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

Peyman Afshani* Mark de Berg† Henri Casanova‡ Benjamin Karsin‡
Colin Lambrechts† Nodari Sitchinava‡ Constantinos Tsirogiannis*

Abstract

Let T be a terrain, and let 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 computing the visibility index of *every* point in P. Most 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. While finding a subquadratic solution to this *2D total visibility-index* problem is an open problem, surprisingly, no subquadratic solution has been proposed for the one-dimensional (1D) version of the problem; 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 in the RAM model. 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 three variations of our algorithm: one employing an existing red-blue line segment intersection algorithm and two 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 distinct 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 our new intersection counting implementations achieve more than 8 times speedup over the existing red-blue line segment intersection algorithm. Our parallel implementation is able to process a terrain of 2^{24} vertices in under 1 minute using 16 cores, achieving more than 7 times speedup over serial execution.

1 Introduction

Analyzing terrains to determine locations with special properties is a common objective in Geographic Information Science (GIS). An important property concerns visibility. In particular, one often wants to find points on a terrain that are highly visible or, conversely, points that are hardly visible. Examples include placement of telecommunication towers, placement of fire guard towers, survey of archaeological sites, military logistics, or survey of 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 [11]), with many proposed algorithms [9, 10, 12, 15, 22, 23] 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). Here a cell c' is visible from a cell c if the point on the terrain surface corresponding to the center of c' is visible from the point on the terrain surface corresponding to c.

Let terrain T be a grid with $N = n^2$ total cells. We define the *visibility index* of cell c to be the number of cells in T that are visible from c. The *total visibility-index* problem (also known as cumulative viewshed [24]) is to find the visibility index for every $c \in T$. One way to solve the total visibility-index problem on T is to compute the *viewshed*

*MADALGO, Aarhus University, Denmark, {peyman,constant}@madalgo.au.dk

†Department of Mathematics and Computer Science, TU Eindhoven, The Netherlands, {mdberg,c.lambrechts}@tue.nl

‡Department of Information and Computer Sciences, University of Hawaii at Manoa, USA, {henric,karsin,nodari}@hawaii.edu. These three authors were partially supported by the National Science Foundation under Grant No. 1533823.

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 [23] 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, let alone for modern DEM datasets, which easily consist of hundreds of millions of cells. One way to deal with this is to use a heuristic that approximates the visibility [10, 22]. Another is to observe that computing the viewsheds of different cells can be done independently, and to apply the parallel execution of a large number of single-viewshed computations [6, 8, 17, 18, 25]. Still, such approaches are not suitable for large DEMs. The fundamental problem is that one cannot afford to explicitly compute all visible cells for any cell c of T, as this may produce an output of size $\Omega(N^2)$. Thus we need to leverage the fact that we do not need to compute which cells are visible (the viewshed) but we only need to compute how many cells are visible from each cell, reducing the output size to linear. However, so far no subquadratic algorithms to solve the 2D total visibility-index problem exist.

Thus, the efficient computation of the total visibility-index of a 2D terrain 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.

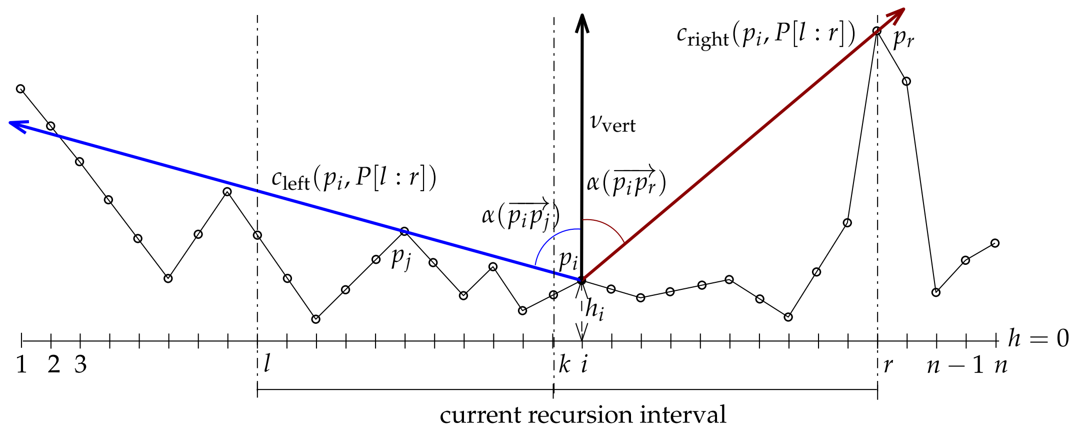
Several previous works have looked a variant of the 1D total visibility-index problem, known as the *1.5D terrain guarding problem* (TGP). The terrain guarding problem involves finding the minimum number of guards needed to view an entire 1-dimensional set of vertices. While similar to the total visibility-index, solving TGP is known to be NP-hard and thus all works provide approximate solutions [13, 14, 19]. This work, however, presents an exact solution to the 1D total visibility-index problem.

