A Parallel Algorithm for Viewshed Computation on Grid Terrains

Chaulio R. Ferreira[1], Marcus V. A. Andrade[1],
Salles V. G. Magalhães[1], W. R. Franklin[2], Guilherme C. Pena[1]

[1] Universidade Federal de Viçosa, Brazil
{chaulio.ferreira,marcus,salles,guilherme.pena}@ufv.br
[2] Rensselaer Polytechnic Institute, USA
wrf@ecse.rpi.edu

Abstract. Viewshed (or visibility map) computation is an important component in many GIScience applications and, as nowadays there are huge volume of terrain data available at high resolutions, it is important to develop efficient algorithms to process these data. Since the main improvements on modern processors come from multi-core architectures, parallel programming provides a promising means for developing faster algorithms. In this paper, we describe a new parallel algorithm based on the model proposed by Van Kreveld. Our algorithm uses the shared memory model, which is relatively cheap and supported by most current processors. Experiments have shown that, with 16 parallel cores, it was up to 12 times faster than the serial implementation, and up to 3.9 times using four parallel cores, which is an almost optimal speedup.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Parallel programming

Keywords: high performance computing, parallel algorithms, viewshed computation

1. INTRODUCTION

An important group of Geographical Information Science (GIScience) applications on terrains concerns visibility, i.e., determining the set of points on the terrain that are visible from some particular observer, which is usually located at some height above the terrain. This set of points is known as *viewshed* [Franklin and Ray 1994] and its applications range from visual nuisance abatement to radio transmitter siting and surveillance, such as minimizing the number of cellular phone towers required to cover a region [Ben-Moshe et al. 2007], optimizing the number and position of guards to cover a region [Magalhães et al. 2011], analyzing the influences on property prices in an urban environment [Lake et al. 1998] and optimizing path planning [Lee and Stucky 1998]. Other applications were presented by Champion and Lavery [Champion and Lavery 2002].

Since visibility computation is quite compute-intensive, the recent increase in the volume of high resolution terrestrial data brings a need for faster platforms and algorithms. Considering that some factors (such as processor physical size, transmission speed and economic limitations) create practical limits and difficulties for building faster serial computers, the parallel computing paradigm has become a promising alternative for computing-intensive applications [Barney 2010]. Also, parallel architectures have recently become widely available at low costs. Thus, they have been applied in many domains of engineering and scientific computing, allowing researchers to solve bigger problems in feasible amounts of time.

In this paper, we present a new parallel algorithm for computing the viewshed of a given observer

This research was partially supported by FAPEMIG, CAPES, CNPq and NSF.

on a terrain. Our parallel algorithm is based on the (serial) sweep line algorithm firstly proposed by Van Kreveld [1996], which is described in Section 2.3.3. Compared to the original algorithm, our new algorithm achieved speedup of up to 12 times using 16 parallel cores, and up to 3.9 times using four parallel cores, which is an almost optimal speedup.

The paper is organized as follows: In Section 2, we present some definitions for the addressed problem and review some important related work. In Section 3, we describe the proposed strategy for the parallel algorithm's implementation. Section 4 presents the results of the computational experiments and in Section 5 is presented the conclusions and some topics for future research.

2. RELATED WORK

2.1 Terrain representation

In what follows, our region of interest is small compared to the radius of the earth, thus, for this discussion the earth can be considered to be flat.

A *terrain* τ is a 2.5-dimensional surface where any vertical line intersects τ in at most one point. The terrain is usually represented approximately either by a *Triangulated Irregular Network (TIN)* or a *Raster Digital Elevation Model (DEM)* [Li et al. 2005]. A TIN is a partition of the surface into planar triangles, i.e., a piecewise linear triangular spline, where the elevation of a point p is a bilinear interpolation of the elevations of the vertices of the triangle containing the projection of p. On the other hand, a DEM is a matrix storing the elevations of regularly spaced positions or posts, where the spacing may be either a constant number of meters or a constant angle in latitude and longitude. In this paper, we will use the DEM representation because of its simpler data structure, ease of analysis, and ability to represent discontinuities (cliffs) more naturally. Finally, there is a huge amount of data available as DEMs.

2.2 The viewshed problem

An *observer* is a point in space from where other points (the *targets*) will be visualized. Both the observer and the targets can be at given heights above τ, respectively indicated by h_o and h_t. We often assume that the observer can see only targets that are closer than the *radius of interest*, ρ. We say that all cells whose distance from O is at most ρ form the *region of interest* of O. A target T is visible from O if and only if the distance of T from O is, at most, ρ and the straight line, the *line of sight*, from O to T is always strictly above τ; see Fig. 1.

