International Journal of Computational Geometry & Applications

PARALLELIZING AN ALGORITHM FOR VISIBILITY ON POLYHEDRAL TERRAIN

Y. ANSEL TENG*
Center for Automation Research, University of Maryland
College Park, MD 20742, U.S.A.

DAVID MOUNT
Department of Computer Science, University of Maryland
College Park, MD 20742, U.S.A.

ENRICO PUPPO
Istituto per la Matematica Applicata, Consiglio Nazionale delle Ricerche
Via De Marini, 6 (Torre di Francia), Genova 16149, ITALY

LARRY S. DAVIS
University of Maryland Institute for Advanced Computer Studies
College Park, MD 20742, U.S.A.

Received 1 December 1993
Revised 19 September 1994
Communicated by Esther M. Arkin

ABSTRACT

The best known output-sensitive sequential algorithm for computing the viewshed on a polyhedral terrain from a given viewpoint was proposed by Katz, Overmars, and Sharir [10], and achieves time complexity $O((k + n\alpha(n)) \log n)$ where n and k are the input and output sizes respectively, and $\alpha()$ is the inverse Ackermann's function. In this paper, we present a parallel algorithm that is based on the work mentioned above, and achieves $O(\log^2 n)$ time complexity, with work complexity $O((k + n\alpha(n)) \log n)$ in a CREW PRAM model. This improves on previous parallel complexity while maintaining work efficiency with respect to the best sequential complexity known.

Keywords: Terrain visibility, parallel algorithms, CREW PRAM, profile tree

1. Introduction

Visibility analysis on terrain is a fundamental problem in computer graphics, navigation, and engineering applications. Several aspects of this problem have been

*Permanent address: Jet Propulsion Lab, 4800 Oak Grove Dr. MS 168-522, Pasadena, CA 91109-8099

The support of the Advanced Research Projects Agency (ARPA Order No. 8459) and the U.S. Army Engineer Topographic Laboratories under Contract DACA76-92-C-0009 is gratefully acknowledged.

faced by many researchers during the last twenty years, from both a practical and a theoretical point of view. For an extensive survey on this subject see [12].

In this paper, we consider the problem of computing in parallel the *viewshed* (i.e., the visible portion) of a terrain from any given viewpoint. We concentrate on polyhedral terrain models, in which the terrain surface is represented by polygonal patches. In this scenario, we define the *size* of a geometric object as the number of its edges.

Major theoretical results that are relevant to our subject are outlined in the following. Cole and Sharir [6] established an $O(n\alpha(n))$ upper bound on the size of the horizon resulting from n edges, where $\alpha()$ is the functional inverse of the Ackermann's function; they also showed that, in the worst case, the size of the viewshed is $\Omega(n^2)$ for an input terrain of size n. However, in most cases the output size is far less than $O(n^2)$. This fact motivated researchers to consider output-sensitive algorithms. Reif and Sen [13] proposed one such algorithm, whose running time is $O((k+n)\log n \log\log n)$. They also designed a parallel algorithm whose running time is $O(\log^4 n)$ using $O((k+n)/\log n)$ processors in a CREW PRAM model. The best known time complexity for an output-sensitive sequential algorithm is due to Katz, Overmars, and Sharir [10]: they propose a general technique for hidden surface removal, that achieves an $O((k+n\alpha(n))\log n)$ running time for the viewshed of a polyhedral terrain.

Here, we present a parallel algorithm, based on the work of Katz, Overmars, and Sharir, using a *profile tree*. Its time complexity is $O(\log^2 n)$ using $O((k+n\alpha(n))/\log n)$ processors in a CREW PRAM model, hence the same work complexity as the sequential algorithm, and an improvement in time complexity over previous parallel algorithms. When processor efficiency is ignored, time complexity can be further reduced to $O(\log^2 n/\log\log n)$ using $O((k+n\alpha(n)))$ processors.

2. Preliminaries

A *polyhedral terrain* T is the graph of a piecewise linear continuous function $z = f(x,y)$ defined over the X-Y plane. Let V and E be the vertices and edges of T, respectively. The projections of V and E on the X-Y plane define a planar straight-line graph (PSLG). Assume that T has n edges (hence, $O(n)$ vertices and faces), and that the input is given as a list of edges. The list of vertices and the adjacency list of each vertex can be derived easily.

