An Efficient Algorithm for the 1D Total Visibility-Index Problem and its Parallelization

PEYMAN AFSHANI, MADALGO, Aarhus University
MARK DE BERG, TU Eindhoven
HENRI CASANOVA, University of Hawaii at Manoa
BENJAMIN KARSIN, University of Hawaii at Manoa
COLIN LAMBRECHTS, TU Eindhoven
NODARI SITCHINAVA, University of Hawaii at Manoa
CONSTANTINOS TSIROGIANNIS, MADALGO, Aarhus University

Let T be a terrain and P be a set of points (locations) on its surface. An important problem in Geographic Information Science (GIS) is computing the *visibility index* of a point p on P, that is, the number of points in P that are visible from p. The *total visibility-index* problem asks for the visibility index of *every* point in P. Many applications of this problem involve 2-dimensional terrains represented by a grid of $n \times n$ square cells, where each cell is associated with an elevation value, and P consists of the center-points of these cells. Current approaches for computing the total visibility-index on such a terrain take at least quadratic time with respect to the number of the terrain cells. Finding a subquadratic solution to this *2D total visibility-index* problem remains an open problem. Furthermore, no subquadratic solution to the 1D version of this problem has been proposed; in the 1D problem, the terrain is an x-monotone polyline, and P is the set of the polyline vertices.

We present an $O(n \log^2 n)$ algorithm that solves the 1D total visibility-index problem. Our algorithm is based on a geometric dualization technique, which reduces the problem into a set of instances of the red-blue line segment intersection counting problem. We also present a parallel version of this algorithm, which requires $O(\log^2 n)$ time and $O(n \log^2 n)$ work in the CREW PRAM model. We implement a naive $O(n^2)$ approach and four variations of our algorithm: one that uses an existing red-blue line segment intersection counting algorithm and three new approaches that perform the intersection counting by leveraging features specific to our problem. We present experimental results for both serial and parallel implementations on large synthetic and real-world datasets, using two hardware platforms. Results show that all variants of our algorithm outperform the naive approach by several orders of magnitude on large datasets. Furthermore, we show that the fastest of our new intersection counting implementations reduces runtime by over 10 times, compared with an existing red-blue line segment intersection counting algorithm. Our fastest parallel implementation is able to process a terrain of more than 100 million vertices in under 3 minutes, achieving up to 85% parallel efficiency over serial execution.

CCS Concepts: •**Theory of computation → Computational geometry;** *Data structures design and analysis; Parallel algorithms; Divide and conquer;*

ACM Reference format:
Peyman Afshani, Mark de Berg, Henri Casanova, Benjamin Karsin, Colin Lambrechts, Nodari Sitchinava, and Constantinos Tsirogiannis. 2017. An Efficient Algorithm for the 1D Total Visibility-Index Problem and its Parallelization. *ACM J. Exp. Algor.* 1, 1, Article 1 (January 2017), 22 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Analyzing terrains to determine locations with special properties is a common objective in Geographic Information Science (GIS). One such property is visibility. In particular, one often wants to find points on a terrain that are highly visible or, conversely, points that are hardly visible. Example applications include the placement of telecommunication towers, placement of fire guard towers, surveying archaeological sites, military logistics, or surveying building sites. Thus, in recent years there has been a fair amount of work in the GIS literature dedicated to visibility analysis and the computations it entails (see the survey by Floriani and Magillo [12]), with many proposed algorithms [10, 11, 13, 17, 23, 24] as well as publicly available implementations [1, 2].

To automate terrain analysis, real-world terrains are approximated by digital models, with one of the most popular models being the *digital elevation model* (DEM). A DEM is a grid of square cells, where each cell is assigned an elevation (which typically corresponds to the elevation of the point on the terrain that appears at the center of the cell).

Let terrain T be a grid with $N = n^2$ total cells. Two cells c and c' are visible from each other if the line segment $\overline{cc'}$ that connects their center-points does not cross on the xy-domain any other cell g such that $\overline{cg}$ is steeper than $\overline{cc'}$. We define the *visibility index* of cell $c \in T$ to be the number of cells in T that are visible from c. The *total visibility-index* problem (also known as cumulative viewshed [25]) consists of finding the visibility index for every $c \in T$. One way to solve the total visibility-index problem on T is to compute the *viewshed* of each cell c of T, that is, to explicitly compute for each cell c which other cells of T are visible from c. With the algorithm of Van Kreveld [24] this takes $O(N \log N)$ time per cell, leading to a total running time of $O(N^2 \log N)$. Even for moderately-sized DEMs this is infeasible in practice, let alone for modern DEM datasets, which can consist of hundreds of millions of cells. One solution is to use a heuristic that approximates the visibility [11, 23]. Another is to observe that computing the viewsheds of different cells can be done independently, and to solve a large number of single-viewshed computations in parallel [5, 9, 19, 20, 26]. Still, such approaches are not suitable for large DEMs. The fundamental problem is that one cannot afford to explicitly compute all visible cells for each cell c of T, as this may produce an output of size $\Omega(N^2)$. Note that the total visibility index problem does not require to explicitly compute the viewshed of each cell in T; it only requires to compute the number of cells that are visible from each cell, therefore the output size for this problem is $\Theta(N)$.

So far finding a subquadratic algorithm to solve the 1D total visibility-index problem remains an open problem. Surprisingly, no efficient algorithm has been proposed even for the 1D version of the problem. In the 1D problem, the terrain T is an x-monotone polyline with n vertices. Similar to the 2D problem, the goal in the one-dimensional version is to compute for each vertex v in the polyline the number of vertices visible from v. We call this problem the *1D total visibility-index* problem. Note that on a 1D terrain T with n vertices, the visibility-index of a single vertex v can be computed in $\Theta(n)$ time; this could be done by moving away from v one vertex at a time, and maintaining two rays that define the horizon to the left and right of v. Using this method to compute the visibility-index for each vertex independently, we can compute the total visibility-index of T in $O(n^2)$ time. We refer to this simple algorithm as Naive. Despite its simplicity and disappointing quadratic performance, to the best of our knowledge, this is the best known solution for this problem to date.

Previous research have examined a problem highly related to the 1D total visibility-index problem, known as the *1.5D terrain-guarding problem* (TGP). The terrain-guarding problem involves finding the minimum number of points needed to view an entire 1-dimensional set of vertices. While

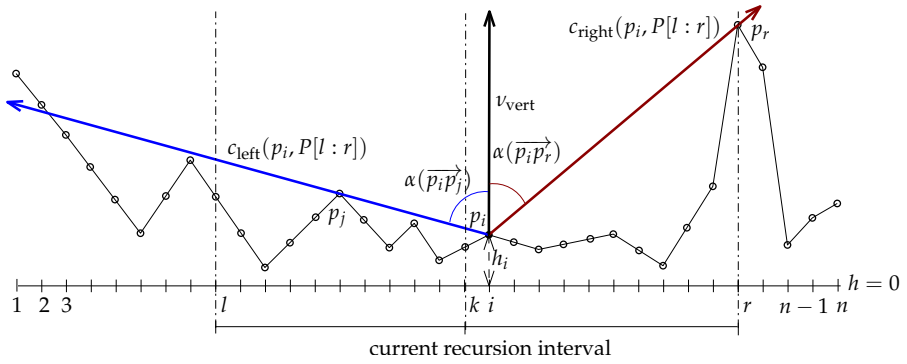


Fig. 1. Illustration of an 1-dimensional terrain, together with the critical rays from vertex p_i.

similar to the total visibility-index, solving TGP is known to be NP-hard and thus all previous results provide approximate solutions [14, 15, 21].

Our contributions. In this paper, we present an algorithm that solves the 1D total visibility-index problem for a terrain of n vertices in $O(n \log^2 n)$ time. Our algorithm uses a geometric dualization technique, which transforms the visibility problem into a set of instances of the *2D red-blue line segment intersection-counting* problem. In fact, we show that the instances of red-blue line segments that we have to process have characteristics that allow us to develop a simpler algorithm for counting intersections. This new intersection counting algorithm performs faster in practice than existing algorithms that solve the general red-blue line segment intersecting problem [22]. We also show how to parallelize our algorithm while keeping the overall work (time-processor product) the same. In particular, we present an adaptation of our algorithm in the CREW PRAM model [18], which requires $O(\log^2 n)$ time and $O(n \log^2 n)$ work. We implement the Naive $O(n^2)$ algorithm, as well as four variations of our algorithm: RedBlue employs an existing red-blue segment intersection counting algorithm [22], while Sweep, ParAoT, and LinPar implement three versions of our new intersection counting technique. Both ParAoT and LinPar allow for parallel execution of an arbitrary number of compute threads to improve performance. LinPar employs a space-efficient data structure to reduce the $O(n \log n)$ memory required by the simpler ParAoT.

We evaluate the performance of our implementations on large synthetic and real-world datasets, showing that all four implementations of our algorithm outperform the naive solution by several orders of magnitude. Additionally, we show that implementations employing our new intersection counting algorithm are able to reduce execution time by up to 18.69x over the existing general-case solution. We provide a detailed analysis of the performance of our two parallel implementations on two hardware platforms. Results indicate that our space-efficient solution, LinPar, provides the highest peak performance and is capable of processing over 100 million vertices in under 3 minutes, achieving up to 85% parallel efficiency.

2 PRELIMINARIES

Let $T[1..n]$ be a one-dimensional terrain, that is an array of cells in $\mathbb{R}^1$. Element $T[i]$ stores the elevation h_i of the i-th cell of the terrain. The array T defines an x-monotone polyline obtained by connecting the vertices $p_i := (i, h_i)$ for $i = 1, \ldots, n$ in order. Let $P = (p_1, p_2, \ldots, p_n)$ denote the sequence of these vertices ordered by their x-coordinates, and let $P[l:r]$ denote the subset of vertices $(p_l, \ldots, p_r)$. We say that a vertex p_j is *visible* from p_i (p_i *sees* p_j), if all vertices p_k between p_i and p_j lie strictly below the segment $\overline{p_i p_j}$. Based on this definition, we conclude that a vertex is

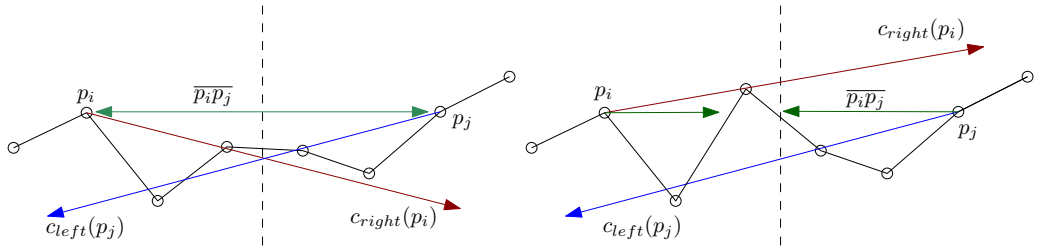


Fig. 2. Illustration of the intuition behind Lemma 2.1. **Left:** example where both points are above critical rays. **Right:** example where a point is below a critical ray.

visible from itself, and if vertex p_j is visible from vertex p_i, then p_i is also visible from p_j. We define the *visibility ray* from p_i to p_j, denoted $\overrightarrow{p_ip_j}$, as the ray that starts at p_i and passes through p_j. We define the visibility ray $\overrightarrow{p_ip_i}$ as the vertical ray that crosses p_i and points downwards. Let $\nu_{\text{vert}}(p_i)$ denote the ray that starts at p_i and points vertically up. We define the *angle* of the visibility ray $\overrightarrow{p_ip_j}$ as the smallest angle between $\overrightarrow{p_ip_j}$ and $\nu_{\text{vert}}(p_i)$. We use $\alpha(\overrightarrow{p_ip_j})$ to denote this angle.

