# Concurrent Computation of Attribute Filters on Shared Memory Parallel Machines

Michael H.F. Wilkinson, *Senior Member*, *IEEE*, Hui Gao, Wim H. Hesselink, Jan-Eppo Jonker, and Arnold Meijster

**Abstract**—Morphological attribute filters have not previously been parallelized mainly because they are both global and nonseparable. We propose a parallel algorithm that achieves efficient parallelism for a large class of attribute filters, including attribute openings, closings, thinnings, and thickenings, based on Salembier's Max-Trees and Min-trees. The image or volume is first partitioned in multiple slices. We then compute the Max-trees of each slice using any sequential Max-Tree algorithm. Subsequently, the Max-trees of the slices can be merged to obtain the Max-tree of the image. A C-implementation yielded good speed-ups on both a 16-processor MIPS 14000 parallel machine and a dual-core Opteron-based machine. It is shown that the speed-up of the parallel algorithm is a direct measure of the gain with respect to the sequential algorithm used. Furthermore, the concurrent algorithm shows a speed gain of up to 72 percent on a single-core processor due to reduced cache thrashing.

**Index Terms**—Attribute filters, connected filters, mathematical morphology, parallel computing, algorithms.

## 1 Introduction

Parallel processing has often been applied to image processing for a number of reasons. First of all, the large data bulk often involved in image processing has meant that the high performance obtained by parallel systems was required. Second, many image-processing problems are readily parallelized [1], usually using the data-parallel approach. Many low-level image-processing routines have a high degree of locality, allowing different segments of the image to be treated independently by different processors. Other global operators such as the Fourier transform are separable, allowing the parallelization by treating the image rows separately in a horizontal pass, whereas the columns are treated separately in the vertical pass. A similar approach has been used for the euclidean distance transform [2], [3].

In mathematical morphology, which is the field this paper deals with, parallel execution has also been used frequently, and the parallelization strategies are similar. However, one class of morphological operator that has, to date, not been parallelized efficiently is that of connected filters. Connected filters are a shape preserving class of image operator [4], [5], [6]. Apart from the older openings by reconstruction [7], [8], this class contains area openings and closings [9], [10] that preserve bright and dark image details of a given minimum area. The class was later extended to attribute filters [5], [11], which are a generalization including both former types [4] and preserve and retain structures in an image based on a wide range of criteria. A subset of these filters, called *shape filters* [12], [13], has been used for extraction of vessels in 3D angiograms [14]. An example is shown in Fig. 1.

As will be shown, the difficulty in parallelizing these filters lies in the fact that they are global but nonseparable; so, they do not fit into any standard parallelization strategy employed in image analysis. In this paper, we present a new concurrent algorithm for a large subclass of these operators, i.e., extensive and antiextensive connected filters. The approach combines the Max-Tree and Min-Tree data structures proposed in [11] with the approach from that in [15] based on Tarjan's union-find algorithm [16]. A variant of the Max-Tree, called component tree [17], can be constructed by postprocessing. A different algorithm for the component tree using union find has been proposed by Najman and Couprie [18], but this is a sequential rather than parallel algorithm. In this paper, the solution is posed in graph-theoretical terms, which may allow its application in other contexts that use undirected graphs with a real-valued function defined on the vertices. Indeed, the join trees used to compute contour trees in computational geometry [19], [20], [21], [22], for which parallel algorithms have been developed [23], are very closely related to Max-Trees, especially when they are augmented to include attribute information in each node [23]. A key difference between the concurrent Max-Tree algorithm presented here and the parallel contour tree algorithm is that the latter assumes that all nodes in the graph have a unique (gray) value. By contrast, Max-Trees were designed to administrate flat zones, i.e., connected sets of vertices with the same gray value. Contour trees are equivalent to *level-line trees* [24], which can be constructed from a Max-tree and a Min-Tree of the same image.

- *M.H.F. Wilkinson, W.H. Hesselink, and J.-E. Jonker are with the Institute for Mathematics and Computing Science, University of Groningen, PO Box 407, 9700 AK Groningen, The Netherlands. E-mail: {m.h.f.wilkinson, w.h.hesselink, j.e.jonker}@rug.nl.*
- *H. Gao is with the School of Computer Science and Engineering, University of Electronic Science and Technology of China, No. 4, Section 2, North Jianshe Rd., Chengdu 610054, P.R. China. E-mail: huigao@uestc.edu.cn.*
- *A. Meijster is with the Centre for High Performance Computing and Visualisation, University of Groningen, PO Box 11044, 9700 CA Groningen, The Netherlands. E-mail: a.meijster@rug.nl.*

*Manuscript received 21 Dec. 2006; revised 24 Aug. 2007; accepted 29 Oct. 2007; published online 4 Dec. 2007.*
*Recommended for acceptance by J. Hu.*
*For information on obtaining reprints of this article, please send e-mail to: tpami@computer.org, and reference IEEECS Log Number TPAMI-0895-1206.*
*Digital Object Identifier no. 10.1109/TPAMI.2007.70836.*

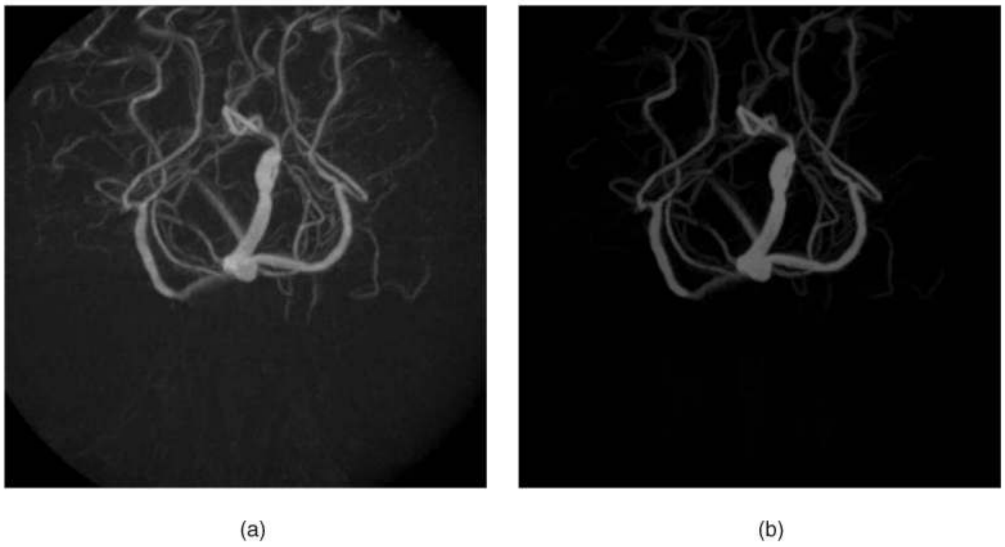
Fig. 1. (a) Maximum intensity projection of $512^3$ CT angiogram, and (b) the same volume enhanced by attribute thinning selective for elongated features. The latter image shows distinct suppression of the background while retaining the vessel structure. Volume data from http://www.volvis.org/, courtesy of Michael Meißner, Viatronix Inc., USA.