Given a viewpoint v above T, and a generic point p on T, the straight-line segment connecting p to v is called the *line-of-sight* to p. For simplicity, assume that the viewpoint is located at $y = -\infty$, and that all the vertices of T have bounded positive y coordinates. Thus, the line-of-sight to p is a ray parallel to the Y axis, extending from p toward the negative y direction, and intersecting the X-Z plane[a]. For clarity of presentation, we finally assume that each vertex has a distinct x-coordinate, therefore no edges are parallel to the viewing direction.

[a]A visibility problem with a general viewpoint at finite position can be easily transformed to this form by using spherical coordinates (R, θ, ϕ) with the viewpoint as the origin: see [6] for further details.

A point p on the terrain is visible if and only if the line-of-sight to p does not intersect the terrain except at p. For each polygonal patch t of T, the portion I_t of t visible from v, if non-empty, consists of one or more simple polygons. The viewshed problem is to find $I \equiv \{I_t | t \in T\}$. The output size, i.e., the size of I, will be denoted as k.

In our algorithm, we usually refer to geometric objects by their projections on either the X-Y plane or the X-Z plane: we will call the *print* of an object its projection on the X-Y plane, and the *image* of an object its projection on the X-Z plane, respectively. Therefore, we refer to the X-Z plane as the *screen*.

Given two objects A and B, we will say that B is *in front* of A if and only if there exist any line-of-sight to A whose print intersects the print of B. We will say that B is *immediately in front* of A if and only if B is in front of A and there exist no object C such that B is in front of C, and C is in front of A. The *in front* relation is a partial order on T.

Note that the image of an edge is obtained by intersecting the screen with the lines-of-sight of such an edge. In the following sections, we will associate each edge with the unbounded trapezoid below it, called a *curtain*. It is straightfoward to see that the image of the visible portion of a curtain below an edge e is coincident with the image of the visible portion of the face of T immediately in front of e.

Our algorithm actually computes the image of the viewshed I. However, since the identity of each element in the image is known, the actual position of I on the terrain can be obtained immediately.

3. Parallel viewshed algorithm

This Section is subdivided into three parts. In 3.1, we briefly outline a technique proposed in [10], that is the basis of our parallel algorithm. In 3.2, we describe a parallel algorithm for the topological sort of edges, based on previous work by Atallah et al. [2,3], that constitutes the first step of the viewshed algorithm. In 3.3, we describe the remaining steps of our algorithm, and we conclude its analysis.

3.1. The basic technique

Katz, Overmars, and Sharir [10] propose algorithms for computing the visible portion of a set of non-intersecting objects in space. Their basic technique consists of the following steps:

(i) Input objects are topologically sorted by depth order (i.e., according to the *in front* relation) and stored in this order in the leaves of a balanced binary tree $\mathcal{T}$, the object nearest to the viewpoint in the leftmost leaf.
(ii) Two divide-and-conquer passes over $\mathcal{T}$ are performed, and the following two sets of polygonal regions are computed for each node d of $\mathcal{T}$, respectively:
  - U_d is the union of the images of all objects in the subtree rooted at d;
  - V_d is the portion of U_d that is visible.

U_d is computed in a bottom-up manner. For a leaf node, U_d is the object itself. For an internal node, let $lson(d)$ and $rson(d)$ be the left and right children of d

respectively. Therefore,

$$U_d = U_{lson(d)} \bigcup U_{rson(d)}. \tag{1}$$

V_d is computed in a top-down manner, by means of the following formulae:

$$\begin{aligned} V_{root} &= U_{root} \\ V_{lson(d)} &= V_d \bigcap U_{lson(d)} \\ V_{rson(d)} &= V_d - U_{lson(d)}. \end{aligned} \tag{2}$$

At the end of the second pass, the set V_d associated to each leaf node contains the visible portion of its corresponding object. More specifically, each leaf contains the image on the screen of the visible region, but the corresponding region on the object can be obtained immediately. Therefore, the result is simply obtained as the union of V_d for all leaf nodes.