One of the key concepts that we use in our analysis is that of the critical ray. Let l, i, and r be three positive integers such that $l \leq i \leq r \leq n$. The *left critical ray* of point p_i with respect to $P[l:r]$, is the visibility ray $\overrightarrow{p_ip_s}$ with the smallest $\alpha(\overrightarrow{p_ip_s})$ among all rays $\overrightarrow{p_ip_k}$ with $l \leq k \leq i$. We denote this ray by $c_{\text{left}}(p_i, P[l:r])$. If $i = l$ then $c_{\text{left}}(p_i, P[l:r])$ is defined as the ray pointing vertically down from p_i. The *right critical ray*, denoted $c_{\text{right}}(p_i, P[l:r])$, of p_i is the visibility ray $\overrightarrow{p_ip_t}$ $(i \leq t \leq r)$ with the smallest $\alpha(\overrightarrow{p_ip_t})$ (or pointing vertically down from p_i if $i = r$). See Figure 1 for an illustration of these rays. We can use critical rays to determine visibility between two points, as the following lemma shows.

Lemma 2.1. *Two points $p_i \in P[l:k]$ and $p_j \in P[k+1:r]$ are visible from each other if and only if p_i is above $c_{left}(p_j, P[k+1:r])$ and p_j is above $c_{right}(p_i, P[l:k])$.*

Proof. Let $c_{\text{right}} := c_{\text{right}}(p_i, P[l:k])$ be the right critical ray of p_i and let $c_{\text{left}} := c_{\text{left}}(p_j, P[k+1:r])$ be the left critical ray of p_j. Consider the line segment $\overline{p_ip_j}$. Assume that p_i is above c_{left} and that p_j is above c_{right}. Then all points $P[i:k]$ are below $\overline{p_ip_j}$, since by definition they lie below or on c_{right}.

Symmetrically, all points $P[k+1:j]$ are below $\overline{p_ip_j}$, due to c_{left}. Hence p_i and p_j are visible from each other. Now assume that p_i and p_j are visible from each other. That means that all points $P[i+1:j-1]$ are below $\overline{p_ip_j}$. All points that can possibly determine c_{right} and c_{left} are therefore also below $\overline{p_ip_j}$. Hence p_i is above c_{left} and p_j is above c_{right}. □

Figure 2 illustrates the intuition behind the previous lemma. Note that, while we use the restriction that visibility requires points to be *above* critical rays, this is just a matter of definition. Changing our visibility definition to include equality would not change the overall algorithm design or performance.

3 DESCRIPTION OF THE ALGORITHM

Let T be a one-dimensional terrain and let P be the set of its vertices. To compute the total visibility-index on T, we consider the following divide-and-conquer approach: first, we split the input polyline P into two subsets of equal size, and we recursively continue this process. After

computing the total visibility-index for each trivial base case, we move up in the hierarchy of recursive calls. At each step, we combine the results that we computed for two consecutive subsets $P[l:k]$ and $P[k+1:r]$ to produce the total visibility-index for subset $P[l:r]$. For each subset that we process, together with computing the visibility-index for each vertex p in the subset, we also construct the left and right critical ray of p with respect to this subset. At any point during this recursive execution, we use an array *VisIndex* such that $VisIndex[i]$ stores the total visibility-index of p_i computed in all previous levels of recursion. Suppose that, at some point during this recursion, we have already calculated the total visibility-index for two subsets $P[l:k]$ and $P[k+1:r]$, and we need to produce the result for their union $P[l:r]$. To do this, we need to compute for each $p_i \in P[l:k]$ the number of vertices of $P[k+1:r]$ that are visible to p_i and add this number to $VisIndex[i]$; similarly, for each $p_j \in P[k+1:r]$ we need to compute the number of points of $P[l:k]$ that are visible from p_j and add this to $VisIndex[j]$. We define *Bipartite Visibility* as this problem of finding the number of visibile vertices only between elements of two distinct subsets. In order to reduce each recursive step of our divide-and-conquer algorithm to an instance of Bipartite Visibility, we define the following invariants that must be satisfied for each $p_i \in P[l:k]$, resp. $p_j \in P[k+1:r]$:

- $VisIndex[i]$, resp. $VisIndx[j]$, has been computed within $P[l:k]$, resp. $P[k+1,r]$,
- $c_{\text{right}}(p_i, P[l:k])$ and $c_{\text{left}}(p_i, P[l:k])$ correspond to the maximum and minimum slope rays, respectively, between p_i and any $p_k \in P[l:k]$,
- symmetrically, $c_{\text{right}}(p_j, P[k+1:r])$ and $c_{\text{left}}(p_j, P[k+1:r])$ correspond to the maximum and minimum slope rays, respectively, between p_j and any $p_m \in P[k+1:r]$.

Additionally, we consider that the upper convex hulls of $P[l:k]$ and $P[k+1]$ have been computed in the previous recursive step. These are needed to update the critical rays for the next recursive step (this process is detailed in Section 3.3). With the above invariants satisfied, we solve an instance of Bipartite Visibility to compute the number of visible vertices between the two distinct subsets $P[l:k]$ and $P[k+1:r]$. We denote the entire divide-and-conquer algorithm that computes the total visibility-index of P as 1DVisibilityIndex. The runtime of 1DVisibilityIndex on P is given by the recurrence $\tau(n) = 2\tau(n/2) + f(n)$, where $f(n)$ is the time it takes to solve Bipartite Visibility for $P[1:n/2]$ and $P[n/2+1:n]$. Therefore, the algorithmic performance of this divide-and-conquer approach depends on an efficient solution for Bipartite Visibility. This section focuses on describing an algorithm that solves Bipartite Visibility in $O(n \log n)$ time, leading to:

Theorem 3.1. *Let T be an 1D terrain that consists of n vertices. We can compute the total visibility-index of T in $O(n \log^2 n)$ time, using $O(n)$ space.*

Let $P[l:k]$ and $P[k+1:r]$ be two parts of the terrain for which we want to solve Bipartite Visibility. Recall that for all vertices in $P[l:k]$ we have already computed the right critical rays with respect to $P[l:k]$, and for all vertices in $P[k+1:r]$ we have computed the left critical rays with respect to $P[k+1:r]$. Let p_i be a vertex in $P[l:k]$, and let p_j be a vertex in $P[k+1:r]$. Recall that, according to Lemma 2.1, vertices p_i and p_j are visible to each other if both p_j lies above the right critical ray of p_i, and p_i lies above the left critical ray of p_j. Therefore, to compute the number of vertices in $P[k+1:r]$ that are visible from p_i, we could explicitly check if this condition holds for each p_j in $P[k+1:r]$. This method, however, is inefficient as it requires that we check all possible pairs of vertices p_i, p_j s.t. $p_i \in P[l:k]$ and $p_j \in P[k+1:r]$.

We improve on this naive solution by using geometric duality [7]. Instead of handling the actual critical rays of the input points, we dualize these rays: we construct exactly one *dual half-line* for the right critical ray of each vertex in $P[l:k]$, and one *dual half-line* for the left critical ray of each vertex in $P[k+1:r]$. We refer to the duals of the right critical rays as the *red* half-lines, and

the duals of the left critical rays as the *blue* half-lines. We detail the construction of these dual half-lines in Section 3.1. As we will show with Lemma 3.2, we can construct the dual half-lines in such a way that the following property holds; a vertex p_i in $P[l:k]$ and a vertex p_j in $P[k+1:r]$ are visible if and only if the duals of their critical rays intersect. Hence, to compute the number of vertices in $P[k+1:r]$ that are visible from p_i, it suffices to count the number of blue half-lines that intersect with the dual of $c_{\text{right}}(p_i, P[l:r])$ (which is a red half-line). Thus, to solve this instance of Bipartite Visibility, we need to count, for each red and each blue dual half-line, the number of intersections that it induces with half-lines of the opposite color. In Section 3.1 we describe how we can do this efficiently in $O(n \log n)$ time.

In addition to computing Bipartite Visibility, at each recursive step of 1DVisibilityIndex, the critical rays of each vertex must be updated with respect to the subset containing it (e.g., $P[l:k]$). Therefore, after computing Bipartite Visibility between $P[l:k]$ and $P[k+1:r]$, we must update the critical rays of each vertex with respect to $P[l:k] \cup P[k+1:r]$. We detail the process of updating critical rays in Section 3.3. The remainder of the current section details the steps of 1DVisibilityIndex, with the pseudocode of the overall algorithm presented in Algorithm 1. In Section 3.1 we explain how we solve Bipartite Visibility by adapting an existing red-blue line segment intersection counting algorithm. We improve on this in Section 3.2, presenting our new, simpler algorithm to solve Bipartite Visibility. In Section 3.3 we describe a fast method of updating critical rays of each vertex at each recursive step.

The approach that we describe for Bipartite Visibility is similar to the method used by Ben-Moshe et al. [4] for computing the visibility graph between a set of points inside a polygon. However, since their goal is to construct the actual visibility graph (which can have quadratic size with respect to the input), they use an output-sensitive approach which is much slower than the methods that we describe for counting red-blue line segment intersections.

Algorithm 1 1DVisibilityIndex (P, l, r, VisIndex, CriticalRays)

Input: array P of n points p_i with elevations and two indices l and r.
Input: $VisIndex[1..n]$, where $VisIndex[i]$ denotes the visibility index of vertex p_i before the call.
Output: $VisIndex[i]$ = number of visible vertices in $P[l:r]$ for vertex p_i with $l \le i \le r$.
Output: $CriticalRays[i].left = c_{\text{left}}(p_i, P[l:r])$ for $l \le i \le r$.
Output: $CriticalRays[i].right = c_{\text{right}}(p_i, P[l:r])$ for $l \le i \le r$.