Apart from the use in image and volume filtering, computing the Min-tree is used as the initial stage of the computation of the topological watershed [25], [26], which is used for image and volume segmentation. Parallel computation of this initial phase is essential for the development of a parallel algorithm for the topological watershed.

In what follows, we first give a more formal description of connected filters in Section 2. In Section 2.2, we focus on binary attribute filters (including reconstruction operators), and in Section 2.3, we introduce the Min-Tree and Max-Tree data structures that allow efficient sequential implementation of these operators [11]. These data structures allow separation of the filtering process into a building phase, in which the trees are constructed, and a filtering phase, in which it is decided which structures to retain in the image. We then discuss potential parallelization strategies and introduce the core idea of our algorithm in Section 2.4. The bulk of the paper is devoted to formally describing the algorithm and proving its correctness. To do this, we cast the problem in graph theoretic terms in Section 3 and describe how Max-Trees can be constructed sequentially using a tree structure similar to that used in Tarjan's union-find algorithm in Section 4. We then show how the filtering phase of the algorithm is readily parallelized in Section 5.

In Section 6, we describe the crucial part of our algorithm in which multiple Max-Trees are merged. Section 6.1 describes an algorithm that can merge Max-Trees of arbitrary image sections into a single Max-Tree of the image. The algorithm also merges the attributes of the nodes of the partial trees. In practical applications, optimal performance can be expected by using the sequential algorithm for slices of the image that are distributed over the processors, and then merging the results on these slices with the alternative algorithm. The complete algorithm, which merges multiple slices, is described in Section 6.2. In Section 7, we give a complexity analysis of the algorithm. In Section 8, the performance of the parallel algorithm is tested on a 16-processor shared memory computer and an AMD dual-core Opteron-based system. Finally, in Section 9, conclusions are presented.

## 2 Connected Operators

### 2.1 Preliminaries

Though connected filters can easily be described in terms of continuous images, for the sake of simplicity, we only consider discrete images or volumes, which we consider as undirected graphs. In these graphs, the vertices are the pixels (or voxels), and the edges define the neighbor relationships. An undirected graph is modeled as a pair $(V, E)$, where $V$ is the finite set of vertices (the image domain) and $E$ is the set of edges, which is a symmetric subset of $V \times V$.

A graph $(V, E)$ is said to be *connected* if, for any $x$, $y \in V$, there exists a path $(x_0, x_1, \ldots, x_n)$ for which every $(x_i, x_{i+1}) \in E$ and every $x_i \in V$, and with $x_0 = x$ and $x_n = y$. In the following, the image domain is connected.

A binary image $X$ is a subset of $V$ that induces a subgraph $(X, E_X)$ of $(V, E)$, such that $E_X = (X \times X) \cap E$. Set $X$ is said to be connected if $(X, E_X)$ is connected. A connected component $C$ of $X$ is a connected subset of $X$ of maximal extent. This means that there exist no $x \in X \setminus C$ such that $C \cup \{x\}$ is connected. A more detailed discussion on connectivity in this context can be found in [27] and [28].

A binary image filter $\gamma$ is extensive if $X \subseteq \gamma(X)$ for all $X$, and antiextensive if $\gamma(X) \subseteq X$. It is idempotent if $\gamma(\gamma(X)) = \gamma(X)$ and increasing if, for any images $X$ and $Y$, $X \subseteq Y \Rightarrow \gamma(X) \subseteq \gamma(Y)$.

A gray-level image is defined using a function $f : V \rightarrow \mathbb{R}$, which assigns a gray level to each vertex. For $h \in \mathbb{R}$, the $h$-threshold set is defined as the

$$V_h(f) = \{x \in V \mid f(x) \geq h\}. \tag{1}$$

For convenience, we will often use the notation $V_h$ for $V_h(f)$ and denote the subgraph induced by it as $(V_h, E_h)$.

A *peak component* or *dome* $D_h^i$ of image $f$, with $k$ from some index set, is a connected region in which $f(x) \geq h$ for all $x \in D_h^i$, and all neighbors of $D_h^i$ have a gray level smaller than $h$. Equivalently, peak component $D_h^i$ can also

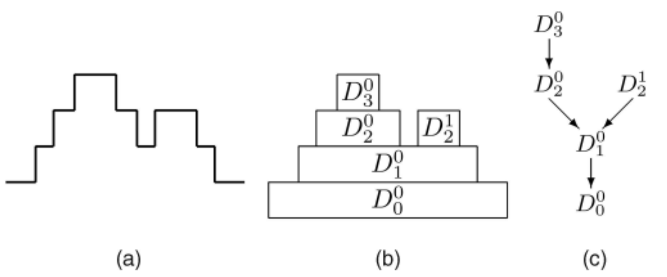

Fig. 2. (a) A 1D signal $f$, (b) the corresponding peak components, and (c) the Max-Tree.

be defined as the $i$th connected component of the threshold set $V_h(f)$.

## 2.2 Binary Attribute Filters

In this section, we discuss the theory of attribute operators. More detail can be found in [5], [11], and [15]. The binary *connected* or *connectivity* opening $\Gamma_x(X)$ of $X$ at point $x \in V$ yields the connected component of $X$ containing $x$ if $x \in X$ and $\emptyset$ otherwise. We use *trivial* attribute filters $\Gamma_T$ with criterion $T : \mathcal{P}(V) \rightarrow \{\text{false}, \text{true}\}$ to determine which components are preserved. If $T(C)$ is true, $\Gamma_T(C) = C$; otherwise, $\Gamma_T(C) = \emptyset$. If $T$ is increasing, i.e., $C \subseteq D \wedge T(C) \Rightarrow T(D)$, $\Gamma_T$ is a trivial opening; otherwise, it is a trivial thinning [5]. Usually, $T$ is defined by using some attribute of the connected set $C$, such as its area, and comparing it to a threshold value $\lambda$. Thus,

$$T(C) = (\mathcal{A}(C) \geq \lambda), \tag{2}$$

with $\mathcal{A} : \mathcal{P}(V) \rightarrow \mathbb{R}$ some attribute function. We call $\mathcal{A}$ increasing if $C \subseteq D \Rightarrow \mathcal{A}(C) \leq \mathcal{A}(D)$. It is easy to see that, if $\mathcal{A}$ is increasing, so is $T$, if it has the form of (2) [5].