3.2. Parallel topological sorting

In our case, input objects correspond to the curtains below the edges of T. Sorting such curtains by depth means obtaining an ordered sequence of curtains such that a curtain with higher position in the sequence will not obstruct the visibility to any curtain with lower position in the sequence. Since such objects are all perpendicular to the X-Y plane, they can be topologically sorted in depth order by working on their prints on the X-Y plane. Thus, we want to obtain a sequence of edges such that, if we call $rank(e)$ the position of an edge e in such an ordered sequence, the following condition holds:

> for any two edges e_1 and e_2, if $rank(e_1) < rank(e_2)$, then e_2 will not be in front of e_1.

Finding one such ordered sequence is a typical topological sorting problem for the set of edges E. Note that with a viewpoint located at infinity, any set of non-intersecting edges can be topologically sorted by depth order, i.e., the relation above is always acyclic [6]. For a viewpoint at finite position this is not always true. Cole and Sharir show a preprocessing method for eliminating cyclicity [6]. Note, anyway, that for *projective* subdivisions (i.e., plane subdivisions obtained as the print of the lower part of a polyhedron in space) the visibility relation is always acyclic, independently of the position of the viewpoint [7]. For polyhedral terrains described by one such subdivision (e.g., a Delaunay triangulation) the algorithm described in the following can be easily adapted for any viewpoint without preprocessing.

Atallah et al. [2] proposed a method for topological sorting of image features on a mesh-connected computer. Later, Atallah [3] outlined a generalization of this method for topological sorting of line segments on PRAM models. Based on such a generalization, the topological order of the prints of edges is derived through the following steps, by using a processor per edge print:

(i) An auxiliary edge is inserted on the X-Y plane, that lies closer to the viewpoint than any of the edge prints, and spans the entire range of x-coordinates of the vertices. This is done in constant time.

(ii) For each edge print e, the edge print that lies immediately in front of its right vertex is found, and it is called its *parent*. For a general PSLG, finding all parent edges requires a trapezoidal decomposition, which can be performed optimally in $O(\log n)$ time using $O(n)$ processors [1]. In case the print of terrain T is a triangulation, this step is even simpler: the parent of an edge is either the clockwise predecessor at its right vertex, or the clockwise successor at its left vertex. If we arrange the adjacency lists in a clockwise order, the parent can be found immediately.

(iii) A directed graph $\hat{G}$ is built, whose vertices are the edges of E and whose edges represent the parent relations in E. If e_1 is the parent of e_2, then (e_1, e_2) is a directed edge in $\hat{G}$. $\hat{G}$ is a tree having the auxiliary edge as root. Since each processor already knows its parent, $\hat{G}$ actually comes free from the previous step.

(iv) For each node in $\hat{G}$, its outgoing edges are ordered by the x coordinates of the right vertices of its children. This can be done in $O(\log n)$ time using $O(n)$ processors, through any optimal sorting algorithm (e.g., [5]).

(v) A pre-order traversal of the tree $\hat{G}$ will give a valid depth order. Since $\hat{G}$ has exactly n edges, a preorder traversal can be performed in $O(\log n)$ time and $O(n)$ processors using the Euler tour technique [14].

In summary, the initial topological sorting of input edges can be preformed in $O(\log n)$ time using $O(n)$ processors in a CREW PRAM model, i.e., with optimal $O(n \log n)$ work.

The edges permuted according to their depth order constitute the leaf level of the tree $\mathcal{T}$ defined in 3.1. The internal levels of the tree can be constructed in the bottom-up computation of U_d. Actually, tree $\mathcal{T}$ is only an abstraction of the relation between data. Since it is a balanced binary tree, all the pointers between data can be derived by arithmetics, therefore an explicit "construction" of $\mathcal{T}$ is not necessary.

3.3. Parallel visibility using profile tree

Although formulae (1) and (2) can be applied to more general cases that involve non-intersecting 3-d objects, for a polyhedral terrain the computation can be further simplified.

In our case, the set U_d for a given node d of $\mathcal{T}$ is defined as the unbounded region below the upper envelope of the edges in the subtree rooted at d, i.e., a *profile*. For this reason, $\mathcal{T}$ will be called a *profile tree*. We further observe that the boundary of any region V_d consists of one or more polygons that are always monotone with respect to the X-axis, that is, if we draw a line on the screen perpendicular to the X-axis, its intersection with V_d is always a single interval. Therefore, the boundary of each such polygon can be decomposed into an upper part and a lower part. Note that the upper boundaries of these polygons are part of the upper boundary of U_d. Since in the leaf nodes U_d is just the edge stored in that node, we need only keep the lower boundary during the computation. Define the *shadow* corresponding to a node d, denoted as S_d, to be the lower boundary of V_d. The formulae (2) given for