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 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 generic solutions. 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 [16], which requires $O(\log^2 n)$ time and $O(n \log^2 n)$ work. We implement a naive $O(n^2)$ implementation, as well as three variations of our algorithm: REDBLUE employs an existing red-blue segment intersection counting algorithm [21], while SWEEP and PARAOT implement two versions of our new intersection counting technique. Furthermore, PARAOT implements the parallel adaptation of our algorithm, allowing it to utilize a variable number of execution threads to improve performance.

We evaluate the performance of our implementations on large synthetic and real-world datasets, showing that all three implementations of our algorithm outperform the naive solution by several orders of magnitude. Additionally, we show that implementations employing our new intersection counting algorithm provide a maximum speedup of 8.25 over the existing solution. We provide a detailed analysis of the performance of our parallel implementation on two hardware platforms. Results indicate that our parallel implementation is capable of processing 2^{24} vertices in 54.15 seconds, providing a maximum parallel speedup of 7.25 using 16 cores.

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 set of these vertices ordered by their x-coordinates, and let $P[l : r]$ denote the subset of vertices $(p_l, \ldots, p_r)$–see Figure 1 for an example. 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 visible from itself, and



$c_{\text{right}}(p_i, P[l:r])$
p_r
ν_{vert}
$c_{\text{left}}(p_i, P[l:r])$
$\alpha(\overrightarrow{p_ip_r})$
$\alpha(\overrightarrow{p_ip_j})$
p_i
p_j
h_i
$h = 0$
1 2 3
l
k i
r
$n-1$ n
current recursion interval

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

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. Note that every vertex is visible to itself and its neighbors and we define the visibility ray between p_i and p_i as pointing vertically down. 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 $l \le i \le r \le n$. 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 $l \le k \le i$. 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 $\overrightarrow{p_ip_t}$ ($i \le t \le r$) 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 $\overline{p_ip_j}$, because c_{right} has the smallest angle. 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 points are strictly *above* critical rays

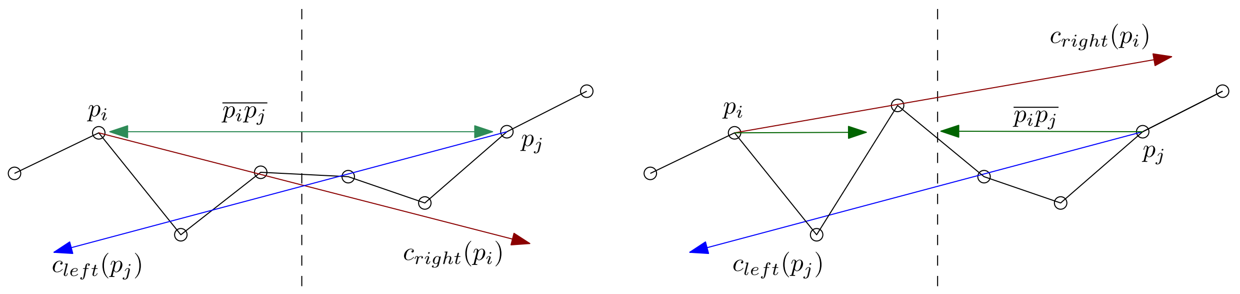


Figure 2: Illustration of the intuition behind Lemma 2.1. **Left:** example where both points are above critical rays. **Right:** example where a point is below a critical ray.

to allow visibility, this is an implementation detail. 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, until we end up with subsets that consist of a single vertex. After computing the total visibility-index for these trivial cases (all vertices are visible to their neighbors), 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 $P[k+1:r]$ which 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 refer to the problem of computing the number of visible vertices only between elements of two distinct subsets $P[l:k]$ and $P[k+1:r]$, as *Bipartite Visibility*. We denote the entire divide-and-conquer algorithm that computes the 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. We next focus on describing an algorithm that solves Bipartite Visibility in $O(n \log n)$ time.

