Computing Visibility on Terrains in External Memory

HERMAN HAVERKORT
Technische Universiteit Eindhoven
and
LAURA TOMA and YI ZHUANG
Bowdoin College

Given an arbitrary viewpoint v and a terrain, the visibility map or viewshed of v is the set of points in the terrain that are visible from v. In this article we consider the problem of computing the viewshed of a point on a very large grid terrain in external memory. We describe algorithms for this problem in the cache-aware and cache-oblivious models, together with an implementation and an experimental evaluation. Our algorithms are a novel application of the distribution sweeping technique and use $O(sort(n))$ I/Os, where $sort(n)$ is the complexity of sorting n items of data in the I/O-model. The experimental results demonstrate that our algorithm scales up and performs significantly better than the traditional internal-memory plane sweep algorithm and can compute visibility for terrains of 1.1 billion points in less than 4 hours on a low-cost machine compared to more than 32 hours with the internal-memory algorithm.

Categories and Subject Descriptors: B.4.4 [**Input/Output and Data Communications**]: Performance Analysis and Design Aids; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Geometrical problems and computations*

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Computational geometry, data structures and algorithms, digital elevation models, I/O-efficiency, terrains, visibility

L. Toma was supported by NSF award 0728780.
A preliminary version of this work appeared in *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments/Workshop on Analytic Algorithms and Combinatorics (ALENEX/ANALCO 2007)*.
Authors' addresses: H. Haverkort, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands; email: cs.herman@haverkort.net; L. Toma and Y. Zhuang, Department of Computer Science, Bowdoin College, 8650 College Station, Brunswick, ME 04011, USA; email: {ltoma, yzhuang}@bowdoin.edu.

ACM Reference Format:
Haverkort, H., Toma, L., and Zhuang, Y. 2008. Computing visibility on terrains in external memory. ACM J. Exp. Algor. 13, Article 1.5 (November 2008), 23 pages. DOI = 10.1145/1412228.1412233 http://doi.acm.org/10.1145/1412228.1412233

1. INTRODUCTION

In this article, we consider the problem of computing visibility on very large terrains in external memory. Given an arbitrary viewpoint v and a terrain, the basic problem we address is computing the *visibility map* or *viewshed* of v, which is the set of points in the terrain that are visible from v (Figure 1). Visibility has applications in graphics and game design, and mainly in geographic information systems (GIS), ranging from path planning, navigation, landscaping, to placement of fire towers, radar sites and cellphone towers [Franklin and Ray 1994; de Floriani and Magillo 1994].

Visibility has been widely studied in computational geometry and graphics; for a survey of the various problems and results see Cole and Sharir [1989] or De Floriani and Magillo [1999]. Recently, researchers have obtained a number of results on visibility addressing the watchtower problem and terrain guarding [Agarwal et al. 2005; Ben-Moshe et al. 2005]. The standard terrain model used in geometry is the *polyhedral terrain*, which is a continuous piecewise linear function defined over the triangles of a triangulation in the plane. In GIS, however, the most common representation of terrain data is the grid, which samples the elevation of a terrain with a uniform grid and records the values in a 2D-matrix. Thus, we are interested in computing visibility on grid terrains.

In recent years, an increasing number of applications involve high-resolution massive terrain data that is becoming available from remote sensing technology. NASA's Shuttle Radar Topography Mission (SRTM) acquired terrain data with one sample per 30m ($900m^2$) in 2002, in total about 10 terabytes of data. More recently, LIDAR and real-time kinematic global positioning system technology offer the capability to collect geospatial data at 1m-resolution.

When working with massive data, only a fraction of the data can be held in the main memory of a computer. Thus, the transfer of data between main memory and disk, rather than the computation as such, is usually the performance bottleneck. One approach to improving performance is to design *external memory* or *I/O-efficient algorithms*—algorithms that specifically optimize the number of block transfers between main memory and disk. In this article, we present the design and experimental evaluation of an I/O-efficient algorithm for viewshed computation on very large grid terrains.

1.1 Problem Definition

Let T be a terrain represented as a grid of n square cells. For computational purposes, we assume that the entire grid cell is represented by its center point. In particular, we assume that we are given the elevation of each grid cell's center point. The *line of sight* from an arbitrary viewpoint v (not necessarily a centerpoint) to a grid cell Q is the line segment that connects v to the center

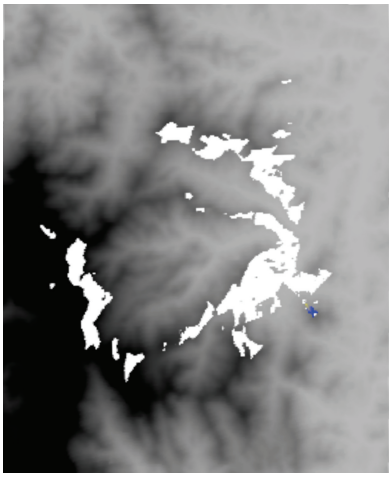


Fig. 1. The viewshed of a point on a terrain is shown in white.

q of Q. The *height above the horizon* of Q with respect to the viewpoint v is defined as the height above the horizon of q, which is

$$\text{height}_q = \arctan \frac{e_q - e_v}{d_{vq}}$$

where e_q and e_v are the elevations of q and v, respectively, and d_{vq} is the distance between q and v. Note that this means a cell is treated as having constant height above the horizon with respect to the viewpoint.

We use the definition of visibility also applied by Van Kreveld [1996]: A grid cell with center q is defined as visible from v if the line of sight vq does not cross any grid cell that appears higher on the horizon—more precisely, if the line segment vq does not cross any grid cell with center q' such that the slope of vq' is higher than the slope of vq. Note that this is a discrete visibility model, where a cell is either completely visible or not. There is no concept of partial visibility: A corner of a high cell is assumed to hide another grid cell Q from the viewpoint v completely if it intersects the line of sight from v to the center q of Q. And conversely, a (corner of a) high cell may intersect the line to the corner of another cell without this being counted as occlusion—it is the line to the *center* not the *corner* of the cell that counts.

Let v be an arbitrary viewpoint. With visibility defined as above, the *visibility map* or *viewshed* of v on a *grid* terrain is the set of all grid cells that are visible from v. In GIS, one is not only interested in computing boolean cell visibility, but also, for cells that are invisible from v, their vertical distance to visibility: how far the cells should be raised to become visible from v. Thus, for a given viewpoint, we want to compute a grid that records the distance of each cell to visibility. We call this the *visibility grid* of v. The viewshed of v is the set of cells with distance 0.

A straightforward approach to determine whether a cell is visible from a given viewpoint requires $O(n)$ time per cell, or $O(n^2)$ time for the entire grid—if

the grid is square the bounds become $O(\sqrt{n})$ and $O(n\sqrt{n})$ respectively. The bound for the entire grid was improved to $O(n \lg n)$ by Van Kreveld [1996] using plane sweeping. There have been many other articles from the GIS community dealing with visibility computation on grids; see for example the articles by Fisher [1993, 1994], Franklin [2002], and Franklin and Ray [1994], and the references therein. They describe experimental studies for fast implementations of approximate visibility computations and explore various trade-offs between speed and accuracy; they do not guarantee worst-case bounds better than the straightforward one, nor do they prove any bounds on the quality of the approximation. An overview of the visibility results on grids, as well as on other terrain representations (triangulations), can be found in the work by De Floriani and Magillo [1999, 1994]. To our knowledge, no I/O-efficient results have been reported for visibility computations.

1.2 I/O-Model and Related I/O-Work

We use the standard two-level I/O-model by Aggarwal and Vitter [1988]. The model defines two parameters: M is the size of internal memory, and B is the size of a disk block. An input/output (or: *I/O*) is the operation of transferring a block of data between main memory and disk. The *I/O-complexity* of an algorithm is the number of I/Os it performs. The basic bounds in the I/O-model are those for scanning and sorting. The *scanning bound*, $scan(n) = \Theta(\frac{n}{B})$ is the number of I/Os necessary to read n contiguous items from disk. The *sorting bound*, $sort(n) = \Theta(\frac{n}{B} \log_{M/B} \frac{n}{B})$ represents the number of I/Os required to sort n contiguous items on disk [Aggarwal and Vitter 1988]. For all realistic values of n, B, and M, $scan(n) < sort(n) \ll n$.