The *viewshed* of O is the set of terrain points corresponding to the vertical projection of all targets that can be seen by O; formally,

$$viewshed(O) = \{p \in \tau \mid \text{the target above } p \text{ is visible from } O\}$$

with ρ implicit. The viewshed representation is a square $(2\rho+1) \times (2\rho+1)$ bitmap with the observer at the center.

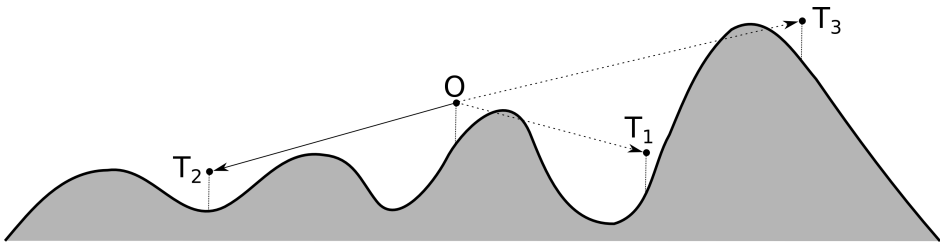


Fig. 1. Targets' visibility: T_1 and T_3 are not visible but T_2 is.

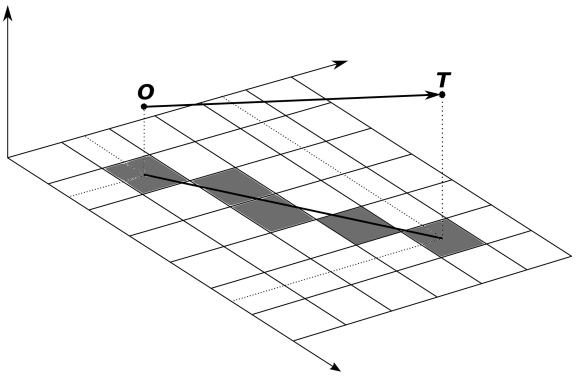


Fig. 2. The rasterization of the line of sight projection.

Theoretically, determining whether a target T is visible from O requires verifying all points in the line of sight connecting O to T. But since τ is represented with a finite resolution, only points close to the rasterized line segment connecting the projections of O and T onto the horizontal plane will be verified. Which points those might be, is one difference between competing algorithms, as the ones we will describe in Section 2.3. The visibility depends on the line segment rasterization method used, see Fig. 2, and how the elevation is interpolated on those cells where the segment does not intersect the cell center.

The visibility of a target above a cell c_t can be determined by checking the slope of the line connecting O and T and the cells' elevation on the rasterized segment. More precisely, suppose the segment is composed of cells $c_0, c_1, \cdots, c_t$ where c_0 and c_t correspond to the projections of O and T respectively. Let α_i be the slope of the line connecting O to c_i, that is,

$$\alpha_i = \frac{\zeta(c_i) - (\zeta(c_0) + h_o)}{dist(c_0, c_i)} \tag{1}$$

where $\zeta(c_0)$ and $\zeta(c_i)$ are, respectively, the elevation of cells c_0 and c_i and $dist(c_0, c_i)$ is the 'distance' (in number of cells) between these two cells. The target on c_t is visible if and only if the slope $\frac{\zeta(c_t)+h_t-(\zeta(c_0)+h_o)}{dist(c_0,c_t)}$ is greater than α_i for all $0 < i < t$. If yes, the corresponding cell in the viewshed matrix is set to 1; otherwise, to 0.

2.3 Viewshed algorithms

Different terrain representations call for different algorithms. A TIN can be processed by the algorithms proposed by Cole and Sharir [1989] and De Floriani and Magillo [2003]. For a DEM, we can point out RFVS [Franklin and Ray 1994] and the algorithm proposed by Van Kreveld [1996], two very efficient algorithms. Another option for processing DEMs is the well-known R3 algorithm [Shapira 1990]. Although this one is not as efficient as the other two, it has higher accuracy and may be suitable for small datasets.

These three algorithms differ from each other not only on their efficiency, but also on the visibility models adopted. For instance, R3 and Van Kreveld's algorithms use a center-of-cell to center-of-cell visibility, that is, a cell c is visible if and only if the ray connecting the observer (in the center of its cell) to the center of c does not intersect a cell blocking c. On the other hand, RFVS uses a less restrictive approach where a cell c may be considered visible if its center is not visible but another part of c is.