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

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

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 [5]. Instead of handling the actual critical rays of the input points, we project these rays onto a dual plane: 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. For now, we claim that 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 correctly set 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 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 segment intersections.

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 red-blue line segment intersection counting problem. We can thereby solve it using existing methods.

The dual of a point $p_i : (i, h_i)$ is the line $p_i^* : y = ix - h_i$, and the dual of a line $l : y = ax + b$ is 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. Figure 3 illustrates how we apply duality to critical rays.

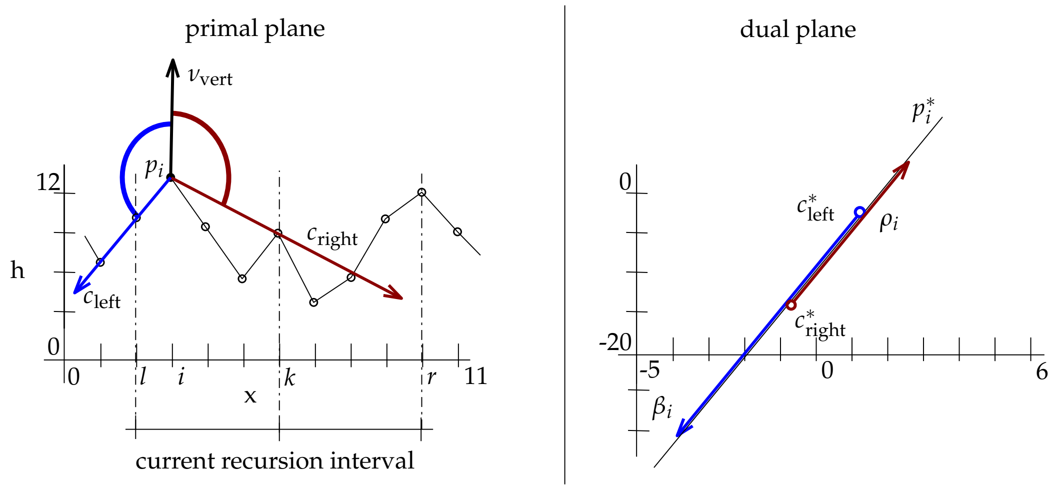


Figure 3: An example of a terrain, its critical rays and their corresponding dual half-lines.

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 $1 \leq i \leq n$). 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 3.1. *Given 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 $c_{left}(p_j, P[k+1:r])$, 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 3.1 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 3.2. *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.*

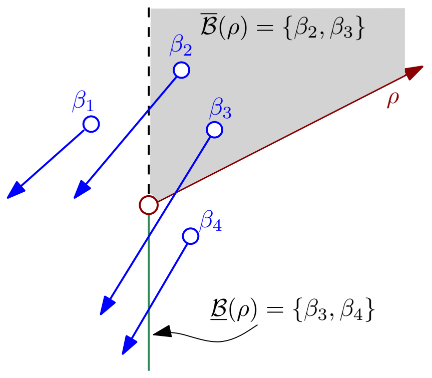
Proof. Suppose for the sake of contradiction that $\rho(p_i, P[l:k])$ and $\rho(p_j, P[l:k])$ do intersect, which means that there is a visibility line $\overline{p_i p_j}$ between the p_i and p_j (in the primal plane). It also means that both $c_{\text{right}}(p_i, P[l:k])$ and $c_{\text{right}}(p_j, P[l:k])$ 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 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 [21] present an algorithm that computes, in $O(n \log n)$ time, the total number of intersections between a set of self-non-intersecting (red) line segments and another set of self-non-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, we can modify their algorithm to produce the desired result without impacting asymptotic performance.