I/O-efficient algorithms have been developed for many problems encountered in GIS, like variants of segment intersection and range searching. The first results were obtained by Goodrich et al. [1993], who developed the technique of *distribution sweeping* as an external memory version of the powerful plane sweep paradigm in internal memory. Distribution sweeping was developed for the problem of orthogonal line segment intersection, and has been subsequently applied to other GIS problems, like the (red-blue) line segment intersection and map overlay [Arge et al. 1995]. For a survey of distribution sweeping and I/O-efficient algorithms for GIS see Arge [1997] and Van Kreveld et al. [1997].

The *cache-oblivious* model was introduced by Frigo et al. [1999]. It aims to analyze computation using the entire memory hierarchy of a computer, which typically includes several levels of cache, main memory and disk. Cache-oblivious algorithms (as opposed to algorithms in the I/O-model which are *cache-aware*) are algorithms that are analyzed in the I/O-model, but B and M, the block and memory sizes, are not known to the algorithm. Cache-oblivious algorithms optimize for any B and M and, as a result, are efficient at any level of the memory hierarchy. The parameters of the memory hierarchy do not need to be known, which makes cache-oblivious algorithms highly portable in theory. Despite the many theoretical results in the cache-oblivious model, practical evaluations of cache-oblivious algorithms and comparison with RAM-, cache- and

I/O-efficient algorithms are still in an early stage. For the basic problem of sorting, evidence suggests that cache-oblivious sorting is at least as fast as RAM-optimized sorting and cache-aware sorting, but slower than customized I/O-efficient sorting [Brodal et al. 2007]. For further details on cache-obliviousness and surveys of the results obtained in this model, see Frigo et al. [1999], Brodal [2004], Arge et al. [2005].

1.3 Our Results

In this article, we show a novel application of the distribution sweeping technique to the computation of the viewshed of a point on a grid terrain. Our algorithm, described in Section 3, runs in $O(sort(n))$ I/Os in the I/O-model and is based on the plane sweep algorithm by Van Kreveld [1996]. In Section 4, we show that the same bound can be obtained in the cache-oblivious model. The idea is to view the sweeping as a bottom-up binary merging process as in Brodal and Fagerberg [2002] and use Funnelsort [Frigo et al. 1999].

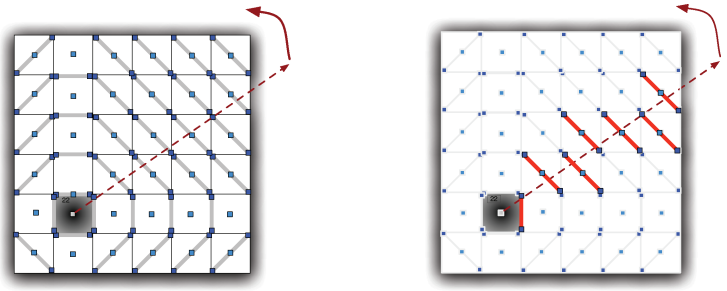
In Section 5, we describe an experimental evaluation of computing viewsheds in external memory on real-life terrains. We compare our I/O-efficient algorithm, `ioviewshed`, with the traditional internal-memory plane sweep algorithm, `kreveld`, and with a module that computes viewsheds in `GRASS`, the most commonly used open-source GIS. On large datasets `ioviewshed` performs significantly better than the internal-memory algorithms and can compute visibility on up to 1.1 billion-point terrains in less than 4 hours, on average. The only I/O-components used in the implementation of `ioviewshed` are scanning and sorting, and we expect that the performance can be further improved by fine-tuning the I/O-library.

2. VAN KREVELD'S ALGORITHM

We start by describing the $O(n \lg n)$ plane sweep algorithm for computing the visibility grid of a point by Van Kreveld [1996]. This is the best-known upper bound for the problem in internal memory.

Given a grid and a viewpoint v, the basic idea is to rotate a (half) line around v and compute the visibility of each cell in the terrain when the sweep-line passes over its center. For this, we maintain a data structure (the *active* structure) that, at any time in the process, contains the cells currently intersected by the sweep line (the *active cells*); refer to Figure 2(b). When a cell starts being intersected by the sweep-line, it is inserted in the active structure; when a cell stops being intersected by the sweep-line, it is deleted from the active structure. When the center of a cell is intersected by the sweep line, the active structure is queried to find out if that cell is visible. Thus, each cell in the grid has three associated *events*: when it is first intersected by the sweep-line and entered in our data structure, when the sweep-line passes over its center, and when it is last intersected by the sweep-line and removed from our data structure; refer to Figure 2(a).

Recall that a cell is defined to be visible if, when the sweep-line passes over its center, there are no active cells closer to the viewpoint and with greater height above the horizon. To query the active cells efficiently, Van Kreveld [1996]



(a) Each cell is marked with its 3 events. (b) Active cells.

Fig. 2. Van Kreveld's plane sweep algorithm.

uses a balanced binary search tree for the active structure in which the active cells are stored from left to right in order of increasing distance from their centers to the viewpoint. In addition, each node in the tree is augmented with the greatest height above the horizon in the subtree rooted at that node; for leaves, this is simply the height above the horizon of the cell stored at the leaf.

The augmented binary search tree can be maintained in $O(\lg n)$ time per insertion and deletion in a straightforward way. To determine whether a cell Q is visible, we search for it in the tree. As we follow the search path, all the cells that could obstruct visibility are to the left of the path; to find the cell among them that appears highest above the horizon, we collect the maximum height stored in all the subtrees to the left of the path in $O(\lg n)$ time. From here we can infer whether Q is visible, and, if not, what is its vertical distance to visibility. Overall, there are $3n$ events, and each is handled in $O(\lg n)$ time. If we wanted to compute only the viewshed of v, the algorithm would still take $\Theta(n \lg n)$ time: Irrespective of the size of the viewshed, we would still sort the events and traverse the entire grid. We have the following.

THEOREM 2.1. *(van Kreveld van Kreveld [1996]) The visibility grid of an arbitrary viewpoint can be computed in $O(n \lg n)$ time for a grid of size n.*

3. COMPUTING VISIBILITY IN EXTERNAL MEMORY

Van Kreveld's algorithm uses four structures: the elevation grid, the visibility grid, the event list and the active structure. Even if the event list is stored as a stream on disk, and sorted I/O-efficiently, the algorithm would still use $\Omega(n)$ I/Os to maintain and query the active structure.

In Section 3.1, we describe simple modifications to the plane sweep algorithm to obtain I/O-efficiency, under the assumption that the active structure fits in memory. Although the resulting algorithm is not guaranteed to be worst-case efficient, it widely extends the size of the problems that can be tackled in practice (as showed by the experimental evaluation). In Section 3.2, we extend the approach to an algorithm that needs only $O(sort(n))$ I/Os in the worst case.

3.1 The Base Case

The inefficiency of van Kreveld's algorithm is caused by three problems. First, the elevation grid is loaded in memory in row-column order, but the grid is accessed (read) in rotating sweep order to determine each cell's height above the horizon as it is entered or queried for in the active structure. Second, the visibility grid is loaded in memory in row-column order and is accessed (written) in sweep order. Third, there is little structure in the way in which the active structure is accessed.

The first problem can be solved, for example, by augmenting the events in the event list with information about the elevation of the cell so that the input grid does not need to be accessed at all once the event list has been built. The second problem can be solved by recording the visibility of each cell in a list in sweep order, and sorting the list into grid order after completing the sweep. With these two modifications, the plane sweep does not need to load the input and output grid into memory. Assuming the event list is stored in a stream, the algorithm can now use all available memory for the active structure. If the active structure is small enough to fit in memory, the algorithm runs in $O(sort(n))$ I/Os.

If the active structure does not fit in memory, an immediate idea would be to implement it with a B-tree, but this would give a running time of $O(n \log_B n)$ I/Os. Below we explain how to do better.