1 **if** $l = r$ **then**
2 Set $VisIndex[l] = 1$
3 Set $CriticalRays[l].left$ and $CriticalRays[l].right$ to be rays pointing downward
end
5 $k \leftarrow \lfloor \frac{r-l}{2} \rfloor + l$
6 1DVisibilityIndex $(T, l, k, VisIndex, CriticalRays)$
7 1DVisibilityIndex $(T, k+1, r, VisIndex, CriticalRays)$
8 $\mathcal{R} \leftarrow \{\rho(p_i, P[l:k]) : l \le i \le k\}$, $\mathcal{B} \leftarrow \{\beta(p_i, P[k+1:r]) : k+1 \le i \le r\}$
9 Count (for each half-line) the intersections between $\mathcal{R}$ and $\mathcal{B}$ using RedBlueIntersectionCount $(\mathcal{R}, \mathcal{B}, VisIndex)$
10 Update $VisIndex$ with intersection counts
11 Update $CriticalRays[i].right$ for every $l \le i \le k$
12 Update $CriticalRays[i].left$ for every $k+1 \le i \le r$

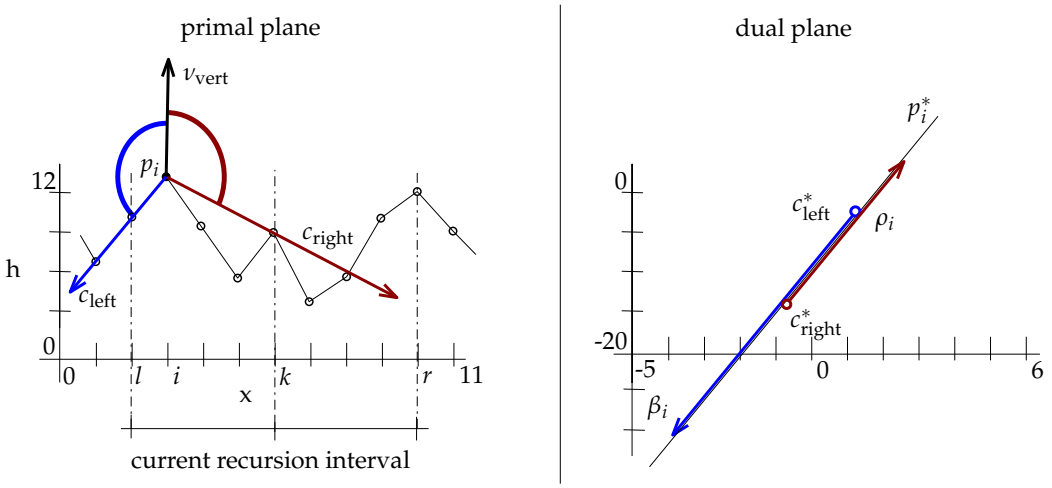


Fig. 3. An example of a terrain, its critical rays and their corresponding dual half-lines.

3.1 Constructing Dual Rays and Counting Red-Blue Intersections

In this section we describe how we utilize duality to reduce the Bipartite Visibility problem to the red-blue line segment intersection counting problem. We can thereby solve it using existing methods.

We define the dual of a point $p_i : (i, h_i)$ as the line $p_i^* : y = ix - h_i$, and the dual of a line $l : y = ax + b$ as the point $l^* : (a, -b)$. Let $P[l : r]$ be a subset of consecutive vertices in the input terrain. Consider vertex $p_i \in P[l : r]$ with the critical rays $c_{\text{right}}(p_i, P[l : r])$ and $c_{\text{left}}(p_i, P[l : r])$ lying along the lines $y = a_r x + b_r$ and $y = a_l x + b_l$, respectively. Let $\rho(p_i, P[l : r])$ be the dual of the set of lines which pass through p_i and have slopes *strictly larger* than a_r and let $\beta(p_i, P[l : r])$ be the dual of the set of lines which pass through p_i and with slopes *strictly smaller* than a_l. Note that for $p_i = (i, h_i)$, the dual objects $\rho(p_i, P[l : r])$ and $\beta(p_i, P[l : r])$ are collinear half-lines supported by the line $y = ix - h_i$ (of positive slope, because $1 \le i \le n$). However, $\rho(p_i, P[l : r])$ is defined over $x \in (a_r, +\infty)$, thus its endpoint is $c^*_{\text{right}} = (a_r, -b_r)$ and it extends to $+\infty$, while $\beta(p_i, P[l : r])$ is defined over $x \in (-\infty, a_l)$, thus its endpoint is $c^*_{\text{left}} = (a_l, -b_l)$ and it extends to $-\infty$. Also note that the half-lines are defined over open intervals $(a_r, +\infty)$ and $(-\infty, a_l)$. Therefore, the endpoints c^*_{right} and c^*_{left} do not belong to the half-lines $\rho(p_i, P[l : r])$ and $\beta(p_i, P[l : r])$, respectively. Refer to Figure 3 for an example.

Lemma 3.2. *Consider two points $p_i \in P[l : k]$ and $p_j \in P[k + 1 : r]$ and the critical rays $c_{right}(p_i, P[l : k])$ and $c_{left}(p_j, P[k + 1 : r])$, then p_i and p_j are visible from each other if and only if there is an intersection between dual half-lines $\rho(p_i, P[l : k])$ and $\beta(p_j, P[k + 1 : r])$.*

Proof. Suppose p_i and p_j are visible from each other. Consider the line l that passes through p_i and p_j. The dual of l is a point l^*. By Lemma 2.1, p_i must be above $c_{\text{left}}(p_j, P[k + 1 : r])$. Therefore, the slope of l must be smaller than the slope of $c_{\text{left}}(p_j, P[k + 1 : r])$ and, consequently, $l^* \in \beta(p_j, P[k + 1 : r])$. Similarly, by Lemma 2.1, p_j must be above $c_{\text{right}}(p_i, P[l : k])$. Therefore, the slope of l must be larger than the slope of $c_{\text{right}}(p_i, P[l : k])$ and, consequently, $l^* \in \rho(p_i, P[l : k])$. Since dual point l^* belongs to both dual half-lines, they must be intersecting at l^*.

Suppose $\beta(p_j, P[k + 1 : r])$ and $\rho(p_i, P[l : k])$ intersect at the dual point q^*. The dual point q^* corresponds to a line q that goes through both p_i and p_j. Since $q^* \in \rho(p_i, P[l : k])$, the slope of q

must be larger than the slope of $c_{\text{right}}(p_i, P[l:k])$, i.e. p_j must be above $c_{\text{right}}(p_i, P[l:k])$. Similarly, since $q^* \in \beta(p_j, P[k+1:r])$, the slope of q must be smaller than the slope of $c_{\text{left}}(p_j, P[k+1:r])$, i.e., p_i must be above $c_{\text{left}}(p_j, P[k+1:r])$. Therefore, by Lemma 2.1 p_i and p_j are visible from each other. □

Lemma 3.2 allows us to solve the Bipartite Visibility problem by computing for each dual half-line $\beta(p_j, P[k+1:r])$, how many half-lines $\rho(p_i, P[l:k])$ it intersects, and vice versa. The next lemma is important for finding an efficient intersection counting algorithm.

Lemma 3.3. *Let p_i and p_j, $i \neq j$, be two points in $P[l:k]$. Then the dual half-lines $\rho(p_i, P[l:k])$ and $\rho(p_j, P[l:k])$ do not intersect. Similarly, $\beta(p_i, P[l:k])$ and $\beta(p_j, P[l:k])$ do not intersect.*

Proof. Suppose for the sake of contradiction that $\rho(p_i, P[l:k])$ and $\rho(p_j, P[l:k])$ do intersect, which means that there is a visibility line $\overline{p_i p_j}$ between the p_i and p_j (in the primal plane). It also means that both $c_{\text{right}}(p_i, P[l:k])$ and $c_{\text{right}}(p_j, P[l:k])$ fall below $\overline{p_i p_j}$ (i.e., $\alpha(\overrightarrow{p_i p_j}) < \alpha(c_{\text{right}}(p_i, P[l:k]))$ and $\alpha(\overrightarrow{p_i p_j}) < \alpha(c_{\text{right}}(p_j, P[l:k]))$). By the definition of the critical ray, no visibility ray between two points in $P[l:k]$ can have a smaller angle α than the critical ray. Hence the angle must be equal to that of the critical ray and therefore the visibility line is the critical ray. This means that the intersection is at the starting point of the dual half-line. The starting point of a dual half-line is not considered part of the dual half-line and therefore $\rho(p_i, P[l:k])$ and $\rho(p_j, P[l:k])$ do not intersect. The proof for $\beta(p_i, P[l:k])$ and $\beta(p_j, P[l:k])$ is symmetric. □

Palazzi and Snoeyink [22] present an algorithm that computes, in $O(n \log n)$ time, the total number of intersections between a set of non-self-intersecting (red) line segments and another set of non-self-intersecting (blue) segments. Half-lines are a special case of line segments, where one endpoint is at ∞ (or $-\infty$). We note that the algorithm by Palazzi and Snoeyink produces only the total number of red-blue intersections (i.e., a single number), while we require intersection counts for *each half-line*. However, it is easy to modify their algorithm to produce the desired result without impacting asymptotic performance.

3.2 A Practical Algorithm for Red-blue Intersection Counting

While using the (modified) red-blue segment intersection algorithm of Palazzi and Snoeyink [22] provides an $O(n \log n)$ solution to Bipartite Visibility, it works for any red-blue line segments. As a result it is more complex than it has to be for our problem. Instead, in this subsection we present a simple plane sweep algorithm to count the intersections between duals of right and left critical rays. This plane sweep algorithm exploits some features of the dual half-lines of critical rays.

Let $\mathcal{R} = \{\rho(p_i, P[l:k])\}$ and $\mathcal{B} = \{\beta(p_j, P[k+1:r])\}$, be the set of *red* and *blue* self-non-intersecting half-lines (i.e., no half-lines intersect others of the same color). To describe our algorithm, we need to introduce some more notation. We denote the x- and y-coordinate of a vertex p by p_x and p_y, respectively. Given any half-line λ, we denote its endpoint by $\lambda_{x,y}$ and the x- and y-coordinates of the endpoint by λ_x and λ_y, respectively, i.e., $\lambda_{x,y} = (\lambda_x, \lambda_y)$. The y-coordinate of λ evaluated at x is denoted by $\lambda(x)$. That is, if λ is defined at x then vertex $p_{x,\lambda(x)} = (x, \lambda(x)) \in \lambda$. If λ is not defined at x, then we say $\lambda(x)$ is *undefined*. Finally, we say a vertex q is *above* (resp. *below*) a half-line λ, if $\lambda(q_x)$ is defined and $q_y > \lambda(q_x)$ (resp. $q_y < \lambda(q_x)$). If $\lambda(q_x)$ is undefined, then the above-below relationship between q and λ is undefined.