**Definition 1.** *The antiextensive binary attribute filter $\Gamma^T$ of set $X$, with criterion $T$, is given by*

$$\Gamma^T(X) = \bigcup_{x \in X} \Gamma_T(\Gamma_x(X)). \tag{3}$$

*If $T$ is* increasing, *$\Gamma^T$ is a binary attribute* opening; *if not, it is a binary attribute* thinning.

By applying the trivial opening or thinning to each of the connected components of $X$, we obtain an attribute opening or thinning [5].

## 2.3 Gray-Scale Attribute Filters

A generalization of attribute filters to gray scale can be made by using thresholded images $V_h(f)$, as in (1). The gray-scale image may be considered as a stack of such threshold sets, each consisting of, as $h$ increases, progressively smaller peak components $D_h^k$. An attribute $\mathcal{A}(D_h^k)$ is assigned to each of these sets. Fig. 2 also shows how the peak components can be combined into a tree structure, called a *Max-Tree* [11]. Put simply, for any two domes $D_h^k$ and $D_{h'}^m$ with $h' < h$, we have either $D_h^k \subseteq D_{h'}^m$ or $D_h^k \cap D_{h'}^m = \emptyset$. This inclusion relation determines the structure of the Max-Tree. The Max-Tree contains both the hierarchy of connected components in the data set, and the attributes for each component to use as a filter criterion. For reasons of memory efficiency, only peak components $D_h^k$ that contain at least one pixel $x$ for which $f(x) = h$ are stored in the tree.

In a Max-Tree, each node, besides the current gray level and data to compute each node's attributes, has a pointer to its parent, and the nodes corresponding to the components with the highest intensity are the leaves (see Fig. 2). To perform extensive filters, we use a Min-Tree, in which all inequalities are reversed and the leaves correspond to the minima. The building of this tree structure is called the *construction phase*, whereas its use for filtering is called the *filtering phase*.

We first consider attribute openings, for which criterion $T$ is increasing [5].

**Definition 2.** *The gray-scale attribute opening $\gamma^T$ of image $f$ with increasing criterion $T$ is the gray-scale image $\gamma^T(f)$ given by*

$$(\gamma^T(f))(x) = \mathbf{max}\{h \mid x \in \Gamma^T(V_h(f))\}. \tag{4}$$

Gray-scale attribute closings can easily be defined by a duality relationship with the gray-scale attribute openings [5], [11] and implemented using Min-Trees. Note that, because $\Gamma^T(V_h(f))$ is insensitive to the gray-level distribution *within* each connected component, the above definition rules out the possibility that $T$ depends on that distribution. Extensions to include such criteria (e.g., using entropy) are discussed in [11] and can be implemented using the Max-tree.

If $\mathcal{A}$ is an increasing attribute function, $\mathcal{A}(D_h^k)$ increases along a root path from leaf to root. Filtering using criterion $T$ as in (2) reduces to descending the Max-Tree from each leaf, until $\mathcal{A}(D_h^k) \geq \lambda$, and removing all nodes in the traversed path, including any children, except $D_h^k$ (which meets $T$). Indeed, the Max-Tree is not necessary for computation of these filters [15].

However, if $T$ is nonincreasing, as is shown in Fig. 2, nodes that meet and do not meet $T$ may alternate in any root path. Salembier et al. [11] describe four different rules for the algorithm to filter the tree: the *Min*, the *Max*, the *Viterbi*, and the *Direct* decision. This latter method reduces to direct implementation of (4) for nonincreasing $T$. In addition, Wilkinson and Urbach [12] introduced another strategy, called the *Subtractive* decision, which is used in most of our work [13], [14]. For a more thorough discussion, see the previous references and also [17], which describes filtering methods that rely on the trace of the attribute value from current node up to the root. In all cases, it is the construction phase that is most time consuming by far.

Fig. 2 shows the peak components of a 1D discrete signal, their attribute values, and the corresponding Max-Tree. Apart from filtering, the Max-tree data structure can also be used to compute univariate and multivariate pattern spectra efficiently [13].

## 2.4 Parallelization Strategies

The simplest parallel algorithm for gray-scale area openings is based on threshold decomposition [29]. By thresholding the image at all gray levels, performing the binary connected operator on each thresholded image, and combining the results into a final gray-scale result, area openings, and, indeed, many other attribute filters can be computed. The algorithm is trivial to parallelize, because each of the threshold images can be treated separately for the binary attribute filter stage, eliminating the need for a true parallel connected filter, and the recombination phase can be done pixel-wise. However, on the average, each processor will have to perform $G/N_p$ binary attribute operations, with $G$ being the number of gray levels and $N_p$ being the number of processors. Because gray-scale attribute openings or closings can be performed almost as fast as a binary attribute opening or closing using the union-find approach proposed in [15], the computing time would be increased by a factor of almost $G/N_p$ compared to a gray-scale operator on a single processor, disregarding the recombination and thresholding stages. Only if $N_p \gg G$ would any significant speed increase be obtained, which is unlikely on shared-memory computers, where $N_p \leq 64$, typically, whereas $G = 256$ or 4,096 in our case.

The more restricted class of filters by reconstruction can be implemented by iterating a conditional dilation or erosion until stability [8]. The conditional dilation or erosion itself can be parallelized efficiently. However, this algorithm can be shown to have a complexity $O(N^2)$, where $N = \#V$, and is far slower than the queue-based approach in [8]. Again, no real performance gains with respect to the fastest sequential algorithm should be expected from this approach.

In [15], some suggestions were made for parallelization of attribute *openings* by the union-find approach. By adding semaphores to each pixel in the image, the algorithm could in principle be parallelized. We did not pursue this avenue of research for two reasons: 1) the overhead caused by $O(N)$ semaphores is large, and 2) the algorithm is unsuitable for attribute thinnings.

The queue-based algorithm in [11] does not lend itself directly to concurrent implementation. We propose, however, to partition the image into $N_p$ connected disjoint regions, the union of which is the entire image domain. We assign each region to one of the $N_p$ processors. We can, of course, build a Max-Tree of each region. The key problem now becomes merging the Max-Trees of all the regions into a single Max-Tree of the image. This requires 1) merging the peak components $D_h^i$, 2) updating the parent relationships, and 3) merging the attributes of the peak components. Once the tree has been built, filtering is easily parallelized.

In Salembier et al.'s approach, the first step cannot be done efficiently because the pixels of each node of the tree are given arbitrary numbers as labels. Merging two regions therefore requires relabeling all of the pixels of one of the two regions. This problem can be addressed easily by adopting the tree structures used in Tarjan's union find algorithm [16], which has been used for concurrent computation of connected component labeling [30]. In this approach, each connected component is represented by a rooted tree and merging two disjoint sets is performed by letting the parent pointer of the root of one of the trees point to the root of the other. This allows near constant time merging of the peak components, as was used in [15], [18], and [31].

