAM J. Comput.*, 1985.

[50] Mark McKenney and Tynan McGuire. A parallel plane sweep algorithm for multi-core systems. In *Proc. SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2009.

[51] Kurt Mehlhorn and Stefan Näher. Implementation of a sweep line algorithm for the straight line segment intersection problem. 1994.

[52] Gabriele Neyer. dD range and segment trees. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.10 edition, 2017.

[53] Mark H Overmars. Geometric data structures for computer graphics: an overview. In *Theoretical foundations of computer graphics and CAD*. 1988.

[54] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an analysis of range query performance in spatial data structures. In *Proc. Symposium on Principles of Database Systems (PODS)*, 1993.

[55] Nick Roussopoulos and Daniel Leifker. Direct spatial search on pictorial databases using packed r-trees. In *ACM Sigmod*

Record, 1985.

[56] Neil Sarnak and Robert E. Tarjan. Planar point location using persistent search trees. *Commun. ACM*, 1986.

[57] Jens M Schmidt. Interval stabbing problems in small integer ranges. In *International Symposium on Algorithms and Computation*, 2009.

[58] Bernhard Seeger and Per-Åke Larson. Multi-disk b-trees. In *ACM SIGMOD Record*, 1991.

[59] M. I. Shamos and D. Hoey. Geometric intersection problems. In *Symp. on Foundations of Computer Science (FOCS)*, 1976.

[60] Nodari Sitchinava. Computational geometry in the parallel external memory model. *SIGSPATIAL Special*, 2012.

[61] Yihan Sun, Guy Blelloch, and Daniel Ferizovic. The PAM library. `https://github.com/cmuparlay/PAM`, 2018.

[62] Yihan Sun and Guy E Blelloch. Parallel range and segment queries with augmented maps. *arXiv preprint arXiv:1803.08621*, 2018.

[63] Yihan Sun, Daniel Ferizovic, and Guy E. Blelloch. PAM: parallel augmented maps. In *Proc. Symp. on Principles and Practice of Parallel Programming (PPoPP)*, 2018.

[64] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.12 edition, 2018.

[65] Dan E Willard. New data structures for orthogonal range queries. *SIAM Journal on Computing*, 1985.

[66] Boseon Yu, Hyunduk Kim, Wonik Choi, and Dongseop Kwon. Parallel range query processing on R-tree with graphics processing unit. In *Dependable, Autonomic and Secure Computing (DASC)*, 2011.

[67] Changxi Zheng, Guobin Shen, Shipeng Li, and Scott Shenker. Distributed segment tree: Support of range query and cover query over DHT. In *IPTPS*, 2006.

A Data Structures for Segment Counting Queries

In this section, we present a simple two-level augmented map *SegMap** structure to answer segment count queries (see the *Segment Count Query* in Table 6 and Figure 1 (d)). This map structure can only deal with axis-parallel input segments. For each input segment (p_l, p_r), we suppose $x(p_l) = x(p_r)$, and $y(p_l) < y(p_r)$. We organize the x-coordinates in the outer map, and deal with y-coordinates in the inner trees. We first look at the inner map. For a set of 1D segments, a standard solution to count the segments across some query point x_q is to organize all end points in sorted order, and assign signed flags to them as values: left endpoints with 1, and right endpoints with -1. Then the prefix sum of the values up to x_q is the number of alive segments. To efficiently query the prefix sum we can organize all endpoints as keys in an augmented map, with values being the signed flags, and augmented values adding values. We call this map the count map of the segments.

To extend it to 2D scenario, we use a similar outer map as range query problem. On this level, the x-coordinates are keys, the segments are values, and the augmented value is the count map on y-coordinates of all segments in the outer map. The combine function is **UNION** on the count maps. However, different from range maps, here each tree node represents *two* endpoints of that segment, with signed flags 1 (left) and -1 (right) respectively, leading to a different base function (g^*_{seg}).

We maintain the inner maps using augmented trees. By using augmented trees and prefix structures as outer maps, we can define a two-level tree structure and a sweepline algorithm for this problem respectively. Each counting query on the count map of size m can be done in time $O(\log m)$. In all, the rectangle counting query cost time $O(\log^2 n)$ using the two-level tree structure *SegTree**, and $O(\log n)$ time using the sweepline algorithm *SegSwp**.