The following lemma is the key for developing a simple plane sweep algorithm for our red-blue half-line intersection counting problem.

Lemma 3.4. *Any two half-lines $\rho \in \mathcal{R}$ and $\beta \in \mathcal{B}$ intersect if and only if the endpoint $\rho_{x,y}$ is above β and the endpoint $\beta_{x,y}$ is above ρ.*

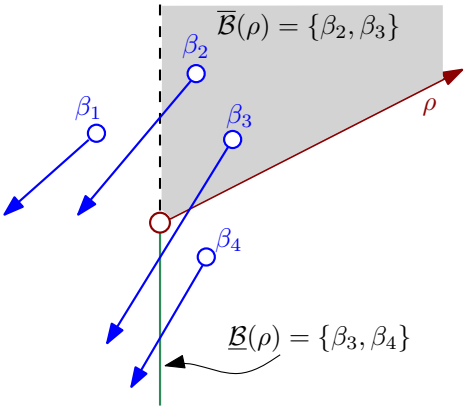


Fig. 4. An illustration of $\underline{\mathcal{B}}(\rho)$ and $\overline{\mathcal{B}}(\rho)$.

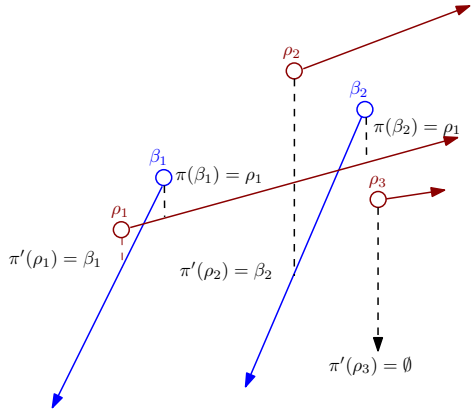


Fig. 5. Illustration of $\pi'(\rho)$ and $\pi(\beta)$ for several half-lines.

Proof. Suppose $\rho_{x,y}$ is above β and $\beta_{x,y}$ is above ρ. There must be a point q with $\rho_x < q_x < \beta_x$ s.t. $\rho(q_x) = \beta(q_x)$. Since ρ is continuous for all $x \geq \rho_x$ and β is continuous for all $x \leq \beta_x$, ρ and β intersect at q_x.

In the primal space, all points from the left merge set have smaller x-coordinates than any point from the right set. Therefore, all $\rho \in \mathcal{R}$ have a smaller slope than all $\beta \in \mathcal{B}$. It follows that, if ρ and β intersect at q, then $\rho(a) > \beta(a)$ and $\beta(b) > \rho(b)$ for all $a < q_x < b$. Since ρ_x has the smallest x-coordinate for which ρ is defined, then $\rho_x < q_x$. Therefore, $\rho_{x,y}$ is above β. Conversely, β_x is the largest x-coordinate of β, so $\beta_x > q_x$. Thus, $\beta_{x,y}$ is also above ρ. □

To compute the number of blue half-lines in $\mathcal{B}$ that each $\rho \in \mathcal{R}$ intersects, consider the following subsets of blue half-lines (see Figure 4):

- $\overline{\mathcal{B}}(\rho)$: blue half-lines $\beta \in \mathcal{B}$ with endpoints that are *above* ρ (i.e., $\beta_y > \rho(\beta_x)$)
- $\underline{\mathcal{B}}(\rho)$: blue half-lines $\beta \in \mathcal{B}$ that are *below* $\rho_{x,y}$ (i.e., $\beta(\rho_x) < \rho_y$)

By Lemma 3.4, the set of blue half-lines which intersect ρ is $\overline{\mathcal{B}}(\rho) \cap \underline{\mathcal{B}}(\rho)$ and by the inclusion-exclusion principle, its cardinality is $|\overline{\mathcal{B}}(\rho)| + |\underline{\mathcal{B}}(\rho)| - |\overline{\mathcal{B}}(\rho) \cup \underline{\mathcal{B}}(\rho)|$. Note that $\overline{\mathcal{B}}(\rho) \cup \underline{\mathcal{B}}(\rho)$ is the set of all blue half-lines with x-ranges that overlap with ρ, i.e. $\overline{\mathcal{B}}(\rho) \cup \underline{\mathcal{B}}(\rho) = \{\beta \in \mathcal{B} : \beta_x > \rho_x\}$. Figure 4 shows an example for a single red half-line and four blue half-lines.

Similarly, we define $\underline{\mathcal{R}}(\beta)$ and $\overline{\mathcal{R}}(\beta)$ and the number of red half-lines that intersect β is equal to $|\overline{\mathcal{R}}(\beta)| + |\underline{\mathcal{R}}(\beta)| - |\overline{\mathcal{R}}(\beta) \cup \underline{\mathcal{R}}(\beta)|$. Thus, it remains to compute each of these quantities.

3.2.1 Computing $|\underline{\mathcal{B}}(\rho)|$ and $|\underline{\mathcal{R}}(\beta)|$. To compute $|\underline{\mathcal{B}}(\rho)|$ we sweep the dual plane from right to left with a sweep line ℓ which is perpendicular to the x-axis. During the sweep, we maintain a balanced binary search tree (BST) $\mathcal{T}$ which stores all blue half-lines β that intersect ℓ, ordered by their slopes. Since blue half-lines do not intersect each other and continue to $-\infty$, this is the same order as the order of the blue half-lines by decreasing slopes. Thus, every time the sweep line encounters a blue half-line end point $\beta_{x,y}$, we insert β to $\mathcal{T}$. Whenever the sweep line encounters the endpoint $\rho_{x,y}$ of a red half-line, the number of blue half-lines below ρ is equal to the number of blue half-lines β with y-coordinate $\beta(\rho_x)$ smaller than ρ_y. And since all blue half-lines in $\mathcal{T}$ are defined at the time of the sweep, the above-below relationship between the endpoint $\rho_{x,y}$ and all blue half-lines in $\mathcal{T}$ is well-defined. Thus, we can compute $|\underline{\mathcal{B}}(\rho)|$ by performing a search in

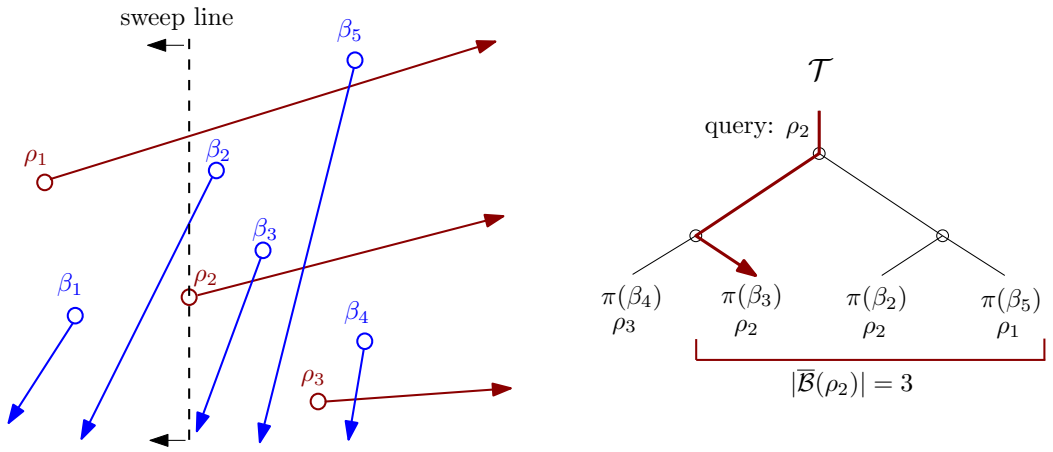


Fig. 6. Example of the plane sweep algorithm used to find $|\overline{\mathcal{B}}(\rho)|$. The vertical sweep line moves from right to left and, when a blue endpoint $\beta_{x,y}$ is encountered, $\pi(\beta)$ is added to the search tree $\mathcal{T}$. When the sweep line encounters a red endpoint $\rho_{x,y}$, the tree is queried by the slope of ρ. The number of leaves in the search tree that have slopes greater than or equal to the slope of ρ is equal to $|\overline{\mathcal{B}}(\rho)|$.

$\mathcal{T}$, comparing ρ_y to $\beta(\rho_x)$. The rank of ρ_y in the set of blue half-lines in $\mathcal{T}$ gives us $|\underline{\mathcal{B}}(\rho)|$ – the number of blue half-lines below ρ.

To implement this plane sweep, we need to sort $\mathcal{B}$ and $\mathcal{R}$ by the x-coordinates of their endpoints. Each insertion of a blue half-line in $\mathcal{T}$ takes $O(\log n)$ time. We can compute the rank of ρ_y in $\mathcal{T}$ in $O(\log n)$ time by augmenting each node v of $\mathcal{T}$ with the size of the subtree rooted at v. Thus, the total computation of $|\underline{\mathcal{B}}(\rho)|$ for all $\rho \in \mathcal{R}$ takes $O(n \log n)$ time.

Note that the size of $\mathcal{T}$ when the sweep line encounters $\rho_{x,y}$ is $|\overline{\mathcal{B}}(\rho) \cup \underline{\mathcal{B}}(\rho)|$ – the number of blue half-lines whose x-ranges overlap with ρ. During the computation we also record for each red half-line ρ the blue half-line $\pi'(\rho)$ that is immediately below ρ (the predecessor of ρ_y in the $\mathcal{T}$). Refer to Figure 5 for an illustration.

Computation of $|\underline{\mathcal{R}}(\beta)|$ is symmetric, with the sweep being performed from left to right. During the computation, we also record $|\overline{\mathcal{R}}(\beta) \cup \underline{\mathcal{R}}(\beta)|$ and $\pi(\beta)$ — the red half-line that is immediately below the endpoint of β. The concepts of $\pi(\beta)$ and $\pi'(\rho)$ will be used for computing $|\overline{\mathcal{B}}(\rho)|$ and $|\overline{\mathcal{R}}(\beta)|$, respectively.

3.2.2 Computing $|\overline{\mathcal{B}}(\rho)|$ and $|\overline{\mathcal{R}}(\beta)|$. The following description focuses on the computation of values $|\overline{\mathcal{B}}(\rho)|$; the computation of $|\overline{\mathcal{R}}(\beta)|$ is symmetric. Since computing $|\underline{\mathcal{B}}(\rho)|$ and $|\underline{\mathcal{R}}(\beta)|$ entails counting half-lines below each given endpoint, the above-below relationship is well-defined at the time the sweep line hits the endpoint in question. Here, instead, we are counting the number of points above a half-line, which must be counted for *every* half-line. To accomplish this efficiently, we assume that we have already computed $\pi(\beta)$ for each blue half-line β as described in Section 3.2.