We will now cast the problem in a graph theoretic formulation and describe the concurrent algorithm based on this general idea. The reason for this formalism is simple: We need it to prove our algorithm correct, which is not trivial.

## 3 Construction of a Max-Tree

In this section, our aim is to compute all the connected components of all threshold sets $V_h$, as defined in (1), for all levels $h$ in a single computation. In order to do so, we regard edge relation $E_h = (V_h \times V_h) \cap E$ as a relation on $V$ rather than on $V_h$. If we do this, its reflexive transitive closure $E_h^* = (E_h)^*$ is an equivalence relation on $V$. For every $x \notin V_h$, the singleton set $\{x\}$ is an equivalence class of $E_h^*$. The other equivalence classes of $E_h^*$ are the connected components of $V_h$. Therefore, now, the aim is to compute the equivalence classes of all relations $E_h^*$.

For any binary relation $R$ on $V$, we define the $h$-restriction $R_h = R \cap (V_h \times V_h)$. We define $R^\sharp$ to be the symmetric reflexive transitive closure of $R$, i.e., the least equivalence relation that contains $R$. We write $R_h^\sharp$ to denote relation $(R_h)^\sharp$, the symmetric reflexive transitive closure of $R_h$ in $V$.

Inspired by Tarjan's union-find algorithm [16], which was used in a similar way in [30], [32], we represent the equivalence classes of the relations $E_h^*$ by a forest structure induced by pointers to parent nodes (suggested in [33]). These pointers are collected in an array `par` that can be regarded as a (modifiable) function $\texttt{par} : V \to V \cup \{\perp\}$, where $\perp$ stands for a `null` pointer. We define $\texttt{par}^n[x]$ by repeated application of `par`. A vertex $x \in V$ is called a *root* of `par` if and only if (iff) $\texttt{par}[x] = \perp$. Array `par` is called *acyclic* iff $\perp$ is the universal ancestor in the sense that, for every vertex $x$, there exists a number $n$ such that $\texttt{par}^n[x] = \perp$. We define $f(\perp) = -\infty$, since we want to exclude $\perp$ from all threshold sets $V_h$. In all, we represent the Max-Tree structure with this `par` array, the original image (needed to store the original gray level, as in [11]), and pointers to auxiliary data needed to administrate the attributes. The size of these auxiliary data depends very much on the attribute used. The only other data structure needed is the `filt` array containing the output image. The `par` array is the key structure encoding the structure of the Max-Tree itself.

### 3.1 Postulates for Parent Pointers

Let $P = \{(x, \texttt{par}[x]) | x \in V\}$ be the binary relation corresponding to array `par`. We say that `par` is a *Max-Tree* of relation $E$ iff it satisfies the postulates

(S0) $E_h^* = P_h^\sharp$ for all levels $h$,
(S1) $f(x) \geq f(\texttt{par}[x])$ for all vertices $x$,
(S2) `par` is acyclic.

An example is shown in Fig. 3.

Let us assume for the moment that (S0), (S1), and (S2) hold. These postulates enable us to determine the equivalence classes of $E_h^*$ in an efficient way, as follows: For any vertex $x$, we define the "oldest ancestor" down to $h$ by means of the recursive function anc given by

$$anc(x, h) = \begin{cases} x & \text{if } h > f(\texttt{par}[x]), \\ anc(\texttt{par}[x], h) & \text{otherwise.} \end{cases} \tag{5}$$

The recursion terminates since `par` is acyclic by (S2). It follows from (S1) that

$$(x, anc(x, h)) \in P_h^\sharp \quad \text{for all } x \in V,\ h \in \mathbb{R}. \tag{6}$$

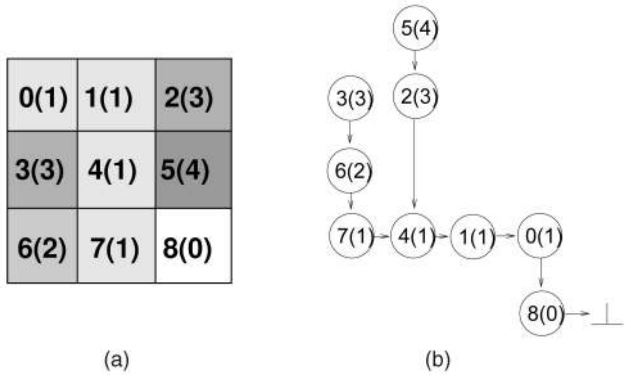

Fig. 3. (a) An image and (b) a possible par-tree structure assuming 4-connectivity. Nodes are numbered $p(q)$, where $p$ is the node number, and $q = f(p)$.

Function *anc* characterizes relation $P_h^\sharp$ by means of

$$(x, y) \in P_h^\sharp \equiv anc(x, h) = anc(y, h) \quad \text{for all } x,\ y \in V,\ h \in \mathbb{R}. \tag{7}$$

This is proved as follows: For fixed $h$, let $A_h$ be the binary relation on $V$ defined by the right-hand side of (7). We have $A_h \subseteq P_h^\sharp$ because of

$$\begin{aligned}
&(x, y) \in A_h \\
\Rightarrow&\{\text{definition of } A_h \text{ and (6)}\} \\
&(x, anc(x, h)) \in P_h^\sharp \wedge anc(x, h) = anc(y, h) \\
&\quad \wedge (y, anc(y, h)) \in P_h^\sharp \\
\Rightarrow&\{P_h^\sharp \text{ is an equivalence relation}\} \\
&(x, y) \in P_h^\sharp.
\end{aligned}$$

On the other hand, we have $P_h \subseteq A_h$ because of the easy property:

$$(x, y) \in P_h \Rightarrow anc(x, h) = anc(y, h) \quad \text{for all } x,\ y \in V,\ h \in \mathbb{R}.$$

Since $A_h$ is an equivalence relation, this implies $P_h^\sharp \subseteq A_h$, and hence, $P_h^\sharp = A_h$. This concludes the proof of (7).

Since $E_h^* = P_h^\sharp$ by (S0), function *anc* characterizes $E_h^*$ in the same way. Indeed, $anc(x, h)$ can be chosen as a representative of the equivalence class of $x$ for relation $E_h^*$.