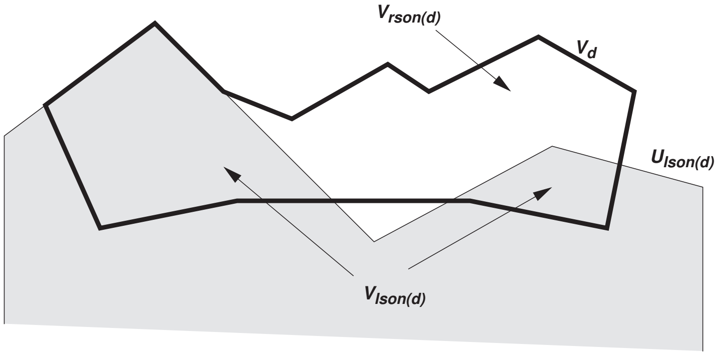

Fig. 1. Computing $V_{lson(d)}$ and $V_{rson(d)}$ from V_d. V_d is the polygon enclosed by the thick solid line and $U_{lson(d)}$ is the shaded area. The shaded area within V_d belongs to $V_{lson(d)}$ while the unshaded area belongs to $V_{rson(d)}$.

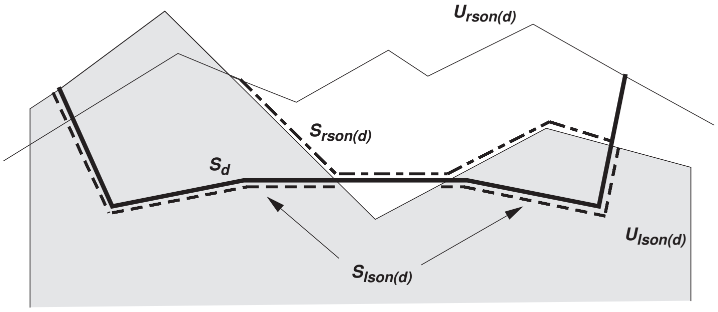

Fig. 2. Computing the shadows of the child nodes. This is the same example as in Figure 1 with $U_{rson(d)}$ shown. S_d is indicated by the thick solid lines. $S_{lson(d)}$ consists of the dashed segments below S_d while $S_{rson(d)}$ consists of those above S_d.

V_d can then be replaced by the following rules:

(i) S_{root} is an imaginary edge with infinite length and height $-\infty$;
(ii) $S_{lson(d)} = S_d$ when S_d is lower than $U_{lson(d)}$; otherwise it is undefined;
(iii) $S_{rson(d)} = max(S_d, U_{lson(d)})$ when $U_{lson(d)}$ is lower than $U_{rson(d)}$; otherwise it is undefined;

The formulae (2) are illustrated in Figure 1 for the case of a terrain. Figure 2 illustrates the rules for computing the shadows of the child nodes in the same situation as Figure 1. Note that if S_d is not defined in an interval of x-coordinates, node d does not have any visible portion in that interval.

Now we take a closer look at these set operations and explain how to perform them in parallel. At each node, the operations involved in computing $U_d, S_{lson(d)}$, and $S_{rson(d)}$ can all be described as merging two monotone chains of line segments. The difference is whether to report the upper or the lower segment in each interval. Since the computation at each node is independent from those at other nodes at the same level, these terms can be computed for all nodes at the same level of the tree in parallel.

We illustrate the computation of U_d in Figure 3. A profile is represented by the