3.2 An $O(sort(n))$ I/O Algorithm.

In this section, we describe how to compute a visibility grid in worst-case $O(sort(n))$ I/Os without any assumptions on the input. The algorithm is based on the distribution sweeping technique, which we describe below.

3.2.1 *Distribution Sweeping.* The general idea in distribution sweeping is to divide the input into $O(M/B)$ and $\Omega((M/B)^\epsilon)$ strips (slabs), each containing an equal number of input objects. Using these strips, we decompose the solution to the problem into a part that can be found recursively in each strip and a part that involves interactions between strips. The recursive part of the solution we can find by solving the problem recursively in each strip. The recursion stops when the strips are small enough to fit in main memory, which happens after $O(\log_{M/B} n)$ recursion steps. The challenging part in distribution sweeping is finding the part of the solution that involves interactions between strips. To get an algorithm that uses only $O(sort(n))$ I/Os, we need to handle interactions in $O(n/B)$ I/Os in total per level of recursion. Two approaches for this problem have been described in the literature: handling the interactions by a plane sweep based on maintaining active lists for each strip (this approach was taken by Goodrich et al. [1993] to compute intersections of orthogonal line segments), and using $O(\sqrt{M/B})$ fan-out and multislabs, used by Arge et al. [1995] for red-blue line segment intersections. We show how to handle strip interactions for the visibility problem by combining a radial (rotational) and a concentric sweep.

Our algorithm follows the general approach of distribution sweeping described above: we divide the grid into $\Theta(M/B)$ radial "strips" (sectors) around

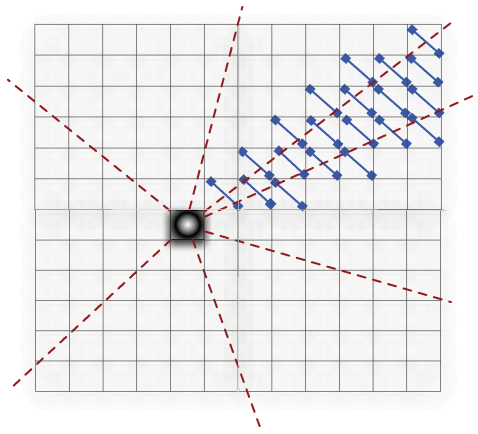


Fig. 3. Distribution sweeping. The grid is divided into M/B radial slabs (sectors); cells may cross zero, one or more sector boundaries.

the viewpoint. A cell can be *narrow*, that is, crossing at most one sector boundary, or *wide*, that is, crossing at least two sector boundaries and thus spanning at least one sector completely—refer to Figure 3. On every level of recursion, we first determine how *wide* cells affect the visibility of cells in the sectors spanned by them, and then compute the visibility grid recursively in each sector. The recursion stops when the number of cells gets small enough to run the base case algorithm described in Section 3.1 while keeping the active structure in memory. Below we detail the steps and their analysis.

We start by scanning the grid to identify the 3 events associated with each cell; for each cell Q, we determine the point $p_1 = (r_1, \theta_1)$ where Q will be hit by the rotating sweep line first, the point $p_2 = (r_2, \theta_2)$ that will be the last point of Q to intersect the sweep line, and the center point $q = (r_q, \theta_q)$ of Q. The points are represented in polar coordinates with respect to the viewpoint. As we scan the grid, we create two lists of events, E_r and E_c. The list E_r consists of all the event points, in the entire grid. The list E_c contains two copies of each cell Q: One copy is marked as *query* and stores the center point q of Q; the other copy is marked as *obstacle* and stores the points p_1 and p_2 where the rotating sweep line will first and last intersect it. With each list element, we also store the height above the horizon of the center point.

After we finish scanning all the cells, we sort E_r by radial order around the viewpoint, and the elements in E_c by distance from the viewpoint to their center points. We sort E_r and E_c only once at the beginning—we construct the sorted event lists for recursive calls by distributing the events of E_r and E_c among the recursive calls while keeping them in order.

Given the two sorted lists E_r and E_c, the recursion step of the algorithm proceeds in two phases: a *radial sweep*, which processes events from list E_r in order of the polar angle with respect to the viewpoint, and a *concentric sweep*, which processes the events from list E_c in increasing order of their distance from the viewpoint. The radial sweep is used to partition the events into approximately

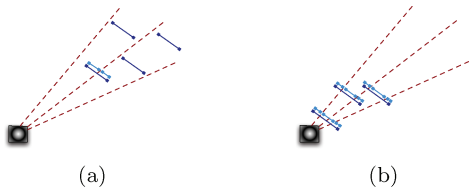


Fig. 4. Segments corresponding to (a) narrow cells. (b) wide cells.

M/B equal-sized sectors. The concentric sweep is used to process wide obstacles and to distribute the query events and narrow obstacles. By processing events concentrically, we are able to maintain, while distributing query events, if there is any wide obstacle closer to the viewpoint that may occlude the center of the query cell.

The details are slightly different, depending on whether we want to compute a visibility grid (distances to visibility) or only a viewshed (whether cells are visible or not). We first discuss the details for the viewshed computation.

3.2.2 *The Radial Sweep.* First, we need to determine approximately M/B sectors such that each sector contains $O(n/(M/B))$ event points. This is easily done by scanning E_r to identify the sector boundaries. While doing so, we also compute, for each sector, a list of the events in that sector in radial order. We will use these lists later as we recurse on the sectors to find radial partitions within each sector.

3.2.3 *The Concentric Sweep.* During this sweep, we scan and process the events (queries and obstacles) in E_c in order of increasing distance from the viewpoint, and for each sector we construct a list of events in that sector in the same order. To do this I/O-efficiently, we keep, for each sector, a buffer of one block of data in memory. Initially, the buffers are all empty. They will get filled with events during the sweep—whenever a buffer runs full, its contents are output to disk and the buffer is emptied.

Furthermore, we keep for each sector S the radii that form its boundaries in memory, and a variable $High_S$ that holds the greatest height above the horizon among all the wide obstacles that completely span that sector and that have been reached by the concentric sweep. Initially $High_S$ is set to $-\infty$ for each sector.

The concentric sweep now proceeds as follows. We go through the events in E_c in order of increasing distance from the viewpoint. Events may be queries or obstacles.

If the event is a query cell Q, we determine which sector S contains its center point, and check if the height of Q above the horizon is at least $High_S$. If so, we write Q to the event buffer of S; if not, we output Q to a list of points that have been found to be invisible, and do not copy it to any event buffer.

If the event is an obstacle E, it may intersect several sectors (Figure 4).

For each sector S that is intersected by E, but not completely spanned by it, we check if the height of E above the horizon is more than $High_S$. If so, we write E to the event buffer of S. If not, we ignore E for this sector, because it

cannot occlude any query cells in S that are not already occluded by the wide obstacle that determined $High_S$.

For each sector S that is spanned by E (i.e, E touches or intersects both radii that delimit S), we update $High_S$ by setting it to the maximum of S and the height above the horizon of (the center point of) E (this takes no I/O, as all the necessary information is kept in memory). As a result, all query cells in S that are occluded by E will be filtered out of the event stream of S.

3.2.4 *In Recursion.* After completing the concentric sweep, each sector is processed recursively. Throughout the recursion cells are marked as invisible and discarded. All query cells that are visible survive until the final level of recursion. The recursion stops when we can run the algorithm from Section 3.1 while keeping the active structure in memory.

Note that as a result of the early discards, the lists E_c and E_r for a sector may get out of sync. That is, E_r may contain events whose corresponding cells have already been discarded in the course of a concentric sweep. This is not a problem, it just means that the size of each recursive problem may actually be less than $O(n/(M/B))$.

3.2.5 *Computing a Visibility Grid.* The algorithm above can be extended to compute the vertical distance to visibility for each invisible cell: As we process a query cell Q, we keep track of $High_Q$, the greatest height above the horizon among the wide obstacles that occluded (the center point of) Q through the recursive steps. When a query cell Q is occluded, we do not discard it, but we update its $High_Q$ variable, insert it in the event list of its sector, and let the recursive calls determine its distance to visibility.