To compute $|\overline{\mathcal{B}}(\rho)|$ we sweep a vertical line from right to left (refer to Figure 6 for an illustration). During the sweep we maintain a balanced binary search tree (BST) $\mathcal{T}$ on the *slopes* of $\pi(\beta)$. That is, when the sweep line encounters an endpoint of a blue half-line β and $\pi(\beta)$ is defined, we insert the slope of $\pi(\beta)$ into $\mathcal{T}$. If $\pi(\beta)$ is undefined, there is no red half-line below the end point of β and since each red half-line ρ is defined for all $x \geq \rho_x$, the endpoint of β does not lie above any red half-line and can be safely ignored.

At time ρ_x of the sweep, that is when the sweep line encounters a red half-line end point $\rho_{x,y}$, the number of entries in $\mathcal{T}$ that are greater than or equal to the slope of ρ is equal to the number of blue half-line endpoints above ρ. To see this, observe that when ρ_x is encountered, tree $\mathcal{T}$ contains all blue half-line endpoints that have a well-defined above-below relationship with ρ. Since the red half-lines do not intersect other red half-lines, the ordering of the slopes of the red half-lines is equivalent to the above-below relationship among the red half-lines which are defined at ρ_x. The above-below relationship between red half-lines and blue half-line endpoints defines a partial order, which means that if $\beta_{x,y}$ is above ρ_1, both ρ_1 and ρ_2 are defined at β_x and $\rho_1(\beta_x) > \rho_2(\beta_x)$, then $\beta_{x,y}$ is also above ρ_2. Consequently, the set of endpoints of blue half-lines above ρ is equal to the set of blue half-lines β with slopes of $\pi(\beta)$ greater than the slope of ρ. See Figure 6 for an illustration of this plane sweep.

Given the above, whenever the sweep line encounters an endpoint of a red half-line ρ, we perform predecessor/successor query on $\mathcal{T}$ using the slope of ρ to find the number of points above ρ. Maintaining and querying $\mathcal{T}$ take $O(\log n)$ time per blue half-line endpoint (insertion) or red half-line endpoint (query), resulting in $O(n \log n)$ time overall to compute $|\overline{\mathcal{B}}(\rho)|$ for each half-line ρ.

3.3 Maintaining Critical Rays

Our overall divide-and-conquer algorithm relies on the knowledge of the critical rays at the beginning of each recursive call. At the base case, subset $P[l:r]$ contains only one point. Therefore, both left and right critical rays of that point are directed vertically downward. Thereafter, at the end of each recursive call, we update these rays by recomputing only the right critical ray for each point in $P[l:k]$ and the left critical ray for each point in $P[k+1:r]$. To do this, we need the next lemma.

Lemma 3.5. *The tangent from $p_i \in P[l:k]$ to the upper convex hull of all vertices in $P[k+1:r]$ is the critical ray $c_{right}(p_i, P[l:r])$ if and only if the vertex p_t on the hull that the tangent goes through is visible to p_i. Symmetrically, the tangent $p_j \in P[k+1:r]$ to the upper hull of vertices in $P[l:k]$ is the critical ray $c_{left}(p_j, P[l:r])$ if and only if the tangent point $p_{t'}$ on the hull is visible to p_j.*

Proof. We first prove that only points on the upper hull of $P[k+1:r]$ can be candidates for defining $c_{right}(p_i, P[l:r])$. Suppose that p_t is the point in $P[k+1:r]$ that defines $c_{right}(p_i, P[l:r])$ and that p_t does not fall on the upper convex hull of $P[k+1:r]$. By definition, no point in $P[k+1:r]$ can fall outside the upper hull of the same point set, therefore p_t must fall inside the hull. In that case, the ray that starts from p_i and goes through p_t intersects the upper hull of $P[k+1:r]$. Let p' be this intersection point, and let p'' be the vertex of the upper hull which is exactly to the right of p'. Then p'' is visible from p_i, which contradicts the assumption that p_j defines $c_{right}(p_i, P[l:r])$.

Let p_t be the point on the upper hull, such that the tangent goes through p_t. If p_t is not visible to p_i, then there is a point p_k s.t. $t < k < i$ that is above the visibility ray $\overrightarrow{p_i p_t}$. Since the tangent from p_i to the upper hull goes through p_t, p_k must not be in the set encompassed by the upper hull. Therefore p_k is in the set included with p_i and the previous critical ray of p_i is steeper than the tangent, so the tangent is not $c_{\text{right}}(p_i, P[l:r])$.

If, however, p_t is visible to p_i, then $c_{\text{right}}(p_i, P[l:k])$ falls below p_t. Furthermore, the visibility ray from p_i to other points on the upper hull are below the tangent (by property of tangents) and therefore the tangent is the only visibility line that is not below any other point of the upper hull. Hence the tangent is $c_{\text{right}}(p_i, P[l:r])$. The proof involving $c_{\text{left}}(p_j, P[l:r])$ is symmetric. □

Thus, to update $c_{\text{left}}(p_i, P[l:r])$ and $c_{\text{right}}(p_i, P[l:r])$ we utilize the upper convex hulls of $P[l:k]$ and $P[k+1:r]$ (computed in the previous recursive step), and for every point in these two subsets

we construct the tangent to the hull of the opposite subset. To construct the $P[l : r]$ for the next recursive step, we simply merge our two upper hulls in $O(n)$ time. Computing tangents is equivalent to binary searches, which takes $O(\log n)$ time per tangent for a total of $O(n \log n)$ time. Combining this with the rest of the analysis presented in Section 3, we conclude that we can solve each recursive level of 1DVisibilityIndex in $O(n \log n)$ time. Hence, the total running time of algorithm 1DVisibilityIndex is $O(n \log^2 n)$.

4 PARALLEL EXTENSION

Persistence [8] is a technique for efficiently maintaining all past versions of a dynamic structure for future queries. An *offline* persistent binary search tree supports all standard update operations given up-front. Since all updates and queries are known before construction, all the updates can be built into the data structure during construction, allowing queries to be performed on any of its past versions. Each of these queries can be performed independently of each other. Thus, if a balanced offline persistent tree can be built efficiently in parallel, n queries can be answered in parallel in $O(\log n)$ time using n processors, i.e., in $O(n \log n)$ work, in the CREW PRAM model. Offline persistent BSTs can be used to solve some problems that are typically solved using a plane sweep algorithm. Therefore, in this section we detail the two offline persistent BST structures that we use to solve our red-blue line segment intersection problem (and thus solve Bipartite Visibility). Note that all other operations performed by our divide-and-conquer algorithm (described in Section 3) can be easily parallelized: all n critical rays can be updated concurrently in $O(\log n)$ time and, using these critical rays, we can merge upper convex hulls in $O(1)$ time and $O(n)$ work.

If we can implement the search tree used in the plane sweep of Section 3 as an offline persistent BST, we can perform the sweep in $O(\log n)$ time and $O(n \log n)$ work. Thus, the parallel runtime and work of the overall algorithm can be defined by the recurrences $\Phi(n) = \Phi(n/2) + O(\log n) = O(\log^2 n)$ and $W(n) = 2W(n/2) + O(n \log n) = O(n \log^2 n)$, respectively. This yields the following theorem.

Theorem 4.1. *The 1D total visibility-index problem can be solved in $O(\log^2 n)$ time and $O(n \log^2 n)$ work in the CREW PRAM model.*

The work complexity of the parallel algorithm matches our sequential algorithm runtime, which is the best we can hope for from a parallel algorithm.

We identify two offline persistent BST structures that we can use to solve Bipartite Visibility in parallel. In Section 4.1 we present an overview of these structures and in Section 4.2 we describe some relevant details of our implementations that leverage these structures.

4.1 Overview of persistent BST structures

In this subsection we provide overviews of two offline parallel BST structures that we employ: the *array-of-trees* [3] and the linear-space persistent BST [6]. We refer interested readers to [3] and [6], where the structures and techniques are presented in detail.

Array-of-trees. Atallah et al. [3] describe a data structure that they call *array-of-trees*, which implements a persistent search tree and can be built in the CREW PRAM model in $O(\log n)$ time and $O(n \log n)$ work. Hence, we can implement the tree structure used in the plane sweep of Section 3.2 as an array-of-trees, and thus perform the sweep in $O(\log n)$ parallel time and $O(n \log n)$ work.

Our first parallel implementation replaces each plane sweep operation described in Section 3.1 with the construction and querying of an array-of-trees (AoT). AoTs are constructed by starting with the input data as a set of pairs (key k, time t), sorted by k. This initial dataset becomes the leaf level of the AoT, on top of which the structure can be constructed bottom-up by a variation of

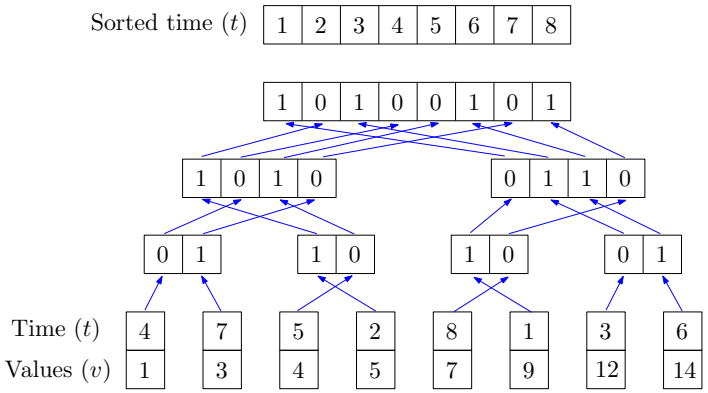


Fig. 7. Illustration of how a LPBST is constructed.

merge sort on t. At each level, pairs of sets are merged to form parent nodes, consisting of all of the t values of its children in sorted order. Each t value also maintains pointers to the elements in each child node with largest t_{child}, s.t. $t_{child} \leq t$. The top level of the AoT contains a single list sorted by t, with each element corresponding to a root node of a BST, searchable by key k.

Querying the AoT involves two steps: 1) finding the correct root and 2) querying the corresponding BST. Since the top level of the AoT is a list sorted by t, the correct BST can be found by performing a binary search using the query time. The associated BST can then be searched with the query key. Each of these two steps requires $O(\log n)$ work per query and replacing a plane sweep operation requires $O(n)$ such queries. Thus, an AoT can be used in place of a plane sweep and requires $O(\log n)$ time and $O(n \log n)$ work in the CREW PRAM model.