3.2 A Practical Algorithm for Red-blue Intersection Counting

While leveraging the red-blue segment intersection algorithm of Palazzi and Snoeyink [21] provides an $O(n \log n)$ solution to Bipartite Visibility, it ignores some features specific to our problem. Thus, 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 simplify our description, we use the following 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



$\overline{\mathcal{B}}(\rho) = \{\beta_2, \beta_3\}$
β_2
β_1
β_3
ρ
β_4
$\underline{\mathcal{B}}(\rho) = \{\beta_3, \beta_4\}$

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

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 3.3. *Any $\rho \in \mathcal{R}$ 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.*

Proof. Assume $\rho_{x,y}$ 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, $\rho_x < q_x$. 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:

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

By Lemma 3.3, 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, in the order they intersect it. Since blue half-lines do not intersect other blue half-lines and continue to $-\infty$, this is the same order as the reverse relative order of the blue half-lines by their slopes. Thus, every time the sweep line encounters a blue half-line end point $\beta_{x,y}$, we insert β to $\mathcal{T}$ performing comparisons on the negative of their slopes. 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 $\mathcal{T}$, 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.

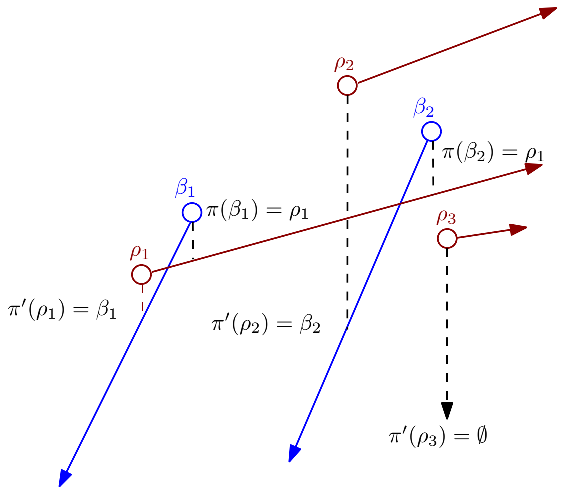


Figure 5: Illustration of $\pi'(\rho)$ and $\pi(\beta)$ for several half-lines.

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, except the sweep is 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, on the other hand 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 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 by the sweep line, 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),

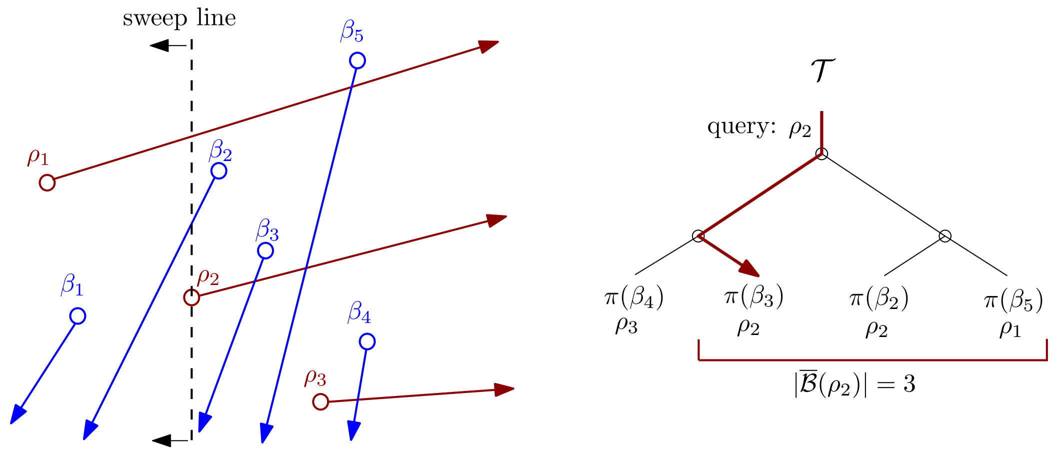


Figure 6: Example of the plane sweep algorithm used to find $|\overline{\mathcal{B}}(\rho)|$. The vertical sweep line moves from right to left and, when a blue endpoint $\beta_{x,y}$ is encountered, $\pi(\beta)$ is added to the search tree $\mathcal{T}$. When the sweep line encounters a red endpoint $\rho_{x,y}$, 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 $|\overline{\mathcal{B}}(\rho)|$.

resulting in $O(n \log n)$ time overall to compute $|\overline{\mathcal{B}}(\rho)|$ for each half-line half-line ρ. Computing $|\overline{\mathcal{R}}(\beta)|$ is symmetric.