We define a vertex $x$ to be a *level root* iff $f(x) > f(\mathtt{par}[x])$. Clearly, every root of `par` is a level root because of the convention $f(\perp) = -\infty$. It is easy to see that $f(x) \geq h$ implies that $x' = anc(x, h)$ satisfies $f(x') \geq h > f(\mathtt{par}[x'])$ and is therefore a level root. We write *LeRo* for the set of level roots

$$LeRo = \{x \in V \mid f(x) > f(\mathtt{par}[x])\}.$$

## 3.2 Sets of Descendants and Path Compression

The final value of `par` can be used as follows: According to (S0) and (7), two elements $x$ and $y$ of $V_h$ belong to the same connected component of threshold set $V_h$ iff $anc(x, h) = anc(y, h)$. It follows that each connected component of $V_h$ is characterized by a unique common ancestor, say, $x$ with $f(x) \leq h < f(\mathtt{par}[x])$. The component of $x$ in $V_h$ is the set of descendants:

$$D(x) = \{x\} \cup \{p \in V \mid (\exists\, n \in \mathbb{N} \setminus \{0\} :: \mathtt{par}^n[p] = x)\}.$$

Informally, $D(x)$ is the set of the nodes of the subtree rooted on $x$. The sets $D(x)$ and $D(y)$ of unrelated vertices $x$ and $y$ are disjoint. Indeed, it is not difficult to verify the equivalence:

$$D(x) \cap D(y) \neq \emptyset \;\equiv\; x \in D(y) \;\vee\; y \in D(x).$$

Determination of the sets $D(x)$ can be made more efficient by shortening the parent paths of nodes whenever ancestors are determined. This is the classical method of path compression in Tarjan's union-find algorithm [16]. In comparison with the classical case, we have to be careful that the parent path of any node must not lose the level root at the current level, which is defined by

$$levroot(x) = anc(x, f(x)).$$

Path compression can be implemented by replacing function *levroot* by the following recursive procedure with side effect:

```
procedure $levroot(x)$ returns $V$ =
  if $f(x) \neq f(\mathtt{par}[x])$ then return $x$ end;
  $\mathtt{par}[x] := levroot(\mathtt{par}[x])$;
  return $\mathtt{par}[x]$;
end.
```

Path compression is incorporated in function *anc* by means of *levroot* via

```
procedure $anc(x, s)$ returns $V$ =
  while $f(\mathtt{par}[x]) \geq s$ do $x := levroot(\mathtt{par}[x])$
    end;
  return $x$;
end.
```

One may also consider using the union-by-rank criterion in [16] to resolve the nondeterministic choice between $x$ and $y$ when $f(x) = f(y)$. We did not investigate this possibility since it would at least complicate the code and require additional memory proportional to the size of the graph for storing ranks.

## 3.3 Accumulation of Attributes

Apart from setting the parent pointers correctly, we wish to compute some kind of attribute for each of the components of the threshold sets $V_h$, for all levels $h$. The easiest special case is the attribute 1 for all vertices, in which case, accumulation of attributes yields the number of elements of the components, equivalent to area in 2D and volume in 3D. More complicated attributes are generally computed by computing some auxiliary data set per Max-Tree node [11]. For the area of the smallest bounding (axis-aligned) rectangle, suggested by Breen and Jones [5], this data set would contain the minimum and maximum $x$ and $y$ coordinates of pixels in each peak component. The actual attributes are computed from these auxiliary data sets during the filtering phase.

We treat a general situation where the attributes of nodes are defined in terms of their components by means of a binary operator $\widehat{+}$ that must be commutative and associative, and to have a neutral element $\widehat{0}$. Such an algebraic structure is called a monoid and the set of values in the monoid is denoted $M$. The properties of $\widehat{+}$ enable us to define an associated "summation" operator $\widehat{\sum}$ for families of attributes. Therefore, for a finite set $I$ and a function $g : I \rightarrow M$, the summation yields the value $\widehat{\sum}_{i \in I} g(i)$, defined recursively by

$$\widehat{\sum_{i \in \emptyset}} g(i) = 0,$$
$$\widehat{\sum_{i \in I}} g(i) = g(j) \widehat{+} \widehat{\sum_{i \in I \setminus \{j\}}} g(i)) \quad \text{for every } j \in I.$$

Now, let $\mathcal{A} : V \rightarrow M$ be a given function that assigns attributes to vertices. We aim at the computation of $\widehat{\sum}_{v \in C} \mathcal{A}(v)$ for all connected components $C$ of the threshold sets $V_h$. For an arbitrary subset $C$ of $V$, we introduce the notation $\mathcal{A}(C) = \widehat{\sum}_{v \in C} \mathcal{A}(v)$ (in other words, $\mathcal{A}$ returns the attribute value of sets of vertices, as well as single vertices). Our computations will be based on the basic rules $\sum \emptyset = \widehat{0}$ and $\mathcal{A}(\{v\}) = \mathcal{A}(v)$ and the union rule that $\mathcal{A}(C \cup B) = \mathcal{A}(C) \widehat{+} \mathcal{A}(B)$ whenever $C$ and $B$ are disjoint.

We introduce an array accat to hold accumulated attributes. Since the components of threshold sets are obtained as sets $D(p)$ for level roots $p$, we aim at the postcondition

(Q0) $(\forall\, p \in LeRo :: \texttt{accat}[p] = \mathcal{A}(D(p)).$

Thus, the algorithm to compute the Max-Tree must reach a postcondition in which both the postulates for the parent pointers and Q0 hold.

## 4 Including Union-Find in the Max-Tree Algorithm

We will now adapt Salembier et al.'s recursive algorithm for Max-Tree construction, including the accumulation of attributes, to include the tree data structures used in Tarjan's union-find method for representation of the Max-Tree nodes. For simplicity, we assume that the graph is connected and nonempty. We present a version, shown in Algorithm 1, of the algorithm by Salembier et al. [11] that constructs a Max-Tree, i.e., an array par that satisfies the postulates (S0), (S1), and (S2). For the sake of the analysis, we assume that initially $\texttt{par}[x] = \perp$ holds for all nodes $x$.

Let *Level* be a set of values such that $f(x) \in Level$ for all $x \in V \cup \{\perp\}$. Let xm be a node where $f$ takes its minimal value. We thus have $f(x) \geq f(\texttt{xm})$ for all $x$.

Algorithm 1 uses the following additional variables:

```
hm: Level;
W: set of Node;
set: array Level of set of Node;
lero: array Level of Node ∪ {⊥};
at: Attribute.
```

Variable hm holds the level for the level set that is currently under investigation. $W$ stands for the set of nodes that have been reached by the algorithm. The sets $\texttt{set}[k]$ hold sets of nodes of level $k$. Variable $\texttt{lero}[k]$ is used to hold the latest level root of level $k$. Initially, all sets are empty, and the elements of lero are $\perp$.