Therefore, the viewsheds obtained by these methods may be different. Some applications may prefer a viewshed biased in one direction or the other, while others may want to minimize error computed

under some formal terrain model. For instance, since Van Kreveld's algorithm presents a great tradeoff between efficiency and accuracy [Fishman et al. 2009], it may be indicated for applications that require a high degree of accuracy. On the other hand, if efficiency is more important than accuracy, the RFVS algorithm could be preferred.

Considering that each one of these algorithms might be suitable for different applications, we will describe them briefly in the next sections.

2.3.1 *R3 algorithm.* The R3 algorithm provides a straightforward method of determining the viewshed of a given observer O with a radius of interest ρ. Although it is considered to have great accuracy [Franklin et al. 1994], this algorithm runs in $\Theta(n^{\frac{3}{2}})$, where $n = \Theta(\rho^2)$. It works as follows: for each cell c inside the observer's region of interest, it uses the digital differential analyzer (DDA) [Maćiorov 1964] to determine which cells the line of sight (from O to the center of c) intersects. Then, the visibility of c is determined by calculating the slope of all cells intersected by this line of sight, as described in Section 2.2. In this process, many rules to interpolate the elevation between adjacent posts may be used, such as average, linear, or nearest neighbor interpolations.

2.3.2 *RFVS algorithm.* The RFVS algorithm [Franklin and Ray 1994] is a fast approximation algorithm that runs in $\Theta(n)$. It computes the terrain cells' visibility along rays (line segments) connecting the observer (in the center of a cell) to the center of all cells in the boundary of a square of side $2\rho + 1$ centered at the observer (see Fig. 3(a)). In each column, it tests the line of sight against the closest cell. Although a square was chosen for implementation simplicity, other shapes such as a circle would also work.

RFVS creates a ray connecting the observer to a cell on the boundary of this square, and then rotates it counter-clockwise around the observer to follow along the boundary cells (see Figure 3(a)). The visibility of each ray's cells is determined by walking along the segment, which is rasterized [Bresenham 1965]. Suppose the segment is composed of cells $c_0, c_1, \cdots, c_k$ where c_0 is the observer's cell and c_k is a cell in the square boundary. Let α_i be the slope of the line connecting the observer to c_i determined according to Equation (1) in Section 2.2. Let μ be the highest slope seen so far when processing c_i, i.e., $\mu = \max\{\alpha_1, \alpha_2, \cdots, \alpha_{i-1}\}$. The target sited on c_i is visible if and only if the slope $(\zeta(c_i) + h_t - (\zeta(c_0) + h_o))\,/dist(c_0, c_i)$ is greater than μ. If yes, the corresponding cell in the viewshed matrix is set to 1; otherwise, to 0. Also, if $\alpha_i > \mu$ then μ is updated to α_i. We say that a cell c_i blocks the visibility of the target sited on c_j if c_i belongs to the segment $\overline{c_0 c_j}$ and α_i is greater or equal to the slope of the line connecting the observer to the target above c_j.

2.3.3 *Van Kreveld's algorithm.* Van Kreveld's algorithm [Van Kreveld 1996] is another fast viewshed algorithm. According to Zhao et al. [2013], it has accuracy equivalent to the R3 algorithm, while running in $\Theta(n\ log\ n)$. Its basic idea is to rotate a sweep line around the observer and compute the visibility of each cell when the sweep line passes over its center (see Figure 3(b)). For that, it maintains a balanced binary tree (the *agenda*) that stores the slope of all cells currently being intersected by the sweep line, keyed by their distance from the observer. When this sweep line passes over the center of a cell c, the *agenda* is searched to check c's visibility.

More specifically, this algorithm defines three types of events: *enter*, *center*, and *exit* events to indicate, respectively, when the sweep line starts to intersect a cell, passes over the cell center and stops to intersect a cell. A list E is used to store these three types of events for all cells inside the region of interest. The events are then sorted according to their azimuth angle.

To compute the viewshed, the algorithm sweeps the list E and for each event it decides what to do depending on the type of the event:

—If it is an *enter* event, the cell is inserted into the *agenda*.