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 3.4. *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. Suppose a point p_t, that is not on the upper hull, would determine the critical ray of p_i, then p_t is outside the upper hull. If p_t is inside the upper hull, the visibility line from p_i to p_t is below a point on the upper hull. So, p_t is outside the upper hull and therefore the convex hull is not a valid convex hull. Hence only points on the upper hull can be candidates for the critical ray.

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., $t < k < i$ or $t > k > i$, 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 the critical ray of p_i 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 $c_{\text{right}}(p_i, P[l : r])$.

Thus, to update $c_{\text{left}}(p_i, P[l : r])$ and $c_{\text{right}}(p_i, P[l : r])$ we build the upper convex hulls of $P[l : k]$ and $P[k + 1 : r]$, and for every point in these two subsets we construct the tangent to the hull of the opposite subset. Building these hulls takes $O(n)$ time because points are pre-sorted by x-coordinates. 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 $O(n \log^2 n)$.

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.*

4 Parallel Extension

Persistence [7] is a technique for efficiently maintaining all past versions of a dynamic structure for future queries. A persistent binary search tree supports all standard update operations 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 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.

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 this sweep in $O(\log n)$ parallel 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.

Our parallel implementation replaces each plane sweep operation described in Section 3.1 with the construction and querying of an array-of-trees (AoT). For each AoT construction, we begin 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 we construct the structure bottom-up by a variation of merge sort on t. At each level, we merge pairs of sets 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. We employ a variation of the parallel Mergepath [20] algorithm to construct the AoT in $O(\log n)$ parallel time and $O(n \log n)$ work.

Querying the AoT involves two steps: 1) find the correct root and 2) query the corresponding BST. Since the top level of the AoT is a list sorted by t, we perform a binary search to find the correct BST for the query. We then search the associated BST with the key. Each of these two steps requires $O(\log n)$ work and at each recursive level our algorithm performs $O(n)$ such queries. Thus, these queries take $O(\log n)$ parallel time $O(n \log n)$ work at each level.

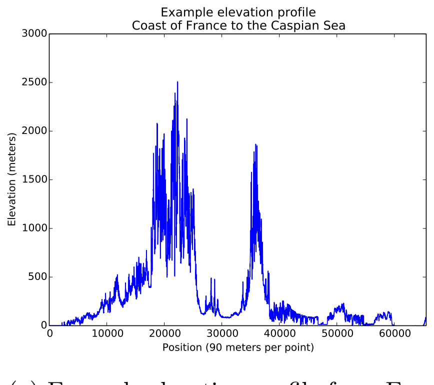
A primary drawback of the AoT structure is its space requirement. At each level of the structure, we must store $O(n)$ elements, so the total structure requires $O(n \log n)$ space. Furthermore, each element stores child pointers and other information (depending on the query function). For large datasets the AoT memory requirement may thus become detrimental to overall performance.

5 Experimental Results

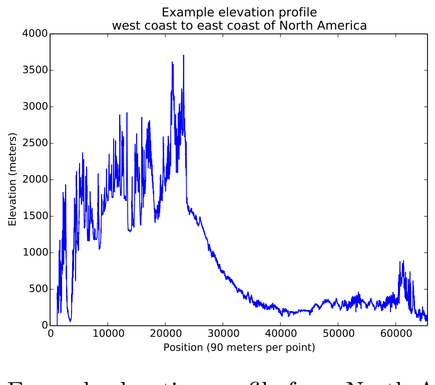
In this section we present an empirical evaluation of the performance of our algorithm on synthetic and real-world datasets. We implement four algorithms: Naive, RedBlue, Sweep, and ParAoT. Naive is the $O(n^2)$ algorithm described in Section 3 and is used as baseline. RedBlue, Sweep, and ParAoT all 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 the Palazzi and Snoeyink [21] algorithm for red-blue line segment intersection counting, Sweep implements the algorithm presented in Section 3 using plane sweep, and ParAoT employs the array-of-trees data structure described in Section 4. Asymptotically, all three algorithms achieve $O(n \log^2 n)$ sequential running time. However, Sweep and ParAoT are much simpler than RedBlue and achieve better performance in practice. Furthermore, ParAoT is amenable to parallelization.