3.2.6 *Analysis.* Initial sorting of the input into two event streams E_r and E_c, and sorting the output into grid order, takes $O(sort(n))$ I/Os. At each level of recursion, a query is inserted in the event stream of at most one sector. An obstacle is inserted in the event stream of at most two sectors; Overall, an obstacle is split into two parts (that do not completely span sectors) at most once, not on every level of the recursion. Hence, both the radial and the concentric sweeps use $O(n/B)$ I/Os per level of recursion, and there are $O(\log_{M/B} n/M)$ levels of recursion. Thus, the entire algorithm uses $O(sort(n))$ I/Os and linear space. We have the following:

THEOREM 3.1. *The visibility grid of an arbitrary viewpoint on a grid of size n can be computed with $O(n)$ space and $O(sort(n))$ I/Os.*

Note that the radial sweep and the sorted list E_r are only necessary for simplicity. It is possible to partition the events in E_c radially by using an $O(n/B)$ pivot-finding algorithm, as in Aggarwal and Vitter [1988]. Since the algorithm is complicated, maintaining the events in radial order throughout the recursion is probably more likely to be efficient in practice. If the boundary of the grid has an easy shape, for example, if the grid is rectangular and all cells are valid, one may simply compute the sector boundaries analytically in memory, instead of determining them by a radial sweep or pivot-founding algorithm.

4. A CACHE-OBLIVIOUS ALGORITHM

In this section, we describe how the algorithm for computing the viewshed of a viewpoint on a grid terrain can be extended to the cache-oblivious model. The result is based on the cache-oblivious distribution sweeping technique described by Brodal and Fagerberg [2002]. The basic idea is to view the distribution sweep as a bottom-up binary merging process, instead of top-down M/B-way distribution, which cannot be done when M and B are not known, and implement it using the framework of Funnelsort [Frigo et al. 1999].

The merging process proceeds as follows. Sort all 3n events—queries and endpoints of obstacles—radially around the viewpoint. Initially each event constitutes its own sector. We assume inductively that each sector has a list of the events in that sector, ordered by increasing distance from the viewpoint. We merge consecutive pairs of adjacent sectors A and B, by scanning their two lists of events E_A and E_B; The merge produces the list of events $E_{A\cup B}$ for the sector $A \cup B$, ordered by distance from the viewpoint. While merging, we will maintain the invariant that if the endpoint of an obstacle is output in $E_{A\cup B}$, then all query events in the sector that are occluded by the obstacle have been marked so and eliminated from $E_{A\cup B}$. Thus, all query events in the final output list of the merger will represent the points that are visible from the viewpoint.

To merge sectors A and B, we scan events in order from E_A and E_B. During the merge, we maintain two variables $High_A$ and $High_B$, which represent the highest obstacle below the sweep line (the merge can be viewed as a sweep in distance order) that has one endpoint in B (respectively A) and spans A completely (respectively B). Let p be the next event in order from $E_A \cup E_B$. If p is a query event in A (B), we check if it is occluded by $High_A$ ($High_B$), and if it is not, we output it in $E_{A\cup B}$; otherwise, if it is occluded, we drop it. If p is the endpoint of an obstacle that is completely contained inside $A \cup B$, then we drop it—by the invariant, all points that are occluded by this obstacle have been marked so while producing E_A and E_B. If p is the end point of an obstacle that starts in A (B) and completely spans B (A), then we update $High_B$ ($High_A$) and output p in $E_{A\cup B}$.

The overall structure of the merge is very similar to Mergesort and can be implemented similar to Funnelsort [Frigo et al. 1999], or using a simplified variant proposed by Brodal and Fagerberg [2002]. We have the following:

THEOREM 4.1. *The visibility grid of an arbitrary viewpoint on a grid of size n can be computed cache-obliviously in $O(sort(n))$ I/Os.*

5. EXPERIMENTAL RESULTS

This section presents an experimental evaluation of computing viewsheds on large real-life grid terrains. We compare our I/O-efficient viewshed algorithm, `ioviewshed`, described in Section 3, with van Kreveld's internal memory algorithm (`kreveld`) described in Section 2 and with a module (`r.los`) that implements this functionality in the widely used open source GIS GRASS.

`r.los`: The open source GIS GRASS provides viewshed computation via a module called `r.los`. The module uses the straightforward $O(n^2)$ ($O(n\sqrt{n})$ on square

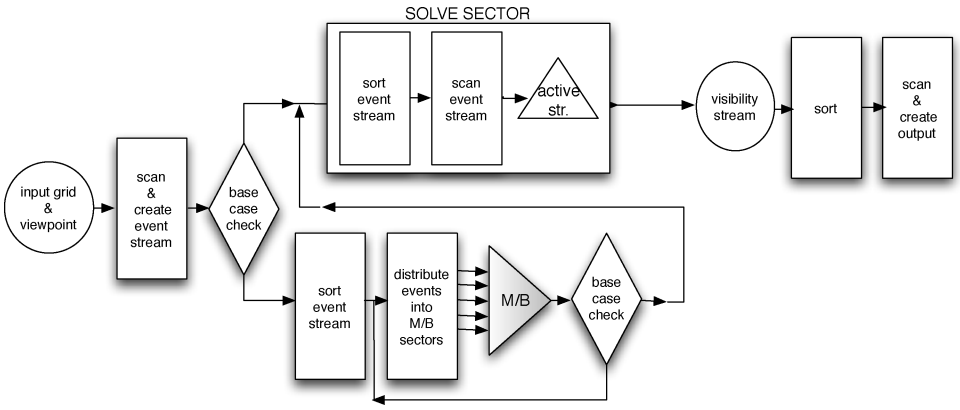


Fig. 5. Flow chart of `ioviewshed`.

grids) algorithm and GRASS' own memory management tool (which essentially mimics a virtual memory manager by moving data between memory and disk). From the experiments, we will see that `r.los` is extremely slow. We include this module in our experiments for perspective, as GRASS is the most used open-source GIS, because it illustrates the thrashing of a super-linear algorithm, and because we used `r.los` to test `ioviewshed` for correctness on boundary cases and undefined data.[1]

`kreveld`: For a description of the algorithm, see Section 2. The program maintains in memory the elevation grid (input), visibility grid (output), the event array and the active structure. The first step scans the input grid and creates the event array. The events are then sorted using an optimized version of in-memory quicksort (the same one used to sort in-memory runs in the I/O-efficient library used in `ioviewshed`, see `IOStreams` below). The main phase in `kreveld` is the sweeping, which reads events in order from the array and updates the active structure and the visibility grid. For the active structure, we used a standard implementation of red-black tree [Cormen et al. 2001].

`ioviewshed`: The implementation of `ioviewshed` follows the I/O-efficient algorithm outlined in Section 3. Its flow-chart is shown in Figure 5. First, it scans the elevation grid to create the event stream and estimates the size of the active structure to check if it can fit completely in memory. If it does, it proceeds with the base-case: it sorts the event stream radially, and then scans the event stream while keeping the active structure in memory. The active structure used is a red-black tree [Cormen et al. 2001], the same as in `kreveld`. During the radial sweep, the visibility of each cell is written to the visibility stream.

If the active structure does not fit in memory, `ioviewshed` proceeds with the recursive distribution: First it sorts the event stream by distance from the viewpoint and computes the boundaries of the $\Theta(M/B)$ sectors; it then scans

[1] `ioviewshed` is being ported to GRASS and will be made available to GRASS users as an alternative to `r.los`.

Table I. `ioviewshed` Statistics: Maximum Size of the Active Structure and Size of the Event Stream. The Size of the Active Structure is Estimated as Grid-Max-Size * 40B per point

Data Set	Size	Active Structure	Event Stream
Kaweah	7MB	.06MB	45MB
Sierra Nevada	40MB	.15MB	440MB
Cumberlands	267MB	.35MB	817MB
Lower NE	311MB	.36MB	1,289MB
East-Coast USA	983MB	.73MB	3,669MB
Horn of Africa	1,370MB	.84MB	8,057MB
Midwest USA	1,122MB	1.02MB	11,048MB
Washington	4,264MB	1.33MB	48,791MB

the events in concentric order while keeping track of the highest spanning cell so far for each sector, and produces a stream of (un-occluded) events in each sector; then it repeats the destribution step for each sector—until a sector can be solved in memory—at which point it switches to the base-case as above.