**Algorithm 1**. Salembier et al.'s recursive Max-Tree algorithm using the union-find-derived data structure to represent level components. Procedure *flood* uses *lev* and *at* as reference parameters as in Pascal or Modula-3. Vp denotes the section of the image to work on. For a sequential algorithm, this equals the image domain V.

```
procedure $flood$ (var $lev : Level$; var $at : Attribute$;
$Vp : Section$) =
  $\texttt{accat}[\texttt{lero}[lev]] := at$;
  while $\texttt{set}[lev] \neq \emptyset$ do
    extract some $p$ from $\texttt{set}[lev]$;
    $\texttt{accat}[\texttt{lero}[lev]] := \texttt{accat}[\texttt{lero}[lev]] \widehat{+} \mathcal{A}(p)$;
    for all neighbors $q \in \mathrm{Vp}$ of $p$ do
      if $q \notin W$, then
        var $fq := f(q)$; $atq := 0$;
        $W := W \cup \{q\}$;
        if $\texttt{lero}[fq] = \perp$, then
          $\texttt{lero}[fq] := q$;
        else $\texttt{par}[q] := \texttt{lero}[fq]$ end;
        $\texttt{set}[fq] := \texttt{set}[fq] \cup \{q\}$;
        while $fq > lev$ do flood($fq$, $atq$, $Vp$) end;
        $\texttt{accat}[\texttt{lero}[lev]] := \texttt{accat}[\texttt{lero}[lev]] \widehat{+} atq$;
  end end end;
  determine $m$ maximal with
    $m < lev \wedge (\texttt{lero}[m] \neq \perp \vee m = -\infty)$;
  $\texttt{par}[\texttt{lero}[lev]] := \texttt{lero}[m]$;
  $at := \texttt{accat}[\texttt{lero}[lev]]$;
  $\texttt{lero}[lev] := \perp$;
  $lev := m$;
end;

hm := $f$(xm);
set[hm] := $W$ := {xm};
lero[hm] := xm;
at := $\widehat{0}$;
flood(hm, at, Vp).
```

The algorithm consists of some initializations and one call of a recursive procedure, shown in Algorithm 1. Algorithm 1 differs from Salembier et al.'s original only in that the nodes of the Max-Tree are labeled using a different convention. The analysis in [11] therefore also applies to this algorithm.

## 5 Filtering Phase

When the global Max-Tree has been computed, either sequentially or concurrently as explained in the next section, we can perform filtering very efficiently. We assume that concurrent reading of a memory location is allowed without using synchronization primitives like semaphores. During this stage, the Max-Tree is not modified; therefore, the processes are allowed to read the data structure simultaneously without using semaphores.

Let $K$ be the number of threads (or processors) to be employed. We number them from 0 to $K - 1$. The set of vertices $V = [0, N)$ is split in $K$ consecutive subdomains $V^p = [\mathrm{lwb}(p), \mathrm{lwb}(p+1))$, where $\mathrm{lwb}(p) = p \cdot N \textbf{ div } K$ for $0 \leq p < K$.

We consider the case of direct filtering, i.e., each pixel is lowered in gray level to the level of the ancestor in the Max-Tree with the highest gray level that satisfies the filter criterion. The algorithm is shown in Algorithm 2. We introduce an array *filt* in which we store the filtered data set. We assume that all points $v$ in the output volume will have a gray value *filt* $[v] \geq 0$. For this reason, *filt* $[v]$ is initialized to $-1$ for all $v \in V$, to flag them as unprocessed. The only thing a process $p$ has to do for each point $v$ of its private domain $V^p$ is to follow `par`-pointers until it reaches an ancestor $w$ of $v$, which satisfies the filter criterion. When $w$ is found, *filt* $[v]$ is set to $f(w)$.

**Algorithm 2**. Concurrent implementation of the filtering phase.

```
procedure $directfilter\ (lambda : attribute; Vp : Section) =$
  for all $v \in Vp$ do
    if $filt[v] = -1$ then
      $w := v;$
      while $par[w] \neq \perp \wedge filt[w] = -1\ \wedge$
        $(f(w) = f(par[w]) \vee accat[w] < lambda)$ do
        $w := par[w];$
      end;
      if $filt[w] \neq -1$ then
        $val := filt[w];$ (* criterion satisfied at level $filt[w]$ *)
      else if $accat[w] \geq lambda$ then
          $val := f(w);$ (* $w$ satisfies criterion *)
        else $val := 0;$ end; (* criterion cannot be satisfied *)
      end;
      (* set $filt$ along $par$-path from $v$ to $w$ *)
      $u := v;$
      while $u \neq w$ do
        if $u \in Vp$ then
          $filt[u] := val;$
        end;
          $u := par[u];$
      end;
      if $w \in Vp$ then
          $filt[w] := val;$
      end;
    end;
  end;
end.
```

## 6 Concurrent Merging of Max-Trees

We will now first discuss how two Max-Trees obtained by Algorithm 1 from adjacent image sections can be merged into a single one, followed by a scheme for the merger on multiple trees efficiently.

### 6.1 Merging Two Max-Trees

We can split the image into $K$ parts as in the filtering phase and first neglect all edges that connect one part to another. These parts with their internal edges can then be distributed over equally many processors. By application of Algorithm 1 concurrently on the parts, we can obtain a data structure that satisfies (Q0) but is restricted to each domain $V^p$. The edges that connect different parts have yet to be processed.

We now develop a sequential algorithm to accommodate the remaining edges. This algorithm mixes the construction of the forest `par` with the accumulation of attributes. In order to do so, we introduce a program variable $F$ to hold a set of pairs of nodes and replace postulate (S0) by the invariant

(J0) $E_h^* = (F \cup P)_h^\sharp$ for all levels $h$

while maintaining (S1) and (S2) as invariants. The idea is that $F$ holds the unprocessed pairs.

We thus assume that the set $F$ and the arrays `par` and `accat` are initialized and satisfy (J0), (S1), (S2), and (Q0). When merging $V^p$ and $V^{p+1}$, this means that every $(x, y) \in E$ such that $x \in V^p$ and $y \in V^{p+1}$ is an element of $F$. For every level $h$, the set $E_h$ is the union of $F_h$ and its converse. It follows that $E_h^* = F_h^\sharp$ for all $h$ and, hence, that (J0) holds.

A general algorithm for merging the results of the sections can be formulated as

```
while $F \neq \emptyset$ do
  extract some pair $(x, y)$ from $F$;
  $connect(x, y);$
end.
```

Removing a pair $(x, y)$ from $F$ violates invariant (J0) but preserves (Q0), (S1), and (S2) since the latter predicates do not refer to $F$. Therefore, the aim of *connect* is to restore the invariant (J0) while preserving (Q0), (S1), and (S2).