We present corresponding definition and illustration on both the multi-level tree structure and the sweepline algorithm in Table 4 and Figure 1 (d).

B Data Structures for Rectangle Counting Queries

In this section, we extend the *RecMap* structure to *RecMap** for supporting fast counting queries. We use the exactly outer map as *RecMap*, but use base and combine functions on the corresponding inner maps. The inner map, however, is the same map as the *count map* in *SegMap** (S^*_I in Table 6). Then in queries, the algorithm will find all related inner maps, which are count maps storing all y-intervals of related rectangles. To compute the count of all the y-intervals crossing the query point y_q, the query algorithm simply apply an **ALEFT** on the count maps.

We maintain the inner maps using augmented trees. Using augmented trees and prefix structures as outer maps, we can define a two-level tree structure and a sweepline algorithm for this problem respectively. The rectangle counting query cost time $O(\log^2 n)$ using the two-level tree structure *RecTree**, and $O(\log n)$ time using the sweepline algorithm *RecSwp**.

We present corresponding definition and illustration on both the multi-level tree structure and the sweepline algorithm in Table 4. The outer representation of *RecMap** is of the same format as *RecMap* as shown in Figure 1 (c).

C Extend RangeSwp to Report All Points

In the main body we have mentioned that by using a different augmentation, we can adjust the sweepline algorithm for range query to report all queried points. It is similar to *RangeSwp*, but instead of the count, the augmented value is the maximum x-coordinate among all points in the map. To answer queries we first narrow the range to the points in the inner map t_R by just searching x_R. In this case, t_R is an augmented tree structure. Then all queried points are those in t_R with x-coordinates larger than x_L and y-coordinate in $[y_L, y_R]$. We still conduct a standard range query in $[y_L, y_R]$ on t_R, but adapt an optimization that if the augmented value of a subtree node is less than x_L, the whole subtree is discarded. Otherwise, at least part of the points in the tree would be

*** Segment Count Query:**

(Inner Map)S_I^*= AM ($\boldsymbol{K}$: Y; $\prec$: $<_Y$; $\boldsymbol{V}$: $D \times D$; $\boldsymbol{A}$: $\mathbb{Z}$; $\boldsymbol{g}$: $(k, v) \mapsto 1$; $\boldsymbol{f}$: $+_{\mathbb{Z}}$; $\boldsymbol{I}$: 0)

- SegMap* S_M^* = AM ($\boldsymbol{K}$: X; $\prec$: $<_X$; $\boldsymbol{V}$: $Y \times Y$; $\boldsymbol{A}$: S_I^*; $\boldsymbol{g}$: g_{seg}^* $\boldsymbol{f}$: S_I^*.union $\boldsymbol{I}$: $\emptyset$)

$\boldsymbol{g}_{\text{seg}}^*(\boldsymbol{x}, (\boldsymbol{l}, \boldsymbol{r})) = C_I.\text{build}(\{(l, 1), (r, -1)\})$

- SegSwp* S_S^* = PS ($\boldsymbol{P}$: $D \times D$; $\prec$: $<_Y$; $\boldsymbol{T}$: C_I; $\boldsymbol{t_0}$: $\emptyset$; $\boldsymbol{h}$: h_{seg}^*; $\boldsymbol{\rho}$: ρ_{seg}^*; $\boldsymbol{f}$: C_I.union)

$\boldsymbol{h}_{\text{seg}}^*(\boldsymbol{t}, (\boldsymbol{p_l}, \boldsymbol{p_r})) = C_I.\text{union}(t, \{(y(p_l), 1), (y(p_r), -1)\})$, $\boldsymbol{\rho}_{\text{seg}}^*(\{(p_l, p_r)\}) = C_I.\text{build}(\{(y(p_l), 1), (y(p_r), -1)\})$

*** Rectangle Count Query:**

(Inner Map)G_I^*= S_I^*

-RecMap* G_M^* = similar as G_M, but use G_I^* as inner maps

- RecSwp* G_S^* = similar as G_S, but use G_I^* as prefix structures

Table 6: **Definitions of *SegMap** and *RecMap**** - X and Y are types of x- and y-coordinates. $D = X \times Y$ is the point type.

relevant and we recursively deal with its two subtrees.