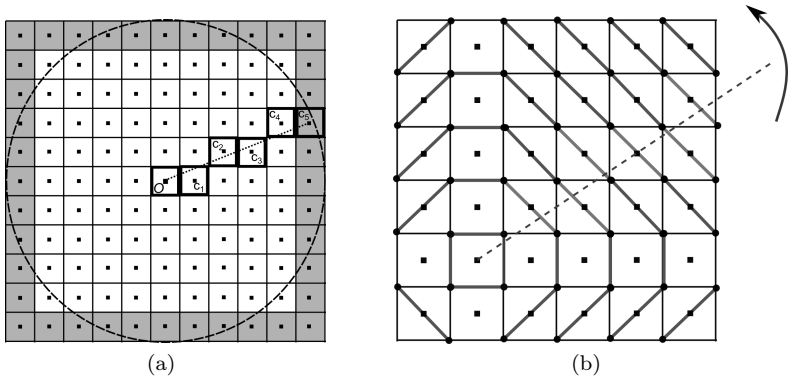


Fig. 3. Viewshed algorithms: (a) RFVS; (b) Van Kreveld - adapted from Fishman et al. [Fishman et al. 2009].

—If it is an *center* event of cell c, the *agenda* is searched to check if it contains any cell that lies closer to the observer than c and has slope greater or equal to the slope of the line of sight to c; if yes, then c is not visible, otherwise it is.

—If it is an *exit* event, the cell is removed from the *agenda*.

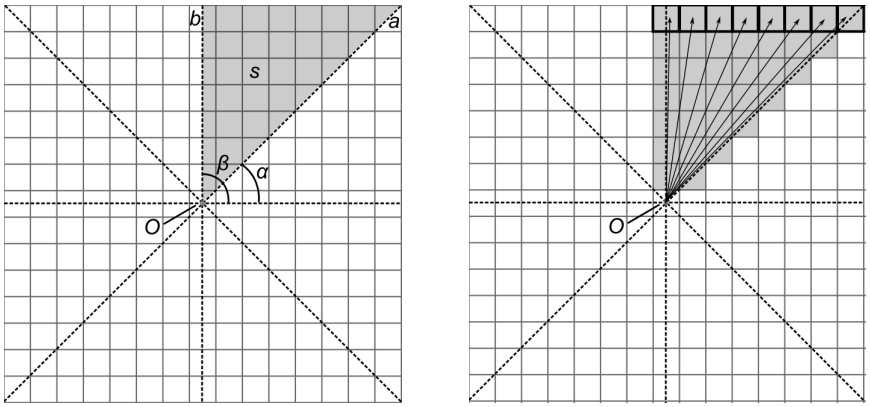
2.3.4 *Parallel viewshed algorithms.* Parallel computing has become a mainstream of scientific computing and recently some parallel algorithms for viewshed computation have been proposed. Zhao et al. [2013] proposed a parallel implementation of the R3 algorithm using Graphics Processing Units (GPUs). The RFVS algorithm was also adapted for parallel processing on GPUs by Osterman [2012]. Chao et al. [2011] proposed a different approach for parallel viewshed computation using a GPU, where the algorithm runs entirely within the GPUs visualization pipeline used to render 3D terrains. Zhao et al. [2013] also discuss other parallel approaches.

However, we have not found any previous work proposing a parallel implementation of Van Kreveld's algorithm. In fact, Zhao et al. [2013] stated that "a high degree of sequential dependencies in Van Kreveld's algorithm makes it less suitable to exploit parallelism". In Section 3 we show how we have overcome this difficulty and describe our parallel implementation of Van Kreveld's sweep line algorithm.

2.4 Parallel Programming models

There are several parallel programming models, such as distributed memory/message passing, shared memory, hybrid models, among others [Barney 2010]. In this work, we used the shared memory model, where the main program creates a certain number of tasks (*threads*) that can be scheduled and carried out by the operating system concurrently. Each thread has local data, but the main program and all threads share a common address space, which can be read from and written to asynchronously. In order to control the concurrent access to shared resources, some mechanisms such as locks and semaphores may be used. An advantage of this model is that there is no need to specify explicitly the communication between threads, simplifying the development of parallel applications.

For the algorithm's implementation, we used OpenMP (*Open Multi-Processing*) [Dagum and Menon 1998], a portable parallel programming API designed for shared memory architectures. It is available for C++ and Fortran programming languages and consists of a set of compiler directives that can be added to serial programs to influence their run-time behaviour, making them parallel.