Procedure *connect* has to merge the two parent paths of the vertices $x$ and $y$. We do this in a repetition where the vertices $x$ and $y$ serve as local variables. Since we are only interested in the values of `accat`$[p]$ for level roots $p$, we approximate (J0) by means of the invariant

(J1) $x \in LeRo \wedge y \in LeRo\ \wedge$
$(\forall\ h :: E_h^* = (F \cup \{(x, y)\} \cup P)_h^\sharp).$

Preservation of this invariant is made more convenient by introducing the function

$$Par(x) = (\texttt{par}[x] = \perp ?\ \perp :\ levroot(\texttt{par}[x])).$$

When applied to a level root, function *Par* yields the level root of the next higher level in the tree (= lower gray level!). We define the set of ancestors of any node $x$ by

$$Anc(x) = \{p \in V \,|, (\exists\ n \in \mathbb{N} :: Par^n(x) = p)\}.$$

It is useful to notice that $D(x) \subseteq D(p)$ for every vertex $p \in Anc(x)$.

Modification of array `par` in the form `par`$[x] := y$ changes the descendant sets $D(p)$ for the ancestors $p$ of either $x$ or $y$. This clearly threatens predicate (Q0). Writing $\bowtie$ for the symmetrical difference of sets: $C \bowtie B = (C \setminus B) \cup (B \setminus C)$, we weaken (Q0) to the invariant

(J2) $(\forall\ p \in LeRo :: p \notin Anc(x) \bowtie Anc(y) \Rightarrow$
`accat`$[p] = \mathcal{A}(D(p))$.

If $x = y$, predicate (J1) implies (J0) and (J2) implies (Q0), since the pair $(x, x)$ is not needed for the symmetric reflexive transitive closure in (J0) and the symmetric difference of two equal sets is empty. We therefore use $x \neq y$ as a guard of the repetition in *connect*.

Predicate (J2) says nothing about the values `accat`$[p]$ for vertices $p$ in the symmetric difference $Anc(x) \bowtie Anc(y)$, i.e., for the ancestors in the parent paths of $x$ and $y$ as long as they do not converge. Here, we have to break the symmetry, since $\texttt{accat}[p]$ may have accumulated more or less than it deserves. We choose $f(x) \geq f(y)$. We introduce an auxiliary variable $U$ to hold a set of vertices and cor for its "sum" of attributes, with the invariant

$$\begin{aligned}\text{(J3)}\ & f(x) \geq f(y) \wedge U \subseteq D(x) \wedge \texttt{cor} = \mathcal{A}(U)\\ & \wedge (\forall\, p \in Anc(x) \setminus Anc(y)::\\ &\qquad \texttt{accat}[p] = \mathcal{A}(D(p) \setminus U)\\ & \wedge (\forall\, p \in Anc(y) \setminus Anc(x)::\\ &\qquad \texttt{accat}[p] = \mathcal{A}(D(p) \cup U).\end{aligned}$$

One can see that $U$ holds the vertices in $D(x)$ for which the attributes are accumulated in the ancestors of $y$, whereas these attributes should be accumulated in the ancestors of $x$. The variable $U$ is not needed in the algorithm but only in its correctness proof.

Procedure *connect* has parameters $x$ and $y$, which also serve as local variables. The invariants (J1), (J2), and (J3) are initialized by

```
Init:
  cor := $\widehat{0}$; $U := \emptyset$;
  $x := levroot(x)$; $y := levroot(y)$;
  if $f(y) > f(x)$ then $swap(x, y)$ end.
```

In the loop body, we have $x \neq y$ and $f(x) \geq f(y)$, and hence, $x \neq \perp$. We also have $x \in Anc(x)$ and $x \notin Anc(y)$, so that (J3) implies that $\texttt{cor}\widehat{+}\texttt{accat}[x] = \mathcal{A}(U)\widehat{+}\mathcal{A}(D(x) \setminus U) = \mathcal{A}(D(x))$. This suggests replacing $\texttt{accat}[x]$ by the value $\texttt{copa} = \texttt{cor}\widehat{+}\texttt{accat}[x]$ and then replacing $x$ by $\texttt{par}[x]$ or rather $Par(x)$.

Indeed, since $x \neq \perp$, we may consider the vertex $z = Par(x)$, which satisfies $z \in LeRo \cup \{\perp\}$ and $f(x) > f(z)$. Moreover, $Anc(x) = \{x\} \cup Anc(z)$. We aim at replacing the pair $(x, y)$ by $(z, y)$ or $(y, z)$. We consider two overlapping cases.

First, assume $f(z) \geq f(y)$. In this case, all three invariants are preserved by

```
LaggingBehind: $\{f(z) \geq f(y)\}$
  accat[$x$] := cor$\widehat{+}$accat[$x$];
  $x := z$.
```

Next, assume that $f(y) \geq f(z)$ and $y \neq z$. We want to consider the assignment $\texttt{par}[x] := y$. This assignment has the effect that some sets $D(p)$ change. Actually, $D(p)$ can only change if $p \in Anc(y) \bowtie Anc(z)$. For such a vertex $p$, let $Dp$ stand for the set $D(p)$ after the assignment $\texttt{par}[x] := y$. We have that $Dp = D(p) \cup D(x)$ if $p \in Anc(y) \setminus Anc(z)$, and $Dp = D(p) \setminus D(x)$ if $p \in Anc(z) \setminus Anc(y)$. It is this appearance of $D(x)$ that inspired us to formulate (J3).

Write $U' = D(x) \setminus U$. We have $\mathcal{A}(U') = \texttt{accat}[x]$ by (J3). If $p \in Anc(y) \setminus Anc(z)$, then (J3) implies that $\texttt{accat}[p] = \mathcal{A}(D(p) \cup U)$, whereas

$$D(p) \cup U = D(p) \cup (D(x) \setminus U') = Dp \setminus U'.$$

On the other hand, if $p \in Anc(z) \setminus Anc(y)$, then (J3) implies $\texttt{accat}[p] = \mathcal{A}(D(p) \setminus U)$, and since $U \subseteq D(x) \subseteq D(p)$, we also have

$$D(p) \setminus U = (D(p) \setminus D(x)) \cup U' = Dp \cup U'.$$

In this way, one can prove that the assumption implies that

$$\begin{aligned}\text{(J3}'\text{)}\ & f(y) \geq f(z) \wedge \texttt{accat}[x] = \mathcal{A}(U')\\ & \wedge (\forall\, p \in Anc(y) \setminus Anc(z) ::\\ &\qquad \texttt{accat}[p] = \mathcal{A}(Dp \setminus U')\\ & \wedge (\forall\, p \in Anc(z) \setminus Anc(y) ::\\ &\qquad \texttt{accat}[p] = \mathcal{A}(Dp \cup U').\end{aligned}$$