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 comprises of Intel Xeon E5-1620 processor (4-core, 3.6 GHz) and 16 GiB of RAM, running the Ubuntu 14.04 operating system. The 20-core Uhhpc platform comprises 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.

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, and ParAoT compute few intersections at each level of recursion, and thus provides a simple cor-

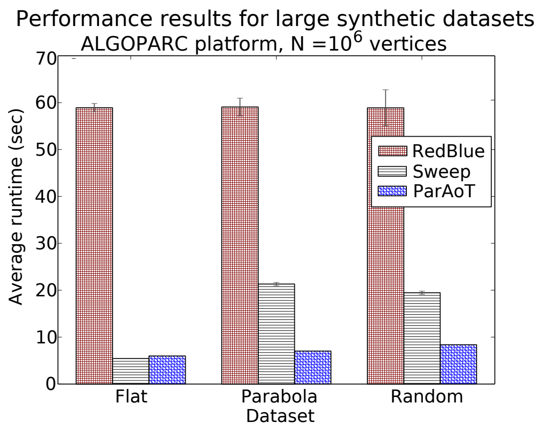


(a) Example elevation profile from Europe.

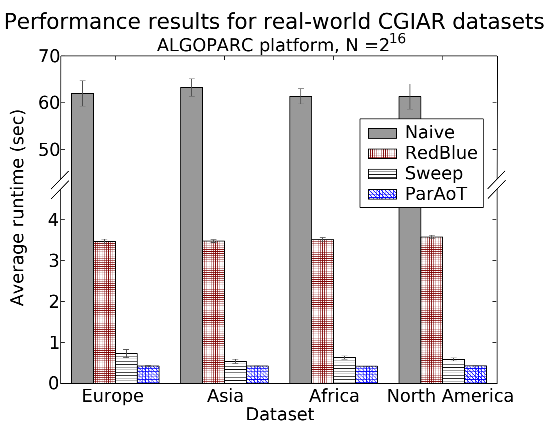


(b) Example elevation profile from North America.

Figure 7: Examples of elevation profiles from 2^{16}-point slices of the Earth dataset (note the different scales).



(a)



(b)

Figure 8: Sequential algorithm performance on synthetic and real-world datasets.

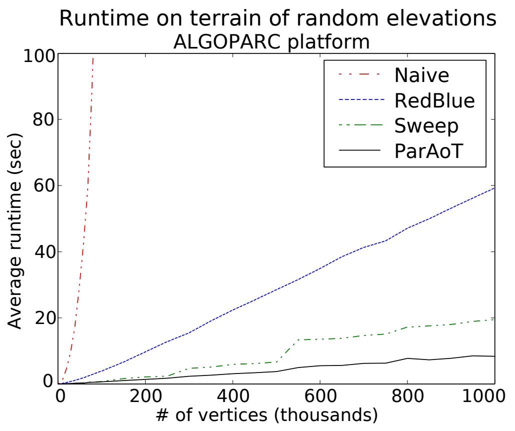


Figure 9: All four algorithms for the Random dataset.

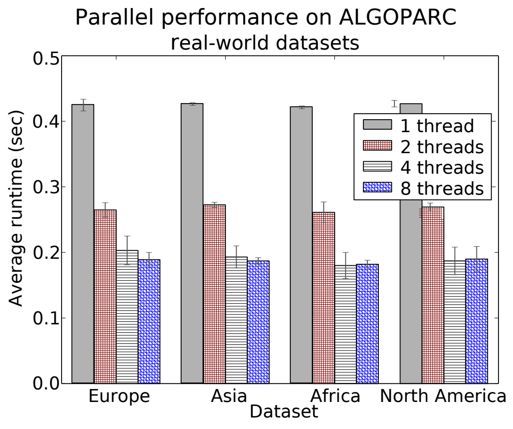
rectness case and a performance baseline. We consider a *parabolic* dataset in which each point's elevation is set to $h_i = i^2$, so that every point can see every other point. For this dataset RedBlue, Sweep, and ParAoT compute many intersections at each level of recursion. Finally, we consider *Random* datasets in which point elevations are uniformly sampled from the range [1,1000].

We also perform evaluations on datasets generated from real-world terrain maps. The CGIAR-CSI Global-Aridity and Global-PET Database [26, 27] consists of elevation data for the entire earth with 90-meter resolution. We extract 1-dimensional slices from 4 different regions: Europe, Asia, Africa, and North America. Each slice consists of 2^{16} points (spanning $\sim$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 7.