We did a couple of simplifications in our implementation of the distribution: First, given the regular structure of the grid, and the large size of the event stream (Table I), we do not keep the event stream sorted both radially and by distance (E_r and E_c in Section 3.2). Instead, we compute sector boundaries symbolically as multiples of $2\pi/(M/B)$. Thus, throughout the recursion, we have only one event stream, which is initially sorted in concentric order, and maintained sorted through recursion. Second, it is hard to estimate the size of the active structure in a sector (the diagonal remains the same, events get fewer as they are dropped because of long-obstacle occlusions). Instead, we stop recursion when the size of one sector fits in memory.

During distribution, we need to find the sectors that contain the end points of a cell. This involves calculating and comparing angles and leads to precision issues as sectors get small. We handle this by creating a separate boundary stream for each sector that contains all ENTER events outside the sector whose corresponding EXIT events fall inside the sector. Throughout distribution, we keep the invariant that all long cells that crosss the boundary of a sector are stored in its boundary streams. Thus, each cell is evaluated precisely once and assigned to a sector stream.

The output of the sweep phase, base-case or recursive, is a stream that records visibility. During the sweep, `ioviewshed` labels each cell (i, j) as visible or not and writes it to the visibility stream; at the end, the visibility stream is sorted in (i, j) order to produce the output grid. We observe that, most of the time, only a small fraction of the terrain is visible for large terrains; when we compute the viewshed we record only the cells that are visible, and assume that cells not recorded are invisible to the observer. This optimization significantly decreases the time needed to sort the output.

For the I/O-efficient components, `ioviewshed` uses `IOStreams` [Toma 2003], an I/O-efficient library derived from `TPIE` [Arge et al. 2005]. `IOStreams` provides basic stream functionality along with an I/O-optimal external mergesort [Aggarwal and Vitter 1988] and an I/O-efficient priority queue [Arge et al. 2001]. The only components that are used in `ioviewshed` are scanning and sorting streams.

Table II. Size of Terrain Data Sets

Data Set	Points	Size	Valid	Valid Points
Kaweah	$1.6 \cdot 10^6$	7MB	56%	$0.9 \cdot 10^6$
Sierra Nevada	$9.5 \cdot 10^6$	40MB	96%	$9.1 \cdot 10^6$
Cumberlands	$67 \cdot 10^6$	267MB	27%	$18 \cdot 10^6$
Lower New England	$78 \cdot 10^6$	311MB	36%	$28 \cdot 10^6$
East-Coast USA	$246 \cdot 10^6$	983MB	36%	$88 \cdot 10^6$
Horn of Africa	$359 \cdot 10^6$	1 370MB	51%	$183 \cdot 10^6$
Midwest USA	$280 \cdot 10^6$	1 122MB	86%	$240 \cdot 10^6$
Washington State	$1\,066 \cdot 10^6$	4 264MB	95%	$1013 \cdot 10^6$

The valid-count excludes undefined points, which are ignored when computing visibility.

(a) Sierra Nevada (b) East-Coast USA (c) Horn of Africa

Fig. 6. Test datasets used for the experiments. The visualization was done with GRASS, with undefined points shown in white.

Configuration: `ioviewshed` and `kreveld` are implemented in C++ and compiled using the g++ 4.0.1 compiler with optimization level −O3. All experiments were run on Apple Power Macintosh G5 computers with dual 2.5 GHz processors (we use only one), 512KB L2 cache per processor, 1GB RAM, and a Maxtor serial-ATA 7,200RPM hard drive. We allowed `ioviewshed` algorithm to use 500MB of the available memory. We also ran experiments by rebooting the machines with 256MB RAM and allowing `ioviewshed` to use 64MB.

Datasets: Table II, describes the data sets, representing real terrains of various characteristics ranging from 1 to over 1 000 million elements. For a visual comparison, three datasets are depicted in Figure 6. Real terrains contain points for which the elevation is unknown or invalid (e.g., oceans, lakes); these points are marked as undefined points. Typically undefined points are ignored when processing the terrain, so the amount of undefined data can significantly influence the running time. The amount of valid data for each terrain is also given in Table II.

Both `kreveld` and `ioviewshed` are optimized to handle undefined points. These points are ignored during the computation and their corresponding events are not generated. For `kreveld`, in particular, which stores the events in an array, this significantly decreases the size of the memory requirement.

Table III. Running Times (seconds) and CPU-utilization (in parentheses) for `r.los`, `kreveld` and `ioviewshed` Running with 1GB RAM

Data Set	`r.los`	`kreveld`	`ioviewshed`	
			Event Sort	Total
Kaweah	2,928	5 (97%)	3 (66%)	6 (82%)
Sierra Nevada	16,493	49 (91%)	35 (68%)	74 (81%)
Cumberlands	>200,000	368 (54%)	115 (42%)	200 (65%)
Lower NE		987 (38%)	193 (40%)	289 (62%)
East-Coast USA		4,114 (35%)	500 (46%)	959 (66%)
Horn of Africa		12,359 (31%)	1,052 (47%)	1,960 (70%)
Midwest USA		17,185 (30%)	1,487 (47%)	2,698 (68%)
Washington		117,525 (27%)	7,393 (43%)	14,012 (70%)

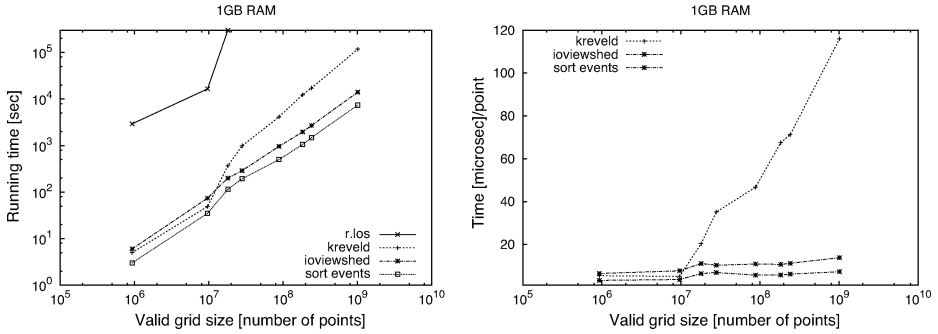


Fig. 7. Running time of `r.los`, `kreveld` and `ioviewshed` at 1GB RAM. (a) Total time (log scale) with valid size (log scale). (b) Total time per valid point.

All viewshed timings were obtained by selecting 10 viewpoints uniformly on each terrain, computing the viewshed for each one, and taking the average time.

5.1 Results with 1GB Main Memory

Figure 7 shows the total running times for `r.los`, `kreveld` and `ioviewshed` with 1GB RAM on the test datasets. For a comparison to sorting, the time for sorting the event stream in `ioviewshed` is depicted separately. The detailed running times, together with the corresponding CPU utilization, are summarized in Table III.

`r.los` is by far the slowest of the three algorithms, for both small and large inputs. It computes a viewshed in 50 minutes on Kaweah, more than 4.5 hours on Sierra Nevada, and did not finish in two days on Cumberlands ($n = 67 \cdot 10^6$ points). The inefficiency of `r.los` is due to the straightforward $O(n^2)$ algorithm ($O(n\sqrt{n})$ on square grids) and the overhead of the `GRASS` segment library.

`kreveld` performs very well as long as data fits in main memory. For a grid of n points, the size of the memory necessary during the algorithm is about $64n$ bytes (elevation grid, visibility grid, and event array); Thus the largest data set that can be processed completely in memory with 1GB RAM has at most 17 million points. If the data structures fit completely in memory, the

algorithm finishes in seconds and its CPU utilization is high, which is the case on Kaweah (1.6 million points) and Sierra Nevada (9.5 million points). On Cumberlands (67 million points) the CPU utilization drops to 54%, the I/O wait time increases, and `kreveld` starts thrashing. The tendency becomes clearer on the larger datasets, where we see the CPU utilization dropping to 25%, 31%, 30% and reaching 27% on Washington State, which is processed in approximately 32 hours, of which more than 23 hours is I/O-wait time.