(a) (b)

Fig. 4. Sector definition: (a) The subdivision of the region of interest and the sector s, defined by the interval $[\alpha, \beta)$; (b) The cells in the perimeter of the region of interest, the rays used to determine which cells are intersected by s and the cells inserted into E_s (shaded cells).

3. OUR PARALLEL SWEEP LINE ALGORITHM

As described in Section 2.3.3, Van Kreveld's algorithm needs information about the cells intersected by the sweep line. It maintains these information by processing the *enter* and *exit* events to keep the *agenda* up to date as the sweep line rotates. Therefore, processing a *center* event is dependent upon all earlier *enter* and *exit* events.

In order to design a parallel implementation for this algorithm, this dependency had to be eliminated. We did that by subdividing the observer's region of interest in S sectors around the observer, O (see Figure 4(a), where $S = 8$). Our idea is to process each one of these sectors independently using Van Kreveld's sweep line algorithm, such that it can be done in parallel.

More specifically, consider sector s defined by the interval $[\alpha, \beta)$, where α and β are azimuth angles. Let a and b be the line segments connecting O to the perimeter of its region of interest, with azimuth angles α and β, respectively (see Figure 4(a)). To process s, the algorithm creates rays connecting O to all cells on the perimeter of the region of interest that are between (or intersected by) a and b (see Figure 4(b)). These rays are rasterized using the DDA method [Maćiorov 1964] and the events related to the intersected cells are inserted into s's own list of events, E_s. Since the grid cells are convex, this process inserts into E_s the events for all cells inside s or intersected by a or b. The inserted cells are shown in Figure 4(b). To efficiently avoid creating the events for a same cell (in a same sector) more than once, we used a $O(1)$ geometric verification: if a cell intersected by the current ray had also been intersected by the previous ray, that cell is ignored.

Then, the algorithm sorts E_s by the events' azimuth angles and sweeps it in the same manner as Van Kreveld's algorithm. Note that, because we have distributed the events into different lists and each list contains all events that are relevant to its sector, each sector may be processed independently, each one with its own *agenda*. This allows a straightforward parallelization of such processing. Also, note that the events of a cell may be included in more than one sector's event list and therefore some cells may be processed twice. But that is not a problem, since this will happen only to a few cells, and it will not affect the resulting viewshed.

It is also important to note that our algorithm might be faster than the original one even with non-parallel architectures. For instance, we achieved up to 20% speedup using only one processor (see Section 4). This happens because both implementations have to sort their lists of events and, while the original (serial) algorithm sorts a list of size n, our algorithm sorts S lists of size about $\frac{n}{S}$. Since

sorting can be done in $\Theta(n\ logn)$, the latter one is faster. In practice, we empirically concluded that, for a computer with N cores, using $S > N$ achieved better results than using $S = N$. This will be further discussed in Section 4, as long with our experimental results.

4. EXPERIMENTAL RESULTS

We implemented our algorithm in C++ using OpenMP. We also implemented the original (serial) Van Kreveld's algorithm in C++. Both algorithms were compiled with g++ 4.6.4 and optimization level -O3. Our main (fastest) experimental platform was a Dual Intel Xeon E5-2687 3.1GHz 8 core with 160GiB RAM. The operational system was Ubuntu 12.04 LTS, Linux 3.5 Kernel. We also made some experiments on a simpler personal computer – see Section 4.1.

The tests on the fastest architecture were done using six different terrains from SRTM datasets and the observer was sited in the center of the terrain, with $h_o = 100$ meters and $h_t = 0$. The radius of interest, ρ, was set to be large enough to cover the whole terrain.

Another important parameter for our program is the number of sectors S into which the region of interest will be subdivided. Changing the number of sectors may significantly modify the algorithm's performance. Empirically, we determined that good results are achieved when the region is subdivided such that each sector contained about 40 cells from the perimeter of the region of interest, so we adopted that strategy. Other strategies for choosing the number of sectors should be further investigated and it could be an interesting topic for future work.