Now we analyze the cost of this algorithm. Let the output size be k. The total cost is asymptotically the number of tree nodes the algorithm visits, which is asymptotically the number of all reported points and their ancestors. For k nodes in a balanced tree of size n, the number of all its ancestors is equivalent to all the nodes visited by the **UNION** function based on split-join model [23] when merging this tree with a set of the k nodes. When using AVL trees, red-black trees, weight-balanced trees or treaps, the cost of the **UNION** function is $O(k \log(n/k + 1))$. Detailed proof for the complexity of the **UNION** function can be found in [23].

D Proof for Corollary 4.1

Proof. To reduce the depth of the parallel sweepline paradigm mentioned in Section 4, we adopt the same algorithm as introduced in Theorem 4.1, but in the last refining step, repeatedly apply the same algorithm on each block. If we repeat for a c of rounds, for the i-th round, the work would be the same as splitting the total list into k^i blocks. Hence the work is still $O(n \log n)$ every round. After c rounds the total work is $O(cn \log n)$.

For depth, notice that the first step costs logarithmic depth, and the second step, after c iterations, in total, requires depth $\tilde{O}(cb)$ depth. The final refining step, as the size of each block is getting smaller and smaller, the cost of each block is at most $O(\frac{n}{b^i} \log n)$ in the i-th iteration. In total, the depth is $\tilde{O}\big(cb + \frac{n}{b^c}\big)$, which, when $b = c^{\frac{c}{c+1}} n^{\frac{1}{c+1}}$, is $\tilde{O}(n^{1/(c+1)})$. Let $\epsilon = 1/(c + 1)$, which can be arbitrary small by adjusting the value of c, we can get the bound in Corollary 4.1.

Specially, when $c = \log n$, the depth will be reduced to polylogarithmic, and the total work is accordingly $O(n \log^2 n)$. This is equivalent to applying a recursive algorithm (similar to the divide-and-conquer algorithm of the prefix-sum problem). Although the depth can be polylogarithmic, it is not work-efficient any more. If we set c to some given constant, the work and depth of this algorithm are $O(n \log n)$ and $O(n^\epsilon)$ respectively.

E Dynamic Update on Range Trees Using Augmented Map Interface

The tree-based augmented map interface supports insertions and deletions (implementing the appropriate rotations). This can be used to insert and delete on the augmented tree interface. However, by default this requires updating the augmented values from the leaf to the root, for a total of $O(n)$ work. Generally, if augmented trees are used to support augmented maps, the insertion function will re-compute the augmented values of all the nodes on the insertion path, because inserting an entry in the middle of a map could completely change the augmented value. In the range tree, the cost is $O(n)$ per update because the combine function (**UNION**) has about linear cost. To avoid this we implemented a version of "lazy" insertion/deletion that applies when the combine function is commutative. Instead of recomputing the augmented values it simply adds itself to (or removes itself from) the augmented values along the path using f and g. This is similar to the standard range tree update algorithm [47].

The amortized cost per update is $O(\log^2 n)$ if the tree is weight-balanced. Here we take the insertion as an example, but similar methodology can be applied to any mix of insertion and deletion sequences (to make deletions work, one may need to define the inverse function f^{-1} of the combine function). Intuitively, for any subtree of size m, imbalance occur at least every $\Theta(m)$ updates, each cost $O(m)$ to rebalance. Hence the amortized cost of rotations per level is $O(1)$, and thus the for a single update, it is $O(\log n)$ (sum across all levels). Directly inserting the entry into all inner trees on the insertion path causes $O(\log n)$ insertions to inner trees, each cost $O(\log n)$. In all the amortized cost is $O(\log^2 n)$ per update.

Similar idea of updating multi-level trees in (amortized) poly-logarithmic time can be applied to *SegTree**, *RecTree* and *RecTree**. For *SegTree*, the combine function is not communicative, and thus update may be more involved than simply using the interface of lazy-insert function.