In contrast, the performance of `ioviewshed` scales nicely and the running time per point stays nearly constant with increasing input size, as can be seen from Figure 7(b). `ioviewshed` processes terrains of approximately 10^9 points in about 3.8 hours—this involves sorting an event stream of 48GB (Table I), which constitutes approximately 50% of the running time.

The most important finding of our experiments concerns the depth of the recursion in distribution sweeping on grid terrains. Table I gives the maximum size of the active structure and the size of the event stream, for each input. We found that with data sets up to 4GB and main memory as low as 256MB (see below), the active structure fits completely in main memory, while the event stream gets very large. The standard base case for an I/O-efficient algorithm is when subproblem size fits in memory. By designing the base case to take advantage of the small active structure, instead of checking problem size, `ioviewshed` avoids recursion on all inputs.

We conjecture this to be true in general: Distribution sweeping on realistic terrains, with realistic values of n and M, can be optimized to avoid recursion in practice. This should be used as a starting point for designing practical applications of distribution sweeping on realistic data: First, for realistic sizes of n and M at present, we have[2] $O(\sqrt{n}) < M$. Second, realistic terrains are not long and skinny and one of their sides, or diagonal, fits in memory.[3]

For terrains that are not realistic in the above sense, our algorithm guarantees $O(\log_{M/B} n)$ levels of recursion in the worst case, and an overall $O(sort(n))$ I/O upper bound; for experimental results with recursion, see Section 5.3.

5.2 Results with 256MB Main Memory

We also ran experiments with 256MB of RAM. While a main memory size of 256MB is certainly not realistic, it illustrates the behaviour of the algorithms as the difference between dataset and memory size increases. The total running times for `r.los`, `kreveld` and `ioviewshed` with 256MB are shown in Figure 8 and Table IV. The results are very similar to those at 1GB RAM—with the crossover point between the internal and external algorithms moving to the left in this case: At 1GB, `kreveld` can fit in memory the smallest two datasets and starts thrashing on the third one; at 256MB, it starts thrashing on the second one.

[2]For e.g., with 64B per point, $M = 1$GB can fit 2^{24} points, which means up to $n = 2^{48}$ points.

[3]A terrain that stretches from the South Pole to the North Pole of the earth at .5m resolution (such a resolution is not currently possible) will have $6400 \cdot 10^3 \cdot 2 \cdot 2 = 25 \cdot 10^6$ points along this dimension—the active structure for the viewshed computation, at 40B per point, totals 1,000MB and fits in memory.

Table IV. Running Times (Seconds) and CPU-Utilization (in Parentheses) for `r.los`, `kreveld` and `ioviewshed` Running with 256MB RAM

Data Set	`r.los`	`kreveld`	`ioviewshed`	
			Event Sort	Total
Kaweah	3,177	5 (92%)	3 (87%)	6 (91%)
Sierra Nevada	19,140	346 (30%)	65 (40%)	93 (58%)
Cumberlands	>200,000	911 (33%)	139 (34%)	198 (68%)
Lower NE		1,701 (30%)	202 (38%)	295 (67%)
East-Coast USA		5,229 (32%)	673 (36%)	995 (62%)
Horn of Africa		16,157 (26%)	1,943 (34%)	2,719 (54%)
Midwest USA		23,291 (24%)	2,745 (32%)	3,906 (51%)
Washington			14,226 (32%)	20,015 (51%)

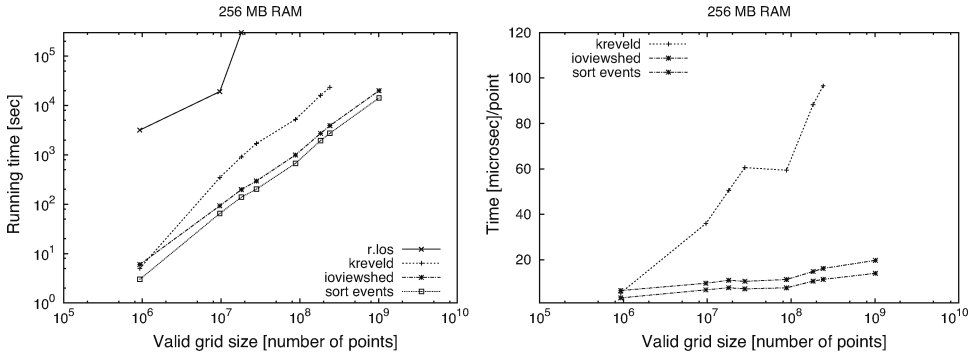


Fig. 8. Running time of `r.los`, `kreveld` and `ioviewshed` at 256MB RAM. (a) Total time (log scale) with valid size (log scale). (b) Total time per valid point.

Figures 9 and 10 show the performance of `kreveld` and `ioviewshed` comparatively at 256MB and 1GB RAM. With 256 MB `kreveld` processes Kaweah in a few seconds, as with 1GB. However, it starts thrashing on Sierra Nevada: 346 seconds with 30% CPU utilization as opposed to 49 seconds, 91% CPU utilization at 1GB. This is not surprising, as the amount of memory needed to process Sierra Nevada is $11 \cdot 9.5 \cdot 10^6 \cdot 4\text{B} = 418\text{MB}$, which does not fit in a main memory of 256MB. The thrashing gets worse on the larger datasets—see Figure 8(b)—with the CPU utilization going down to 24% on Midwest US; the 4GB-dataset (Washington State) cannot be handled by `kreveld` at 256MB, as it consistently crashes the machine.

The performance of `ioviewshed` is practically the same at 256MB and 1GB for the smaller datasets. On the largest dataset, Washington State, the running time increases from 14,012 seconds (or 14 μsec per point) with 1GB to 20,015 seconds (or 20 μsec per point) with 256MB. The cause of this increase is essentially sorting the event stream, which takes more time with less memory (7,393 seconds or 53% of the total running time with 1,GB and 14,226 or 71% of the total running time with 256,MB). The I/O-wait time, currently at 50%, is also due to sorting the event stream and can be improved by fine-tuning the I/O-library to the platform.

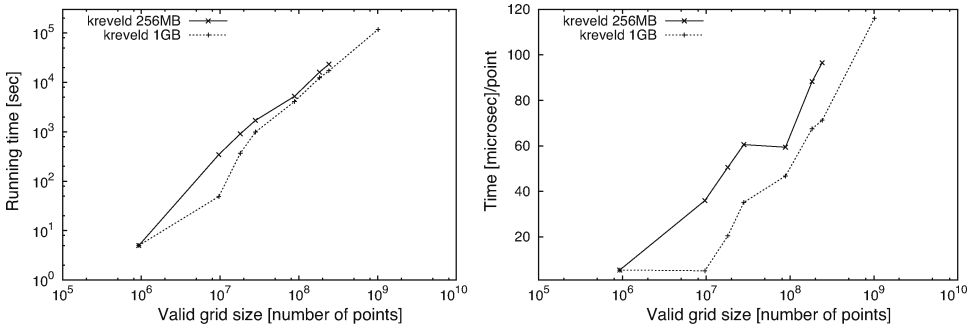


Fig. 9. Running time of `kreveld` at 256MB and 1GB RAM. (a) Total time (log scale) with valid size (log scale). (b) Total time per valid point.

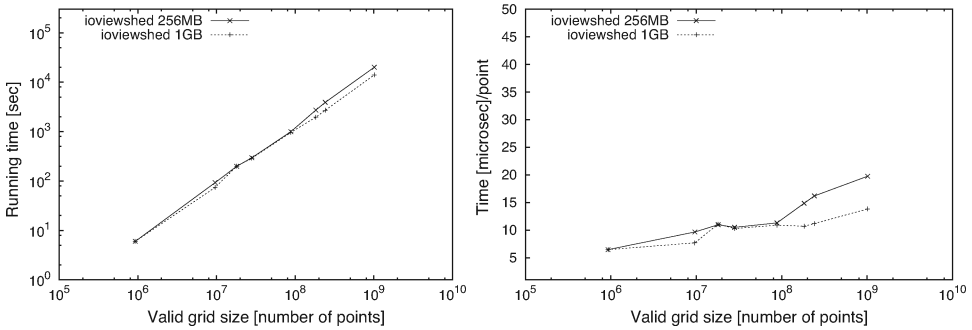


Fig. 10. Running time of `ioviewshed` at 256MB and 1GB RAM. (a) Total time (log scale) with valid size (log scale). (b) Total time per valid point.