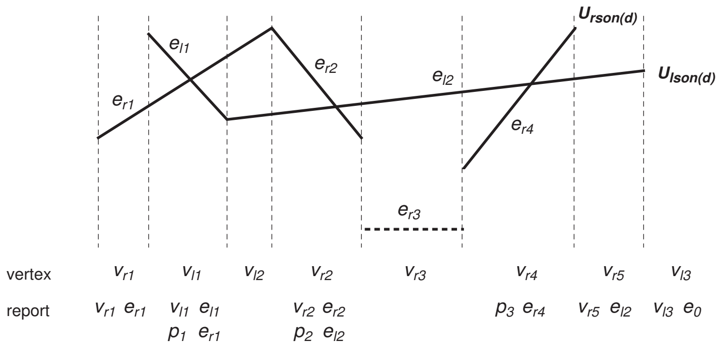

Fig. 3. Merging $U_{lson(d)}$ and $U_{rson(d)}$ into U_d. The screen is partitioned into intervals determined by the x-coordinates of the vertices. The row labeled "vertex" shows the name of the vertex that starts the interval. Vertices are named in a way such that a segment and its left endpoint has the same index. The row labeled "report" shows the vertex-segment pair(s) reported in each interval. U_d is represented by the list of the reported pairs. p_1, p_2, and p_3 are the new vertices due to intersecting edges. In the last interval, e_0 is the imaginary segment implied by a null pointer.

endpoints of its segments sorted in ascending order of their x-coordinates. Each vertex has a pointer to the edge that embeds the next segment. Note that a vertex in a profile is not necessarily a vertex of T. It may be the intersection of two edge images. Also note that a profile may consist of several disconnected chains. In such cases, a null pointer will mark the interval in which the profile is disconnected and it can be interpreted as a segment with height $-\infty$. For example, $U_{rson(d)}$ in Figure 3 consists of two chains, and an imaginary segment e_{r3} is inserted to maintain a uniform representation.

Let n_1 and n_2 be the number of vertices in the profile of $U_{lson(d)}$ and $U_{rson(d)}$ respectively. The lists of vertices are merged according to their x-coordinates. The x-coordinates of these vertices partition the x-axis into intervals. In order to determine U_d in each interval, each vertex needs to compare the two segments in the child profiles that span the interval to its right. The rank of each vertex in its original profile is stored with it in the new list, so the vertex knows the segment in its original profile. By subtracting its old rank from its new rank, each vertex determines the segment in the other profile. For example, the rank of vertex v_{l2} in Figure 3 is 3 in the merged list and 2 in $U_{lson(d)}$, so the segments for the third interval are e_{l2} and e_{r1}. Therefore, each vertex can compare the two segments and the higher one will be in U_d. Since several consecutive intervals may report the same segment for U_d, we should avoid storing them redundantly in such a case. A segment, together with the vertex that starts the interval, is reported only if the preceding interval has a different higher segment. If two segments intersect within an interval, the intersection and the higher segment starting at the intersection are also reported. The pairs of reported vertices and segments represent U_d in the format mentioned above, so the operation can be applied iteratively over the tree.

As for the complexity, merging two sorted lists of length $O(m)$ takes $O(\log\log m)$ time and $O(m)$ operations using $O(m)$ processors on a CREW PRAM model [11]. After the merging, computing the intersections and determining the segments in each interval takes only $O(1)$ time. Since the number of segments in the profile of n edges is of $O(n\alpha(n))$ [6], computing U_d over one level of the tree takes time $O(\log\log n)$ with $O(n\alpha(n))$ operations. However, in order to conduct these operations level by level, we need to reallocate processors since some vertices in the child nodes will no longer be in U_d and some new vertices are created due to intersecting segments. That is, we have to pack the reported vertices at the end of each merging operation so that they occupy consecutive locations. $O(m)$ processors can be reallocated in $O(\log m)$ time, because such a task takes a constant number of parallel prefix operations [9]. Thus, this constraint adds a factor of $O(\log n)$ to the time complexity at each level during the upward computation. Since the profile tree has $\log n$ levels, the complexity for the first pass is $O(\log^2 n)$ time with $O(n\alpha(n)\log n)$ operations.[b]

Similar operations are performed to compute $S_{lson(d)}$ and $S_{rson(d)}$. Although computing $S_{rson(d)}$ requires knowledge of those intervals where $U_{lson(d)}$ is lower than $U_{rson(d)}$, it can be obtained with virtually no extra cost when we compute U_d. In terms of its complexity, a bound on the number of segments involved in the computation is provided by the following result, that is valid in the general case, and has been proved in [10]:

Lemma 1 *Any vertex of V_d is a vertex of $V_{d'}$ for some leaf d' in the subtree rooted at d.*

Since S_d is the lower boundary of V_d, this is also true for S_d. A vertex of S_d in a leaf node d is either a vertex of the final visible region, or an endpoint of S_d in the interval defined by the edge e_d. In the latter case, the corresponding endpoint of e_d is a visible vertex of T. Each visible vertex of T will create at most two such copies in the leaf level. Therefore, the total number of vertices in $\bigcup S_d$ over all leaf nodes d is of $O(k)$. By Lemma 1, the number of vertices in all S_d at any level of the tree is also $O(k)$. Since k is at most n^2, $O(\log k) = O(\log n)$. The complexity of the second pass, and also the entire algorithm, is $O(\log^2 n)$ time with $O((k + n\alpha(n))\log n)$ operations in a CREW PRAM model. Since the total operation count is the same as the sequential algorithm, this parallelization is processor-efficient.