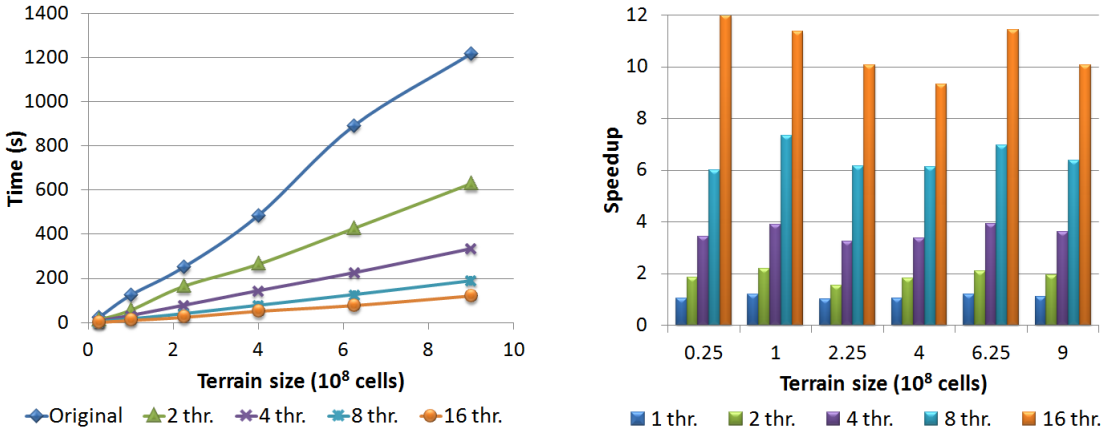
To evaluate our algorithm's performance, we compared it to the original (serial) algorithm. We ran several experiments limiting the number of parallel threads to the following values: 16, 8, 4, 2 and 1. The results are given in Table I and plotted in Figure 5(a), where the times are given in seconds and refer just to the time needed to compute the viewshed. That is, we did not consider the time taken to load the terrain data and to write the computed viewshed into disk, since it was insignificant (less than 1% of the total time in all cases). Also, the time represents the average time for five different runs of the same experiment.

We calculated our algorithm's speedup compared to the original algorithm. They are also shown in Table I (in parenthesis) and plotted in Figure 5(b). Our algorithm has shown very good performance, achieving up to 12 times speedup, when running 16 concurrent threads. Also, as discussed in Section 3, the experiments with only one thread show that our strategy can be faster than the original program even with serial architectures.

To assess how some different terrain features can the affect computational performance, we made further experiments siting the observer at locations with different topographic characteristics. More specifically, for each of the six terrains used on the experiments above, we defined three points: one in a "flat" region (i.e. a region with no significant elevation variations), one on a peak of a mountainous

Table I. Running times (in seconds) for the serial algorithm and the parallel algorithm with different number of threads. Speedups of the parallel algorithm compared to the serial one are shown in parenthesis.

Terrain size		Serial Alg.	Parallel Alg. Number of threads									
# cells	GiB		1		2		4		8		16	
$5\,000^2$	0.09	24	23	(1.04)	13	(1.85)	7	(3.43)	4	(6.00)	2	(12.00)
$10\,000^2$	0.37	125	105	(1.19)	57	(2.19)	32	(3.91)	17	(7.35)	11	(11.36)
$15\,000^2$	0.83	252	246	(1.02)	165	(1.53)	78	(3.23)	41	(6.15)	25	(10.08)
$20\,000^2$	1.49	485	464	(1.05)	265	(1.83)	144	(3.37)	79	(6.14)	52	(9.33)
$25\,000^2$	2.33	891	740	(1.20)	427	(2.09)	226	(3.94)	128	(6.96)	78	(11.42)
$30\,000^2$	3.35	1 216	1 100	(1.11)	629	(1.93)	335	(3.63)	191	(6.37)	121	(10.05)



(a) (b)

Fig. 5. (a) Running (in seconds) times for the serial algorithm and the parallel algorithm with different number of threads; (b) Speedups achieved by our parallel algorithm, with different number of threads.

Table II. Running times (in seconds) for both algorithms siting the observer on regions with three different characteristics (pit, flat and peak) of a same terrain. The parallel algorithm was run with 16 parallel threads.

Terrain size # cells	GiB	Viewpoint type	Radius (ρ)	# visible cells	Serial	Parallel	Speedup
$5\,000^2$	0.09	Pit	435	55730	0.52	0.22	2.36
		Flat	435	97077	0.52	0.23	2.26
		Peak	435	163860	0.56	0.22	2.55
$10\,000^2$	0.37	Pit	1500	437908	6.41	1.30	4.93
		Flat	1500	901124	6.72	1.34	5.01
		Peak	1500	936784	6.66	1.37	4.86
$15\,000^2$	0.83	Pit	3400	55901	35.85	5.39	6.65
		Flat	3400	2666378	39.79	5.60	7.11
		Peak	3400	8401033	36.06	5.99	6.02
$20\,000^2$	1.49	Pit	5500	3985689	120.02	15.12	7.94
		Flat	5500	7643595	113.86	15.70	7.25
		Peak	5500	6479386	114.38	15.07	7.59
$25\,000^2$	2.33	Pit	7000	432670	188.09	24.71	7.61
		Flat	7000	1702399	195.39	25.11	7.78
		Peak	7000	11099771	194.27	25.45	7.63
$30\,000^2$	3.35	Pit	12000	113669	555.19	70.81	7.84
		Flat	12000	19936775	608.01	72.28	8.41
		Peak	12000	30972487	558.65	71.93	7.77