5.3 Testing `ioviewshed` with Recursive Sweeping

Since none of our real datasets can trigger recursion, we tested the recursive phase of `ioviewshed` by skipping the base-case check and forcing it to enter recursion. We call this version of the algorithm `R-ioviewshed`. The results are given in Figure 11. The total time is broken into the time to sort the event stream by distance from the viewpoint (`R-sort`), and the recursive sweep time (`R-sweep`)—which is the time to distribute the events into sectors recursively until each sector fits in memory, and solve each sector in memory; so `R-sweep` does not include the initial concentric sort time, but it includes the time to sort the events radially once sectors become small enough. The time for input and output (initialization, sorting the visible cells and assembling the output grid), which accounts for less than 1% of the total time, is not shown. The total time is dominated by `R-sort`.

Figure 11(c, d) shows comparatively `R-ioviewshed` and `ioviewshed`. Similarly with `R-ioviewshed`, `ioviewshed` time is broken into the time to sort the event stream (radially), and `sweep`—the time to scan the sorted event stream using the active structure in memory. The slowdown for `R-ioviewshed` is caused both by recursion and by sorting; sorting the event stream by distance is

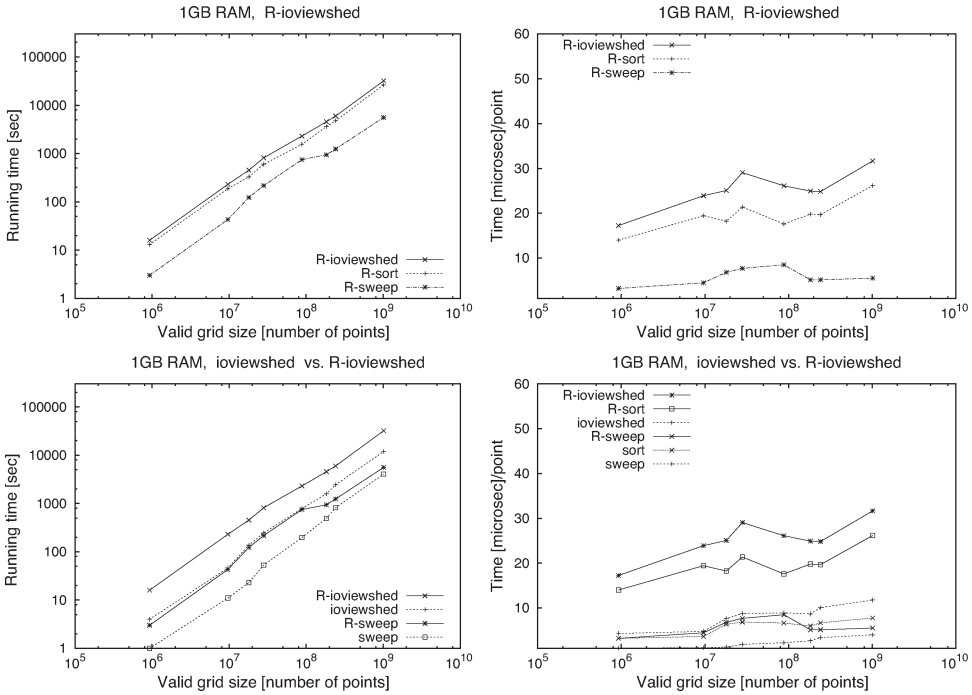


Fig. 11. (a, b) Running time of `R-ioviewshed` at 1GB RAM: Total time, event stream sort time, and recursive sweep time with valid size. (c, d) Comparison between `ioviewshed` (base-case with radial sweep) and `R-ioviewshed` (recursive concentric sweep).

more CPU intensive and significantly slower than sorting by angle. At present, `ioviewshed` is optimized to make the common-case fast and stores with each event its angle, but not its distance wrt viewpoint; the distance is thus computed on-the-fly; therefore, the increase in sorting time. This can be easily improved.

Figure 12 shows in detail the difference between `R-sweep` and `sweep`. Both use less than 10 microseconds per point on all inputs and are scalable, but `R-sweep` shows a lot of variation. The times in Figure 12(a) reflect average times per viewpoint; Figure 12(b) shows the minimum and maximum sweep times for all viewpoints. The nonrecursive sweep is relatively stable, while `R-sweep` depends strongly on the input; the depth of recursion depends on the number of events that get pruned when they become invisible due to closer long cells that span the sector. In all experiments, the recursion depth is 1 or 2 for all datasets. For example, on Midwest US, `R-sweep` takes between 498 seconds, and 2,033 seconds; in the first level of recursion anywhere between 191 million and 670 million events are dropped out of total 724 million events.

Across all datasets, we found that the number of events pruned is consistently high. Though in the worst case, the events will survive for $\log_{M/B} n$ levels, the common case seems to be much better. Pruning of the events in concentric sweeping is a feature that can be exploited for a RAM optimized version of

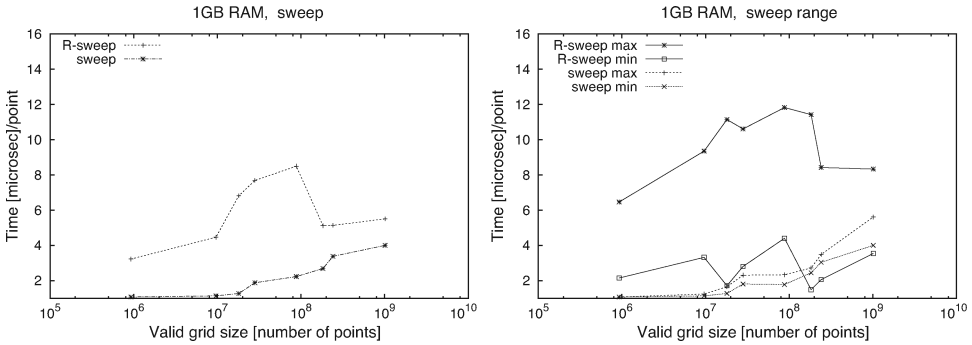


Fig. 12. Comparison of radial sweep and recursive concentric sweep. (a) Total (average) time per valid point. (b) Min-max sweep range.

viewshed; combined with the cache-oblivious approach in Section 4 it has potential to win over `kreveld`.

5.4 Testing a Different I/O-Library

As Table III and IV show, at least 50% of the running time of `ioviewshed` is spent in sorting. In all experiments above, `ioviewshed` uses `IOStreams` I/O-library [Toma 2003]. The only I/O-components used in the implementation are scanning and sorting.

It is possible that the performance can be further improved with a careful tuning of the parameters, or substituting another library, like `TPIE` [Arge et al. 2005] or `STXXL` [Dementiev et al. 2005]. Both these libraries allow for fine tuning of parameters and exploit special properties of the underlying hardware. However, it will not change the order of magnitude of the running time, nor the relative times of the phases in `ioviewshed`; thus it will not affect the conclusion of the experiments.

To this end, we have performed experiments with `ioviewshed` plugging in `TPIE`, which was configured with the standard default settings [Danner 2007]. Figure 13 shows the results. We see that the performance stays approximately the same, and running times do not differ by more than 10% on the largest datasets.

6. CONCLUSION AND DISCUSSION

In this article, we show that the visibility map of a point on a grid terrain can be computed I/O-efficiently both in theory and in practice. We give algorithms for this problem that run in $O(sort(n))$ I/Os in the cache-aware and cache-oblivious model. Like many of the algorithms for GIS problems, our algorithms use the technique of distribution sweeping. We show that in practice the visibility on a terrain of about 1.1 billion points can be computed in less than 4 hours on a low-cost machine, compared to 32 hours using the internal memory algorithm. As far as we know, this is the first experimental evaluation of distribution sweeping in external memory.