While the AoT structure can be constructed simply and allows for easy parallelization, its primary drawback is the space requirement. At each level of the structure, $O(n)$ elements are stored, so the total structure requires $O(n \log n)$ space. When using an AoT to replace plane sweep operations on a large dataset, the memory requirement may become detrimental to overall performance.

Linear Space Persistent BST. To avoid the $O(n \log n)$ space requirement of the AoT data structure, we consider a more complex data structure. Chazelle and Edelsbrunner [6] present a technique to solve some types of range queries using only $O(n)$ additional space in the word-RAM model [16]. Recall that the AoT data structure, described above, allows querying at any time t by storing $O(n)$ key values (and pointers) at each level. The linear-space persistent BST (LPBST) presented in [6], however, stores only $O(n)$ *bits* at each level, resulting in a total space requirement of $O(\frac{n \log n}{w})$, where w is the number of bits stored in a word. If we assume a constant number of duplicate values, $w = \Theta(\log n)$. Thus, a LPBST structure requires only $O(n)$ space.

The process of constructing a LPBST is similar to that of an array-of-trees. An input of pairs (key k, time t), sorted by key (k) is provided as input. As with an AoT, the LPBST structure can be built bottom-up by merging pairs of elements, resulting in sublists sorted by t. However, unlike AoTs, "nodes" of a LPBST store only a single bit per merged element to identify which child list the element came from. A 0, resp. 1, bit is stored if the element was merged from the left, resp. right list. This merging process is repeated until all $\log n$ levels are merged, resulting in a list sorted by t (and $n \log n$ bits are stored). Figure 7 illustrates an example of the construction of a LPBST. Note that, since each bit within a node represents which subtree (left or right) a particular value came from, the total number of 0 or 1 bits represents the sizes of a nodes' left and right subtrees, respectively.

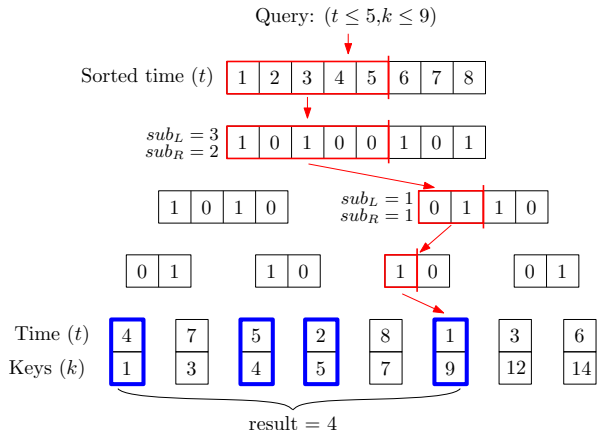


Fig. 8. Example of a query being performed on a LPBST.

A given query (Q_t, Q_k) asks to find the number of elements that have time $t \leq Q_t$ and key $k \leq Q_k$. Therefore, querying involves first finding the *rank* of Q_t in the sorted list of time values ($rank(Q_t)$). Since any entry in the root node with index $i > rank(Q_t)$ cannot match the query, the query we continues down the BST, concerned only with bits of index $i \leq rank(Q_t)$. The query process then obtains the left and right subtree sizes, sub_L and sub_R, by counting the number of bits with 0s and 1s, respectively (with index $i \leq rank(Q_t)$). Since the LPBST is a BST on *keys*, the query process simply traverses the tree while counting bits to determine sub_L and sub_R at each node. These subtree sizes are used count the total number of query matches. Figure 8 provides an example illustrating how a query is performed on a linear space persistent BST.

Computing sub_L and sub_R for a given node is accomplished by performing a prefix sum operation on the bits contained in the node. While scanning a node may take $O(n)$ time, Chazelle and Edelsbrunner [6] reduce the time to compute the prefix sum of a node by storing partial prefix sums every $\log n$ bits within a node. This only requires an additional $O(n)$ space and, using a lookup table to count bits within a word of $\log n$ bits, allows queries to be performed in $O(\log n)$ time. Thus, the LPBST can be used in place of some plane sweep operations and requires $O(\log n)$ time, $O(n \log n)$ work, and $O(n)$ additional space.

4.2 Implementation Details

Aside from our implementation of the Naive algorithm, all of our implementations employ the divide-and-conquer approach described in Section 3. At each recursive level, we perform a total of $O(n \log n)$ work. However, the size of each independent task depends on the recursive level (e.g., at the lowest level, we determine visibility between pairs of vertices). Therefore, at low levels of recursion, our parallel implementations are able to concurrently perform each task without requiring parallelization. At the top level of recursion, however, we have a single task that must be executed in parallel. Thus, our parallel implementations attempt to avoid parallelization overhead by dynamically parallelizing tasks only when necessary at the top levels of recursion.

Both the construction and querying of AoTs and linear space persistent BSTs are similar in many ways. Therefore, our implementations of these structures use many of the same methods. We construct both structures bottom-up by merging pairs of sublists while storing resulting values (or bits). To perform this merging in parallel, we employ the techniques outlined in [18] to merge two lists in $O(\log n)$ time and $O(n)$ work. However, when constructing LPBSTs, efficiently storing

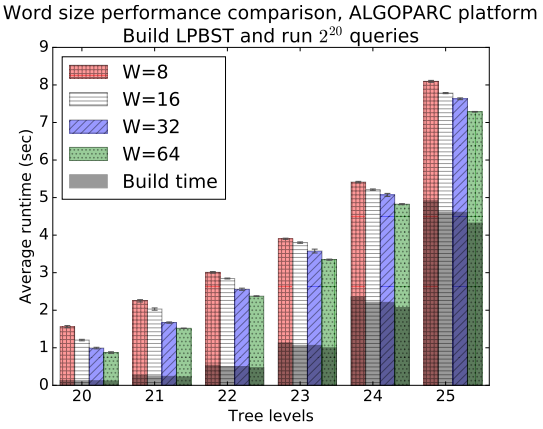


Fig. 9. Average runtime to construct a LPBST and perform 2^{20} queries. The dark shaded portion of each bar indicates construction time, while the remaining time is spent querying.

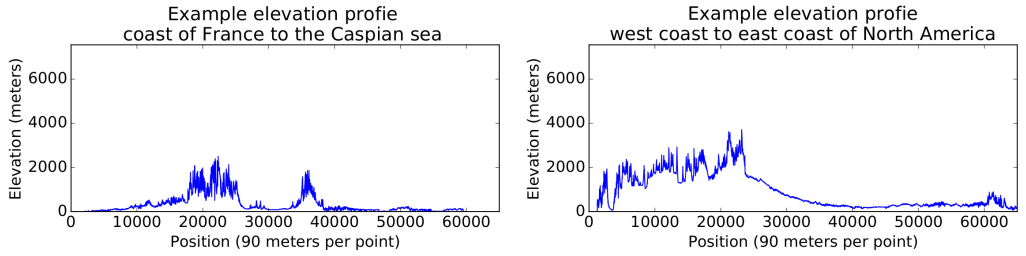
bits requires careful consideration. On modern CPUs, the smallest addressable unit of memory is a byte (8 bits). Thus, if we define each bit independently (e.g., as a boolean datatype), each bit will require 8 bits of storage space. To avoid wasting bits, we use bitwise operations to manually pack bits of data into words of w bits each. We leave w as a parameter and empirically measure the ideal configuration for our hardware platforms.

While the querying process of these two structures is also similar, querying a LPBST requires computing the prefix sum of bits at each node. For our implementation, we store partial prefix sums every w elements. However, while Chazelle and Edelsbrunner [6] use lookup tables to count bits within words of $\log n$ bits, we employ the *popcount* hardware operation. *popcount* is available on our hardware platforms (described in Section 5.1) and returns the number of 1 bits in a word. Since variations of *popcount* are available for words of 8, 16, 32, and 64-bits, our choice of w is limited to these options.

Since our implementation stores a total of $\frac{n}{w}$ partial prefix sum values, our choice of w affects our space requirement. Furthermore, depending on the details of the *popcount* operation,w may impact query performance. To determine the ideal w value for our hardware platforms, we measure the relative query and construction performance while varying w on a range of synthetic, random datasets (see Section 5.2 for details on dataset construction). Figure 9 contains the average runtime to build a LPBST and perform 2^{20} queries on it on the ALGOPARC platform (detailed in Section 5.1). Results indicate that $w = 64$ provides the best performance for our hardware. This is not surprising, since smaller w values require that we store more partial prefix sums. We use $w = 64$ for all experiments hereafter unless otherwise noted. We note that larger w values may further improve performance, but *popcount* is not available for larger word sizes and lookup tables would be far too large to be practical.

5 EXPERIMENTAL RESULTS

In this section we present an empirical evaluation of the performance of our algorithms on synthetic and real-world datasets. We develop five implementations: NAIVE, REDBLUE, SWEEP, PARAOT, and LINPAR. NAIVE employs the $O(n^2)$ algorithm described in Section 3 and is used as baseline. REDBLUE, SWEEP, PARAOT, and LINPAR all use the divide-and-conquer approach presented in Section 3 but they differ in the implementation of the half-line intersection counting step: REDBLUE implements



(a) Example elevation profile from Europe.

(b) Example elevation profile from North America.

Fig. 10. Examples of elevation profiles from 2^{16}-point slices of the Earth dataset (note the different scales).

the Palazzi and Snoeyink [22] algorithm for red-blue line segment intersection counting, Sweep implements the algorithm presented in Section 3 using plane sweep, ParAoT employs the array-of-trees data structure described in Section 4, and LinPar uses the linear space structure, which is also described in Section 4. Asymptotically, all four algorithms achieve $O(n \log^2 n)$ sequential running time. However, RedBlue is more complex than our other implementations and has the poorest performance in practice among our non-naive implementations. While all five implementations run sequentially, ParAoT and LinPar can also run in parallel mode, using multiple threads to improve performance. Though they are both amenable to parallelization, ParAoT requires more memory, while LinPar is more complex and relies on the hardware-specific *popcount* operation.

5.1 Methodology

All algorithms are implemented in C++ and compiled with gcc 4.8.5 using the -Ofast optimization flag. Parallel execution is performed using the openMP library that is included with the gcc compiler. All geometric structures, predicates, and primitives used by all of our algorithms are custom implementations. We use two hardware platforms for our evaluation. The 4-core Algoparc platform is comprised of an Intel Xeon E5-1620 processor (4-core, 3.6 GHz) and 16 GiB of RAM, running the Ubuntu 16.04 operating system. Algoparc has hyperthreading enabled, providing 8 virtual cores. The 20-core Uhhpc platform is comprised of two Intel Xeon E5-2680 processors (10-core, 2.80 GHz), 128 GiB of RAM, and runs the Red Hat Server 6.5 operating system. Note that Uhhpc has 2 CPU sockets, each with 4 memory channels to RAM and an independent L3 cache. All experimental results are averaged over 10 iterations with error bars shown when significant.