region, and one on a pit of a mountainous region. Each of these three points was used as the observer position on a experiment using both (serial and parallel) algorithms. All experiments with the parallel algorithm were made running 16 parallel threads. Also, to provide a fair comparison between the running times, for each terrain we chose a radius of interest (ρ) such that all three observers had their region of interest entirely inside the terrain. If we did not so, the running times could be influenced because of non-uniform sectors divisions (for the parallel algorithm) and also because of some non-processed cells (for both algorithms). The results of these experiments for both algorithms and the obtained speedups are presented in Table II.

Notice that there is no correlation between the number of visible cells in a computed viewshed and the time spent to compute it. Thus, for both algorithms we can conclude that the terrain and its

Table III. Running times (in seconds) for the serial algorithm and the parallel algorithm with different number of threads on a personal computer and the achieved speedup using 4 threads.

Terrain size		Serial Alg.	Parallel Alg. Number of threads			Speedup 4 thr.
# cells	MiB		1	2	4	
$1\,000^2$	3.81	1.16	1.23	0.73	0.51	2.27
$2\,000^2$	15.3	4.67	5.74	2.66	1.59	2.94
$3\,000^2$	34.3	11.16	15.52	6.27	3.66	3.05
$4\,000^2$	61.0	22.32	21.11	11.48	6.79	3.29
$5\,000^2$	95.4	34.61	33.82	18.20	10.72	3.23
$6\,000^2$	137	49.67	49.78	26.78	15.81	3.14
$7\,000^2$	187	74.66	69.53	36.93	23.22	3.22
$8\,000^2$	244	98.08	92.76	90.22	29.70	3.30

topographic characteristics have no influence on the execution time.

4.1 Experiments on a personal computer

To evaluate our algorithm's performance on a simpler architecture, we made some experiments using a personal computer with four cores: a Intel Core i5-3330 3.00GHz with 8.00 GiB RAM. The operational system was Ubuntu 12.10, Linux 3.5 Kernel.

In these experiments, we also sited the observer in the center of the terrain and computed the viewshed of the whole terrain. The number of parallel threads was limited to 4, 2 and 1. Considering the available RAM memory on this computer (8GiB), we could execute the experiments on terrains with up to 8000^2 cells. In Table III, we present the running times for these experiments and the achieved speedup considering the experiments using 4 threads.

The results of these experiments are very similar to the results of the experiments with our fastest computer. Even with only four threads, we were able to achieve noticeable speedups, often more than 3 times faster than the serial algorithm. Thus, considering that processors with four cores have become usual and relatively cheap nowadays, our algorithm can considerably accelerate viewshed computation on regular computers.

5. CONCLUSIONS AND FUTURE WORK

We proposed a new parallel sweep line algorithm for viewshed computation, based on an adaptation of Van Kreveld's algorithm. Compared to the original (serial) algorithm, on our fastest computer we achieved speedup of up to 12 times with 16 concurrent threads, and up to 3.9 times using four threads. Even with a single thread, our algorithm was faster than the original one, running up to 20% faster. We also evaluated experiments on a simpler personal computer with 4 cores, achieving up to 3.3 times.

Compared to other parallel viewshed algorithms, ours seems to be the only that uses Van Kreveld's model, which presents a great tradeoff between efficiency and accuracy [Fishman et al. 2009]. Also, most of them use other parallel computing models, such as general purpose GPU programming. On the other hand, ours uses the shared memory model, which is simpler, requires cheaper architectures and is supported by most current computers.

A drawback for Van Kreveld's algorithm (and our parallel implementation as well) is that it requires too much memory. On both (serial and parallel) implementations, each event requires 21 bytes. Thus, each terrain cell requires 63 bytes for its three events. Because of this, using a personal computer with 8GiB RAM, we could only process terrains with up to 244 MiB. As a future work, we propose

trying to optimize memory usage and/or using external memory strategies to allow the algorithms to process larger terrains.