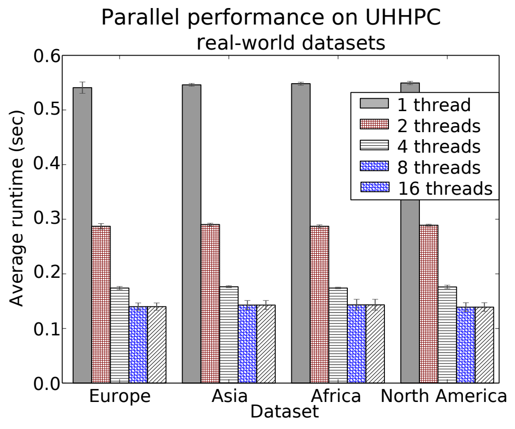
5.3 Sequential Performance Results We evaluate our sequential implementations on the Algoparc platform. Figure 9 shows average runtime vs. dataset size (n) for synthetic random datasets. As expected, the quadratic complexity of Naive results in much sharper runtime growth compared to the $O(n \log^2 n)$ algorithms. Additionally, we see that the simplified half-line intersection counting algorithm described in Section 3.1 gives Sweep and ParAoT a significant practical performance advantage over RedBlue.

Figure 8a shows average runtimes for RedBlue, Sweep, and ParAoT for our three classes of synthetic datasets, for $n = 10^6$ vertices (we omit Naive results since its runtime is prohibitive for such large n). These results confirm that Sweep and ParAoT are consistently faster than RedBlue, with an average speedup of 3.79 and 8.19, respectively. Figure 8a 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 major impact on algorithm performance. The performance of ParAoT, however, is not dependent on the dataset, and therefore outperforms Sweep on all but the *flat* synthetic datasets.

Figure 8b 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, both Sweep and ParAoT greatly outperform RedBlue with an average speedup of 5.65 and 8.25, respectively. We conclude that the half-line intersection algorithm employed by Sweep and ParAoT provides a significant performance im-



(a)



(b)

Figure 10: Parallel performance on real-world datasets for varying number of compute threads, on each of our two hardware platforms.

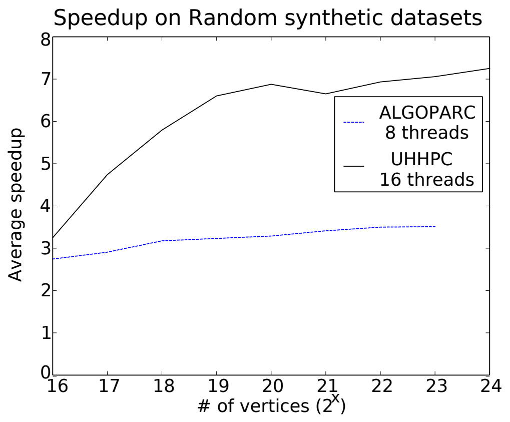


Figure 11: Average parallel speedup obtained on two hardware platforms for varying dataset sizes.

provement over the red-blue intersection counting used by RedBlue [21]. Furthermore, even sequentially, ParAoT is faster and more consistent that Sweep, indicating that the AoT data structure is an effective alternative to plane sweep for our problem.

5.4 Parallel Performance Results Besides the performance benefit of using AoT structure seen in the above results, another advantage is that it is amenable to parallelization. In this section, we evaluate the performance of our parallel implementation of ParAoT. While ParAoT employs parallel AoT construction and querying, the convex hull computations are performed sequentially. At lower levels of recursion, we can create and query convex hulls concurrently. However, at the top levels of recursion there are fewer hulls and computation is serialized. While parallelization of this step may improve speedup, we determine that other factors limit overall parallel performance. Thus, we leave parallelization of individual convex hull creation for future work.

To assess parallel performance from a practical standpoint, we present results obtained on our real-word dataset. Figure 10a and 10b show the average runtime of ParAoT on our real-world datasets, for varying numbers of threads, on both our hardware platforms. While parallelization provides significant speedup the benefits are limited, especially as the total number of threads increases (note that Algoparc has 4 cores but 8 hardware threads due to hyper-threading). Overall, the parallel implementation achieves a maximum speedup of 2.44 and 3.86 on Algoparc and Uhhpc, respectively. We note that our real-world datasets contain 2^{16} data points, which may not be sufficiently large to make the parallelization overhead costs negligible.