According to the previous analysis, the reallocation of processors takes more time than the actual computation at each level. So it might be desirable to apply a *lazy update* technique to balance the two costs over the entire tree. That is, we reallocate the processors once every r levels, for some r, instead of at each level. The reallocation time for one pass over the tree will be

$$O(\frac{\log^2 n}{r}) \tag{3}$$

By delaying the reallocation, one processor may have to take care of two intervals at the next level, so the computation will be slowed down by a factor of two for each delay. The order of the computation time for r levels without reallocation is

---

[b] Previous works such as [4,8] achieved better time complexities for the upper envelope computation, but they all require stronger or non-conventional PRAM models.

$$\log\log n + 2\log\log n + 2^2\log\log n + \ldots + 2^{r-1}\log\log n = (2^r - 1)\log\log n \quad (4)$$

and the total time for one pass is

$$O(\frac{\log n}{r} \cdot (2^r - 1)\log\log n) \quad (5)$$

Equating the two costs of (3) and (5), we obtain

$$(2^r - 1)\log\log n = c\log n \quad (6)$$

for some constant c. Therefore,

$$r = \log(c\log n + \log\log n) - \log\log\log n \quad (7)$$

and the time complexity is

$$O(\frac{\log^2 n}{\log\log n}) \quad (8)$$

Although the time complexity is improved, the total number of operations will increase to

$$O((k + n\alpha(n))\frac{\log^2 n}{\log\log n}) \quad (9)$$

and thus is not processor-efficient.

References

1. M. J. Atallah, R. Cole, and M. T. Goodrich, "Cascading divide-and-conquer: A technique for designing parallel algorithms", *SIAM J. Comput.*, **18** (1989) 499–532.
2. M.J. Atallah, S.E. Hambrusch, and L.E. TeWinkel, "Topological numbering of features on a mesh", *Algorithmica*, **6** (1991) 762–769.
3. M.J. Atallah, "Parallel techniques for computational geometry", *Proc. IEEE*, **80** (1992) 1435–1448.
4. S. Chandran, *Merging in parallel computational geometry*, PhD thesis (University of Maryland, College Park, 1989), also published as *Technical Report* CAR-TR-468 (Center for Automation Research, College Park, 1989).
5. R. Cole, "Parallel merge sort", *SIAM J. Comput.*, **17**(4) (1988) 770–785.
6. R. Cole and M. Sharir, "Visibility problems for polyhedral terrains", *Journal of Symbolic Computation*, **7** (1989) 11–30.
7. H. Edelsbrunner, "An acyclicity theorem for cell complexes in d-dimensions", *Combinatorica*, **10** (1990) 251-260.
8. M. T. Goodrich, "Using approximation algorithms to design parallel algorithms that may ignore processor allocation", *Proc. 32nd Annual Symposium on Foundations of Computer Science*, 1991, pp. 711–722.
9. J. JáJá, *An Introduction to Parallel Algorithms* (Addison-Wesley, Reading, MA, 1992).

10. M.J. Katz, M.H. Overmars, and M. Sharir, "Efficient hidden surface removal for objects with small union size", *Proc. 7th ACM Symposium on Computational Geometry*, 1991, pp. 31–40.

11. C. Kruskal, "Searching, merging, and sorting in parallel computation" *IEEE Transactions on Computers*, **32** (1983) 942–946.

12. G. Nagy, "Terrain visibility", *Computers & Graphics*, to appear.

13. J. Reif and S. Sen, "An efficient output-sensitive hidden-surface removal algorithm and its parallelization", *Proc. 4th ACM Symposium on Computational Geometry*, 1988, pp. 193–200.

14. R.E. Tarjan and U. Vishkin, "Finding biconnected components and computing tree functions in logarithmic parallel time", *SIAM J. Computing*, **14** (1985) 862–874.