5.2 Datasets

We evaluate our algorithm implementations on three synthetic datasets. We consider a *flat* dataset in which all points' elevations are set to $h_i = 1$, so that each point can only see its (at most two) neighboring points. For this dataset, RedBlue, Sweep, ParAoT, and LinPar compute few intersections at each level of recursion, and thus provides a simple correctness case and a performance baseline. We consider a *parabolic* dataset in which each point's elevation is set to $h_i = i^2$, so that every point can see every other point. For this dataset our four recursive implementations compute many intersections at each level of recursion. Finally, we consider *Random* datasets in which point elevations are uniformly sampled from the range $[1,10^6]$.

We also perform evaluations on datasets generated from real-world terrain maps. The CGIAR-CSI Global-Aridity and Global-PET Database [27, 28] consists of elevation data for the entire earth with 90-meter resolution. We extract 1-dimensional slices from 4 different regions: Europe, Asia, Africa,

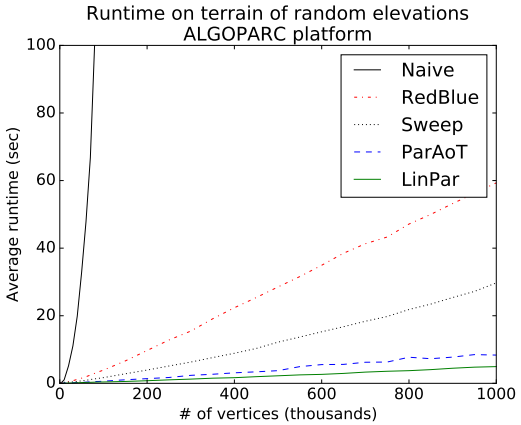
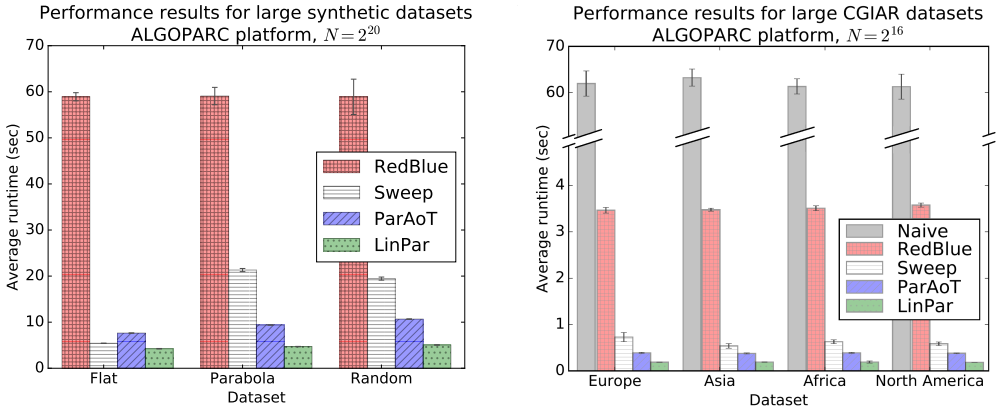


Fig. 11. Results with all five implementations for the Random dataset.



(a) Performance on synthetic datasets ($n = 2^{20}$). (b) Performance on real-world datasets ($n = 2^{16}$).

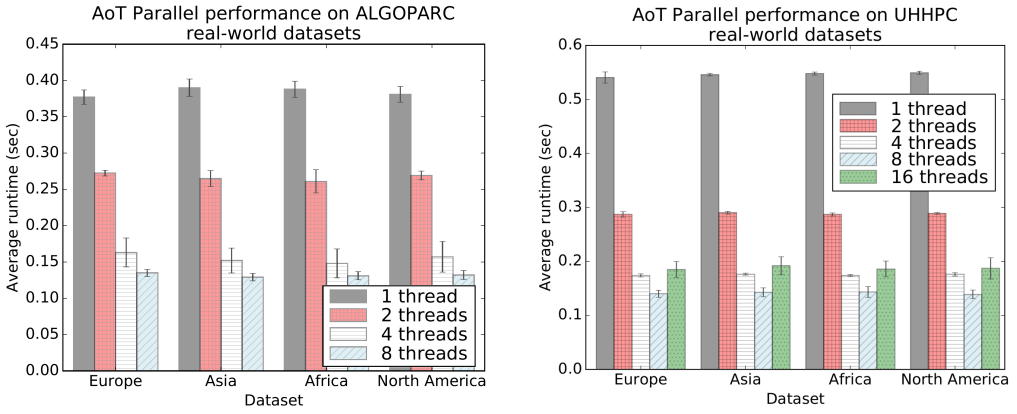
Fig. 12. Sequential Performance of our five implementations.

and North America. Each slice consists of 2^{16} points (spanning ~5000 km). For each of the four regions, we extract ten East-West slices at 1 km North-South intervals. These slices lead to diverse elevation maps, as seen in Figure 10.

5.3 Sequential Performance Results

We evaluate our sequential implementations on the ALGOPARC platform. Figure 11 shows average runtime vs. dataset size (n) for synthetic random datasets. As expected, the quadratic complexity of NAIVE results in much sharper runtime growth compared to the $O(n \log^2 n)$ algorithms. Additionally, we see that the simplified half-line intersection counting algorithm described in Section 3.1 gives SWEEP, PARAOT, and LINPAR a significant practical performance advantage over REDBLUE.

Figure 12a shows average runtimes of our four sub-quadratic implementations for our three classes of synthetic datasets of $n = 2^{20}$ vertices (we omit NAIVE results since its runtime is prohibitive for such a large n). These results confirm that SWEEP, PARAOT, and LINPAR are consistently faster than REDBLUE, with an average decrease in runtime (across all synthetic inputs) of 5.56x, 6.51x, and



(a) ALGOPARC platform. (b) UHHPC platform.

Fig. 13. Parallel performance of the PARAOT implementation on real-world datasets for varying number of compute threads, on each of our two hardware platforms.

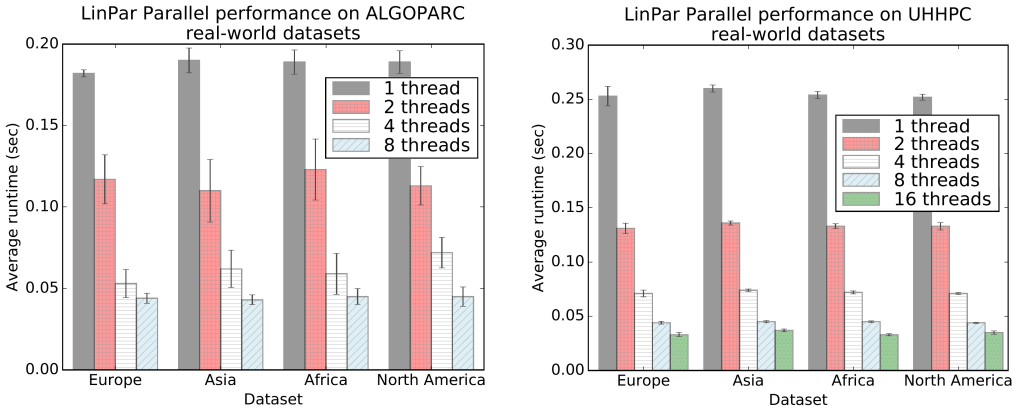
12.70x, respectively. Figure 12a further reveals that SWEEP has a significant variance in execution time across different synthetic datasets, indicating that the overhead of maintaining and balancing a large BST during the plane sweep has a major impact on algorithm performance. The performance of PARAOT and LINPAR, however, are not as dependent on the dataset, and they therefore outperform SWEEP on all but the *flat* synthetic datasets. LINPAR is our fastest implementation on all synthetic datasets, decreasing runtime over PARAOT by 1.79x, 1.99x, and 2.11x on flat, parabolic, and random inputs, respectively.

Figure 12b shows runtimes for each algorithm when applied to data from each region of our real-world dataset, averaged over all 10 slices. As with synthetic datasets, SWEEP, PARAOT, and LINPAR greatly outperform REDBLUE with an average decrease in runtime of 5.72x, 8.25x, and 18.69x, respectively. We conclude that our simplified half-line intersection algorithm provides a significant performance improvement over the general red-blue line segment intersection counting algorithm [22] used by REDBLUE. Furthermore, even sequentially, PARAOT and LINPAR are faster and more consistent that SWEEP, indicating that the AoT and LPBST data structures provide an effective alternative to plane sweep for this problem. Additionally, LINPAR has a significant performance advantage over all of our other implementations, indicating that the reduced memory usage of LINPAR provides practical performance gains.

5.4 Parallel Performance Results

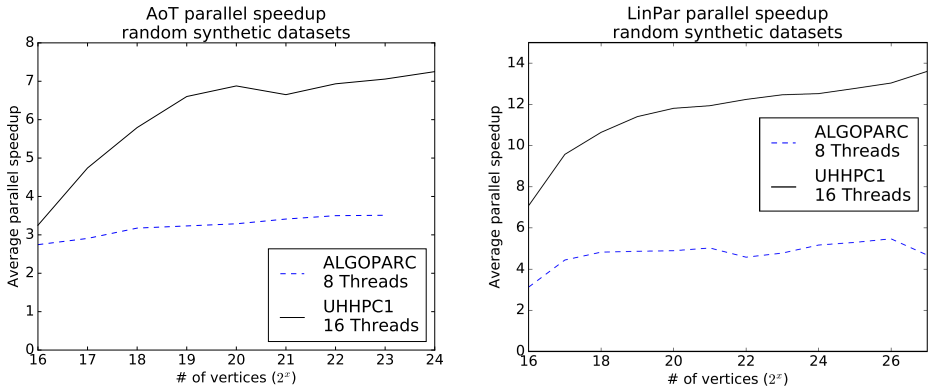
In addition to providing the performance benefits seen in the results above, the AoT and LPBST structures are amenable to parallelization. In this section, we evaluate the performance of our parallel implementations of PARAOT and LINPAR.

To assess parallel performance from a practical standpoint, we present results obtained using our real-world datasets. Figure 13 and Figure 14 show the average runtime of PARAOT and LINPAR, respectively, using our real-world datasets for various numbers of threads on our two hardware platforms. While parallelization provides some performance improvement, the speedup is far from the peak, especially as the number of threads increases. On the 4-core (8 virtual cores due to hyperthreading) ALGOPARC platform, PARAOT and LINPAR achieve a maximum speedup of 2.92 and 4.24, respectively, with 8 threads. On the UHHPC platform, however, PARAOT and LINPAR achieve



(a) ALGOPARC platform (b) UHHPC platform

Fig. 14. Parallel performance of the LINPAR implementation on real-world datasets for varying number of compute threads, on each of our two hardware platforms.