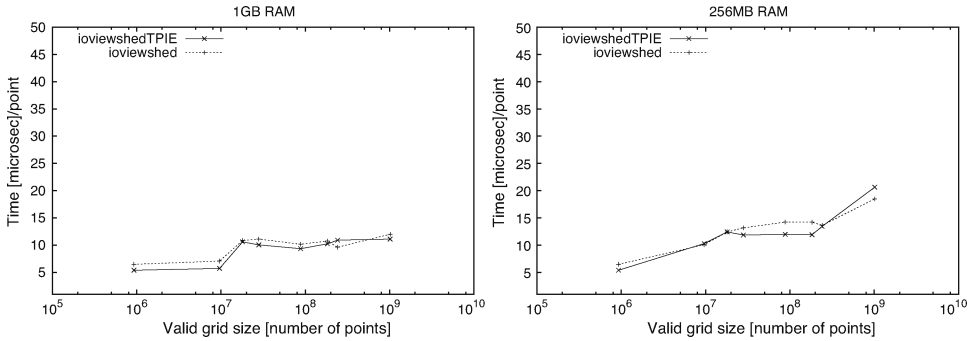


Fig. 13. Running time per valid terrain point of `ioviewshed` with two I/O-libraries: `IOStreams` and `TPIE`. (a) 1GB RAM. (b) 256MB RAM.

An important empirical finding is that for realistic terrains and realistic memory sizes, the diagonal of a terrain fits in memory, which implies that the active structure fits in memory during sweeping. By designing the base case of the recursive step to use both disk and memory, one can widely extend the size of the terrain that can be processed without the actual recursive distribution step. We believe this to be true for other applications of distribution sweeping, on other types of datasets, which are not arbitrarily long and skinny. Should recursion become necessary, we prove that the fully recursive viewshed computation is also efficient and scalable.

A question for further investigation is the practical efficiency of the cache-oblivious version of our algorithm. Evidence of the practical advantages of cache-oblivious algorithms in external memory is not yet convincing. The recent work of Ajwani et al. [2007] discusses the penalty for cache obliviousness in external memory BFS. As reported by Brodal et al. [2007], their implementation of cache-oblivious sorting (Lazy Funnelsort) is at least as efficient as the RAM- and cache-optimized sorting algorithm, however, clearly less efficient than `TPIE` sorting. Since `ioviewshed` spends most of the time in sorting, this suggests that a cache-oblivious `ioviewshed` may not be a good idea for external-memory. It is possible that sweeping can be improved by using the binary merge as described in Section 4, which would optimize it for all levels of the hierarchy. However, since sweeping represents less than 25% of the total running time, the overall improvement, if any, would not be significant.

However, cache-obliviousness may be interesting in a RAM-optimized version of the algorithm, especially in view of the fact that radial sweep does not take advantage of the input, while distribution appears output-sensitive in practice and is helped by heavy event pruning. A related question is finding a provably output sensitive algorithm, running for example in $O(scan(n) + sort(k))$, where k is the size of the viewshed.

ACKNOWLEDGMENTS

We thank Andrew Danner for his help with `TPIE`.

REFERENCES

AGARWAL, P. K., BEREG, S., DAESCU, O., KAPLAN, H., NTAFOS, S., AND ZHU, B. 2005. Guarding a terrain by two watchtowers. In *Proceedings of the ACM Symposium on Computational Geometry*. 346–355.

AGGARWAL, A. AND VITTER, J. S. 1988. The Input/Output complexity of sorting and related problems. *Communications of the ACM 31*, 9, 1116–1127.

AJWANI, D., MEYER, U., AND OSIPOV, V. 2007. Improved external memory bfs implementations. In *alenex*.

ARGE, L. 1997. External-memory algorithms with applications in geographic information systems. In *Algorithmic Foundations of GIS*, M. van Kreveld, J. Nievergelt, T. Roos, and P. Widmayer, Eds. Springer-Verlag, LNCS 1340, 213–254.

ARGE, L., BARVE, R. D., HUTCHINSON, D., PROCOPIUC, O., TOMA, L., VAHRENHOLD, J., VENGROFF, D. E., AND WICKREMESINGHE, R. 2005. TPIE user manual and reference, edition 1.0. Duke University, NC, "http://www.cs.duke.edu/TPIE/". (In Preparation).

ARGE, L., BRODAL, G. S., AND FAGERBERG, R. 2005. *Cache-oblivious Algorithms*. Vol. Handbook of Data Structures and Applications. CRC Press, Chapter 34.

ARGE, L., TOMA, L., AND VITTER, J. S. 2001. I/O-efficient algorithms for problems on grid-based terrains. *ACM J. Experimental Algorithmics 6*. Article 1.

ARGE, L., VENGROFF, D. E., AND VITTER, J. S. 1995. External-memory algorithms for processing line segments in geographic information systems. In *Proceedings of the European Symposium on Algorithms*. LNCS 979. 295–310.

BEN-MOSHE, B., KATZ, M. J., AND MITCHELL, J. S. B. 2005. A constant-factor approximation algorithm for optimal terrain guarding. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*. 515–524.

BRODAL, G. S. 2004. Cache-oblivious algorithms and data structures. In *Proceedings of the Scandinavian Workshop on Algorithms Theory*. LNCS 3111. 3–13.

BRODAL, G. S. AND FAGERBERG, R. 2002. Cache oblivious distribution sweeping. In *Proceedings of the International Colloquium on Automata, Languages, and Programming*. LNCS 2389. 426–438.

BRODAL, G. S., FAGERBERG, R., AND VINTHER, K. 2007. Engineering a cache-oblivious sorting algorithm. *J. Exp. Algorithmics 12*, 2.2.

COLE, R. AND SHARIR, M. 1989. Visibility problems for polyhedral terrains. *J. Symb. Comput. 7*, 1, 11–30.

CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., AND STEIN, C. 2001. *Introduction to Algorithms*, 2nd ed. The MIT Press, Cambridge, Mass.

DANNER, A. 2007. Private communication.

DE FLORIANI, L. AND MAGILLO, P. 1994. Visibility algorithms on digital terrain models. *International Journal of Geographic Information Systems 8*, 1, 13–41.

DE FLORIANI, L. AND MAGILLO, P. 1999. *Geographic Information Systems: Principles, Techniques, Managament and Applications*. John Wiley and Sons, Chapter Intervisibility of Terrains, 543–556.

DEMENTIEV, R., KETTNER, L., AND SANDERS, P. 2005. Stxxl: Standard template library for xxl data sets. In *Proceedings of the European Symposium on Algorithms*. LNCS 3669. 640–651.

FISHER, P. 1993. Algorithm and implementation uncertainty in viewshed analysis. *International Journal of GIS 7*, 331–347.

FISHER, P. 1994. Stretching the viewshed. In *Proceedings of the Symposium on Spatial Data Handling*. 725–738.

FRANKLIN, W. R. 2002. Siting observers on terrain. In *Proceedings of the Symposium on Spatial Data Handling*.

FRANKLIN, W. R. AND RAY, C. 1994. Higher isn't necessarily better: Visibility algorithms and experiments. In *Proceedings of the Symposium on Spatial Data Handling*. 751–763.

FRIGO, M., LEISERSON, C. E., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*. 285–298.

GOODRICH, M. T., TSAY, J.-J., VENGROFF, D. E., AND VITTER, J. S. 1993. External-memory computational geometry. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*. 714–723.

TOMA, L. 2003. External memory graph algorithms and applications to geographic information systems. Ph.D. thesis, Duke University.
VAN KREVELD, M. 1996. Variations on sweep algorithms: efficient computation of extended viewsheds and class intervals. In *Proceedings of the Symposium on Spatial Data Handling*. 15–27.
VAN KREVELD, M., NIEVERGELT, J., ROOS, T., AND (EDS.), P. W. 1997. *Algorithmic Foundations of GIS*. LNCS 1340. Springer-Verlag.

Received May 2007; revised March 2008; accepted July 2008