To understand the cause of the poor parallel performance we perform a series of experiments on synthetic random datasets (for which we can vary the size). Figure 11 shows average speedup vs. data set size, using the best (empirically) number of threads for each hardware platform. For 2^{16} points we see similar parallel speedup close to 3.00 on both hardware platforms. As the dataset size increases parallel speedup also increases on both hardware platforms. At the largest input size (limited by memory), we see a maximum speedup of 3.51 and 7.25 on Algoparc and Uhhpc, respectively. Parallel speedup remains well below peak parallel performance, especially for Uhhpc where we would expect a speedup nearing 16 for 16 threads.

One drawback of the array-of-trees data structure is the large $O(n \log n)$ memory requirement. Furthermore, the querying of the AoT involves random memory accesses that may lead to poor cache utilization. Therefore, we conjecture that ParAoT is memory bound, thus causing memory bandwidth to be a bottleneck limiting parallel speedup. This is supported by the fact that the parallel speedup we obtain 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.

Our parallel results indicate that, while the AoT data structure allows for efficient parallelization of the half-line intersection computation, it creates a memory bottleneck that limits parallel performance. Reducing the memory requirement of the AoT or improving the memory access pattern of AoT queries may alleviate this issue, which we will consider in future work.

6 Conclusions

We presented an $O(n \log^2 n)$ algorithm for the *1D total visibility-index* problem. To the best of our knowledge it is the first subquadratic-time algorithm for this problem. We implemented three versions of this algorithm and, not surprisingly, all implementations are much faster than the existing quadratic solution. Among the three versions, our new *red-blue half-line intersection counting* solution provides significantly better performance. Additionally, we presented a parallelization of this intersection counting solution using the *array-of-trees* data structure, allowing us to solve the 1D total visibility problem in $O(\log^2 n)$ time and $O(n \log^2 n)$ work in the CREW PRAM model.

Empirical results presented for two hardware platforms indicate that, even when running sequentially, our array-of-trees implementation outperforms the simple plane sweep method on most synthetic and real-world experiments. Furthermore, our array-of-trees implementation provides significant parallel performance gain when leveraging multiple threads. Our results indicate that our parallel speedup may be bound by memory bandwidth, and we leave further investigation, as well as further optimization, for future work.

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.

References

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

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

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

[4] Boaz Ben-Moshe, Olaf Hall-Holt, Matthew J. Katz, and Joseph 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] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[6] 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):471486, 2011.

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

[8] 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*, page 8596, 2013.

[9] 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*, page 494497, 2012.

[10] 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*, page 121130, 2009.

[11] L. De Floriani and P. Magillo. Algorithms for Visibility Computation on Terrains: a Survey. *Environment and Planning B: Planning and Design*, 30(5):709728, 2003.

[12] W. R. Franklin and C. K. Ray. Higher isnt Necessarily Better: Visibility Algorithms and Experiments.

In *Proc. of Advances in GIS Research: 6th International Symposium on Spatial Data Handling*, page 751 770, 1994.

[13] 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.

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

[15] 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.

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

[17] 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):183207, 1996.

[18] 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.

[19] Maarten Löffler, Maria Saumell, and Rodrigo 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.

[20] S. Odeh, O. Green, Z. Mwassi, O. Shmueli, and Y. Birk. Merge path - parallel merging made simple. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1611–1618, 2012.

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

[22] 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):17, 2014.

[23] 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*, page 1315, 1996.

[24] D. Wheatley. *Cumulative Viewshed Analysis: a GIS-based method for investigating intervisibility and its archaeological application.* Routlege, London, 1995.

[25] 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):363384, 2013.

[26] RJ. Zomer, DA. Bossio, A. Trabucco, L. Yuanjie, DC. Gupta, and VP. Singh. Trees and water: Smallholder agroforestry on irrigated lands in northern india. Technical Report 122, International Water Management Institute, Colombo, Sri Lanka, 2007.

[27] RJ. Zomer, A. Trabucco, DA. Bossio, O. van Straaten, and LV. 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.