(a) PARAOT parallel speedup. (b) LINPAR parallel speedup.

Fig. 15. Average parallel speedup obtained on random inputs of varying size. Note that axis scales differ.

a maximum parallel speedup of 3.86 and 7.39, respectively. Furthermore, while LINPAR achieves its maximum speedup with 16 threads, PARAOT achieves its maximum parallel speedup with only 8 threads and *slows down* when using 16 threads. We note that our real-world datasets contain only 2^{16} data points. On such small datasets, the affect of the cache system may significantly impact parallel performance. Both of our hardware platforms have L3 caches that can store more than 2^{16} elements (ALGOPARC and UHHPC have L3 caches of 10MB and 25MB, respectively). This results in very fast sequential execution, limiting the achievable parallel speedup and increasing the relative cost of parallel overhead (e.g., spawning new threads). We speculate that the large memory requirement of the AoT data structure further increases the impact of the cache systems on parallel speedup, resulting in the decrease in performance when using 16 threads on the UHHPC platform.

To better understand the cause of the limited parallel performance on real-world datasets, we perform a series of experiments on synthetic random datasets (for which we can vary the size) using the (empirically) best number of threads for each hardware platform. Figure 15 shows the

average parallel speedup vs. data set size on each hardware platform for ParAoT and LinPar. For $n = 2^{16}$ vertices, parallel performance results are similar to our real-world results. As the dataset size increases, however, parallel speedup increases for both implementations, on both hardware platforms. At the largest input size ParAoT can process (due to memory requirements), we see a maximum speedup of 3.51 and 7.25 on Algoparc and Uhhpc, respectively. ParAoT's parallel speedup remains well below expected parallel performance, especially for Uhhpc, where we would expect a speedup nearing 16 for 16 threads. LinPar, however, achieves significantly higher parallel speedup, with a maximum of 5.46 and 13.60 on Algoparc and Uhhpc, respectively. We note that the performance drop seen in Figure 15b when $n = 2^{27}$ on Algoparc is due to limited memory, causing the system to begin swapping to disk. On Uhhpc, however, we have much more available memory, and we see that LinPar continues to gain additional parallel speedup as we increase the input size.

As discussed in Section 4, the array-of-trees data structure requires $O(n \log n)$ memory, while LPBST requires only $O(n)$ additional space. The results in Figure 15 suggest that this memory requirement may be causing ParAoT to be memory bound, resulting in a memory bandwidth bottleneck that limits parallel speedup. This is supported by the fact that the parallel speedup obtained by ParAoT on each hardware platform corresponds to the number of available memory channels. Algoparc, while running 8 hardware threads, is limited by 4 memory channels and achieves a maximum speedup of 3.51. Uhhpc has 8 memory channels (4 per socket) and achieves a maximum speedup of 7.25, despite using 16 hardware threads. We speculate that LinPar, requiring only linear additional space, does not suffer from this memory bottleneck and is therefore able to achieve much higher parallel speedup. We conclude that LinPar is our fastest implementation, both sequentially and in parallel.

6 CONCLUSIONS

In this work, we presented an $O(n \log^2 n)$ algorithm to solve the *1D total visibility-index* problem. Our divide-and-conquer approach uses dualization to reduce the problem to $\log n$ instances of the red-blue line segment intersection counting problem, each of which can be solved in $O(n \log n)$ time. To the best of our knowledge, this is the first subquadratic-time algorithm to solve this problem.

We implemented and four versions of this algorithm and evaluated their performance on two distinct hardware platforms. Each of our implementations solves the red-blue line segment intersection counting problem differently: RedBlue relies on an existing general-case solution, Sweep uses a plane sweep algorithm, and ParAoT and LinPar employ *persistent* search tree data structures. While all four implementations have $O(n \log^2 n)$ asymptotic runtime and are at least an order of magnitude faster than the naive $O(n^2)$ solution, their relative performance differs greatly. Empirical results show that RedBlue is, on average, at least 5 times slower than our other three implementations, indicating that our special-case red-blue line segment intersection counting technique provides significant performance gains. Furthermore, our two implementations that rely on persistent data structures out-perform our plane sweep implementation on most synthetic and all real-world datasets.

In addition to sequential performance gains over other implementations, ParAoT and LinPar can leverage multiple threads to further improve performance. While both implementations achieve parallel speedup on both of our hardware platforms, ParAoT's performance gains are limited due to its large memory requirement. LinPar, however, requires only $O(n)$ extra space and, therefore, achieves up to 85% parallel efficiency on large synthetic datasets. Our fastest implementation finds the 1D total visibility-index of over 100 million vertices in under 3 minutes using 16 threads.

An interesting open problem is to determine whether the dualization used in our solution can be applied to the *2D total visibility-index* computation to achieve a subquadratic solution on two-dimensional terrains. Another interesting avenue for future research is to see if our solution can be applied for faster *approximate* solutions to the *2D total visibility-index* problem by computing total visibility-index on a number of 1D slices of the 2D terrain and then using interpolation to approximate visibility indices to all points in the 2D terrain. Finally, this work indicates that persistent data structures can be leveraged to improve the performance of algorithms that use plane sweep approaches. A practical direction for future research is to apply these persistent structures to improve the many existing algorithms that use plane sweep approaches.

REFERENCES

[1] ArcGIS. http://www.esri.com/software/arcgis, 2016.

[2] GRASS (Geographic Resources Analysis Support System). https://grass.osgeo.org, 2016.

[3] M.J. Atallah, M.T. Goodrich, and S.R. Kosaraju. Parallel algorithms for evaluating sequences of set-manipulation operations. *J. ACM*, 41(6):1049–1088, 1994.

[4] B. Ben-Moshe, O. Hall-Holt, M.J. Katz, and J.S.B. Mitchell. Computing the visibility graph of points within a polygon. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 27–35, 2004.

[5] F. Chao, Y. Chongjun, C. Zhuo, Y. Xiaojing, and G Hantao. Parallel algorithm for viewshed analysis on a modern GPU. *International Journal of Digital Earth*, 4(6):471–486, 2011.

[6] B. Chazelle and H. Edelsbrunner. Linear space data structures for two types of range search. *Discrete Comput. Geom.*, 2(2):113–126, June 1987.

[7] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 3rd edition, 2008.

[8] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 109–121, 1986.

[9] C. Ferreira, M. V. Andrade, S. V. Magalhaes, W. R. Franklin, and G. C. Pena. A parallel sweep line algorithm for visibility computation. In *Proc. of GeoInfo*, pages 85–96, 2013.

[10] C.R. Ferreira, S.V.G. Magalhaes, M.V.A. Andrade, W.R.Franklin, and A.M. Pompermayer. More efficient terrain viewshed computation on massive datasets using external memory. In *Proc. of the 20th International Conference on Advances in Geographic Information System*, pages 494–497, 2012.

[11] J. Fishman, H. Haverkort, and L. Toma. Improved visibility computation on massive grid terrains. In *Proc. of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 121–130, 2009.

[12] L. De Floriani and P. Magillo. Algorithms for visibility computation on terrains: a survey. *Environment and Planning B: Planning and Design*, 30(5):709–728, 2003.

[13] W. R. Franklin and C. K. Ray. Higher isnfit necessarily better: Visibility algorithms and experiments. In *Proc. of Advances in GIS Research: 6th International Symposium on Spatial Data Handling*, pages 751–770, 1994.

[14] S. Friedrichs, M. Hemmer, J. King, and C. Schmidt. The continuous 1.5D terrain guarding problem: descretization, optimal solutions, and PTAS. *Journal of Computational Geometry*, 7(1):256–284, 2016.

[15] A. Haas and M. Hemmer. Efficient algorithms and implementations for visibility in 1.5D terrains. In *31st European Workshop on Computational Geometry*, pages 216–219, 2015.

[16] T. Hagerup. Sorting and searching on the word RAM. In *Proc. 15th Symposium on Theoretical Aspects of Computer Science*, pages 366–398, 1998.

[17] H. Haverkort, L. Toma, and Y. Zhuang. Computing visibility on terrains in external memory. *Journal of Experimental Algorithmics*, 13:5:1.5–5:1.23, 2009.

[18] J. JáJá. *An Introduction to Parallel Algorithms*. Addison Wesley, 1st edition, 1992.

[19] D. B. Kidner, P. J. Rallings, and J. A. Ware. Parallel processing for terrain analysis in GIS: visibility as a case study. *GeoInformatica*, 1(2):183–207, 1996.

[20] M. Llobera, D. Wheatley, J. Steele, S. Cox, and O. Parchment. Calculating the inherent visual structure of a landscape (inherent viewshed) using high-throughput computing. In *Proc. of Beyond the Artifact: Digital Interpretation of the Past: the 32nd Computer Applications and Quantitative Methods in Archaeology conference (CAA)*, pages 146–151, 2004.

[21] M. Löffler, M. Saumell, and R.I. Silveira. A faster algorithm to compute the visibility map of a 1.5 d terrain. In *Proc. 30th European Workshop on Computational Geometry*, 2014.

[22] L. Palazzi and J. Snoeyink. Counting and reporting red/blue segment intersections. *CVGIP*, 56(4):304–310, 1994.

[23] S. Tabik, A. Cervilla, E. Zapata, and L. Romero. Efficient data structure and highly scalable algorithm for total-viewshed computation. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 8(1):1–7, 2014.

[24] M. van Kreveld. Variations on sweep algorithms: Efficient computation of extended viewsheds and class intervals. In *Proc. of the 7th International Symposium on Spatial Data Handling*, pages 13–15, 1996.
[25] D. Wheatley. *Cumulative Viewshed Analysis: a GIS-based method for investigating intervisibility and its archaeological application.* Routlege, London, 1995.
[26] Y. Zhao, A. Padmanabhan, and S Wang. A parallel computing approach to viewshed analysis of large terrain data using graphics processing units. *International Journal of Geographical Information Science*, 27(2):363–384, 2013.
[27] R.J. Zomer, D.A. Bossio, A. Trabucco, L. Yuanjie, D.C. Gupta, and V.P. Singh. Trees and water: Smallholder agroforestry on irrigated lands in northern india. Technical Report 122, International Water Management Institute, Colombo, Sri Lanka, 2007.
[28] R.J. Zomer, A. Trabucco, D.A. Bossio, O. van Straaten, and L.V. Verchot. Climate change mitigation: A spatial analysis of global land sustainability for clean development mechanism afforestation and reforestation. *Agric. Ecosystems and Envir.*, 126:67–80, 2008.

Received May 2017