Also as future work, we propose the development of another adaptation of Van Kreveld's model using GPU programming. Since modern GPUs have thousands of cores, one could obtain much greater speedups than we did. However, that will not be a straightforward adaptation, because GPU architectures are much different from multi-core CPU architectures. For example, within a GPU it is not possible to use pointers to implement the balanced binary tree Van Kreveld's algorithm uses. Thus, this future work will require a more detailed study on GPUs architectures and their different memory layers characteristics.

REFERENCES

Barney, B. Introduction to Parallel Computing. *Lawrence Livermore National Laboratory* 6 (13): 10, 2010.

Ben-Moshe, B., Ben-Shimol, Y., and Y. Ben-Yehezkel, A. Dvir, M. S. Automated Antenna Positioning Algorithms for Wireless Fixed-Access Networks. *Journal of Heuristics* 13 (3): 243–263, 2007.

Bresenham, J. An Incremental Algorithm for Digital Plotting. *IBM Systems Journal* 4 (1): 25–30, 1965.

Champion, D. C. and Lavery, J. E. Line of Sight in Natural Terrain Determined by L_1-Spline and Conventional Methods. In *Proceedings of the Army Science Conference*. Orlando, Florida, 2002.

Chao, F., Chongjun, Y., Zhuo, C., Xiaojing, Y., and Hantao, G. Parallel Algorithm for Viewshed Analysis on a Modern GPU. *International Journal of Digital Earth* 4 (6): 471–486, 2011.

Cole, R. and Sharir, M. Visibility Problems for Polyhedral Terrains. *Journal of Symbolic Computation* 7 (1): 11–30, 1989.

Dagum, L. and Menon, R. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE* 5 (1): 46–55, 1998.

De Floriani, L. and Magillo, P. Algorithms for Visibility Computation on Terrains: a survey. *Environment and Planning B: Planning and Design* 30 (5): 709–728, 2003.

Fishman, J., Haverkort, H. J., and Toma, L. Improved Visibility Computation on Massive Grid Terrains. In *Proceedings of the International Conference on Advances in Geographic Information Systems*. Seatle, Washington, USA, pp. 121–130, 2009.

Franklin, R., Ray, C. K., Randolph, P. W., Ray, C. K., and Mehta, S. Geometric Algorithms for Siting of Air Defense Missile Batteries. Tech. rep., Rensselaer Polytechnic Institute, 1994.

Franklin, W. R. and Ray, C. Higher Isn't Necessarily Better: visibility algorithms and experiments. In *Proceedings of the International Symposium on Spatial Data Handling*. Edinburgh, Scotland, pp. 751–770, 1994.

Lake, I. R., Lovett, A. A., Bateman, I. J., and Langford, I. H. Modelling Environmental Influences on Property Prices in an Urban Environment. *Computers, Environment and Urban Systems* 22 (2): 121–136, 1998.

Lee, J. and Stucky, D. On Applying Viewshed Analysis for Determining Least-Cost Paths on Digital Elevation Models. *International Journal of Geographical Information Science* 12 (8): 891–905, 1998.

Li, Z., Zhu, Q., and Gold, C. *Digital Terrain Modeling — principles and methodology*. CRC Press, 2005.

Maćiorov, F. *Electronic digital integrating computers: digital differential analyzers*. Iliffe Books (London and New York), 1964.

Magalhães, S. V. G., Andrade, M. V. A., and Franklin, W. R. Multiple Observer Siting in Huge Terrains Stored in External Memory. *International Journal of Computer Information Systems and Industrial Management* vol. 3, pp. 143–149, 2011.

Osterman, A. Implementation of the r.cuda.los Module in the Open Source GRASS GIS by Using Parallel Computation on the NVIDIA CUDA Graphic Cards. *Elektrotehniëski Vestnik* 79 (1-2): 19–24, 2012.

Shapira, A. Visibility and terrain labeling. *Master's thesis, Rensselaer Polytechnic Institute*, 1990.

Van Kreveld, M. Variations on sweep algorithms: efficient computation of extended viewsheds and class intervals. In *Proceedings of the Symposium on Spatial Data Handling*. Delft, Netherlands, pp. 15–27, 1996.

Zhao, Y., Padmanabhan, A., and Wang, S. A parallel computing approach to viewshed analysis of large terrain data using graphics processing units. *International Journal of Geographical Information Science* 27 (2): 363–384, 2013.