We thus obtain that the invariants (J1), (J2), and (J3) are preserved by

```
Overtaking: $\{f(y) \geq f(z) \wedge y \neq z\}$
  copa := cor$\widehat{+}$accat[$x$];
  cor := accat[$x$];
  accat[$x$] := copa;
  par[$x$] := $y$; $U := D(x) \setminus U$;
  $x := y$; $y := z$.
```

Putting these arguments together and resolving the nondeterminacy in the simplest way, we obtain

```
procedure $connect(x, y)$ =
  Init;
  while $x \neq y$ do
    $z := Par(x)$;
    if $f(z) \geq f(y)$ then LaggingBehind
    else Overtaking end
  end
end connect.
```

It now turns out that the variable $U$ is computationally irrelevant. It is a so-called ghost variable or auxiliary variable, useful for the proof but superfluous in the program. Such variables are well known in proofs of concurrent programs, e.g., see [34], but they are rarely used for sequential programs. Variable $U$ can be eliminated in every implementation.

The convention $f(\perp) = -\infty$ can be implemented by adding a virtual vertex $\perp$ to the image. In Algorithm 3, we have chosen to eliminate this vertex by splitting the loop into two loops, where the second is taken only if the parent paths of $x$ and $y$ only merge at $\perp$. An example is shown in Fig. 4.

**Algorithm 3**. Merging two Max-Trees while ensuring the correct accumulation of attributes.

```
procedure $connect(x, y)$ =
  cor := $\widehat{0}$;
  $x := levroot(x)$; $y := levroot(y)$;
  if $f(y) > f(x)$ then $swap(x, y)$ end
  while $x \neq y \wedge y \neq \perp$ do
    $z := Par(x)$;
    if $z \neq \perp \wedge f(z) \geq f(y)$ then
      accat[$x$] := cor$\widehat{+}$accat[$x$];
      $x := z$;
    else
      copa := cor$\widehat{+}$accat[$x$];
      cor := accat[$x$];
      accat[$x$] := copa;
      par[$x$] := $y$;
      $x := y$; $y := z$.
    end
  end
  if $y = \perp$ then
    while $y \neq \perp$ do
      accat[$x$] := cor$\widehat{+}$accat[$x$];
```

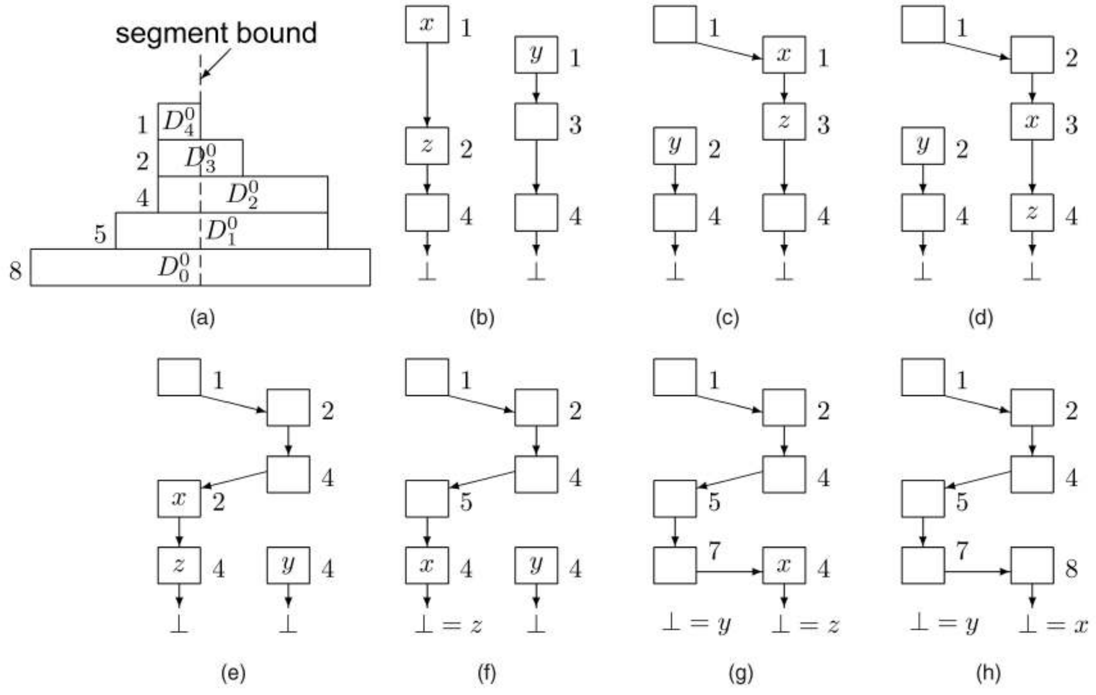

Fig. 4. Merging two simple Max-Trees: (a) Signal. A simple signal split into two domains, showing correct area attributes. (b) Start; cor=0. The two Max-Trees showing partial area attributes, initial settings of $x$, $y$, $z$, and cor. (c) Step 1; cor=1. First, the parent pointer of the initial $x$ is updated, and cor is set to 1. (d) Step 2; cor=1. Next, cor is added to area of top node of right-hand tree. (d)-(g) Stepwise merger, updating cor each time $x$ switches form left to right tree, or vice versa. (e) Step 3; cor=3. (f) Step 4; cor=3. (g) Step 5; cor=4. (h) Final tree. Note that the root node of the original left-hand tree is no longer a level root, so its attribute is ignored in further processing.

```
            $x := Par(x)$;
        **end**
    **end**
**end** *connect*.
```

## 6.2 Concurrent Merging of Multiple Trees

Now that we have developed procedure *connect*, we can construct the Max-Tree of an image by means of a concurrent algorithm, shown in Algorithm 4.

Each thread $p$ first executes Algorithm 1 for its own subdomain $V^p$. After this, the subdomains are merged by means of a binary tree in which thread $p$ accepts all subdomains $V^{p+i}$ with $p+i<K$ and $0 \leq i < 2^a$, where $2^a$ is the largest power of 2 that divides $p$. In particular, odd-numbered threads accept no subdomains, and all subdomains are eventually collected by thread 0. If $K$ is not a power of 2, the binary tree is not complete, but the algorithm stays correct (although one may prefer to use a more balanced tree for reasons of performance).

A thread that needs to accept the domain of its right-hand neighbor has to wait until the neighbor has completed its Max-Tree computation. The final combination is computed by thread 0. Therefore, all other threads must wait for thread 0 before they can resume their computation for the filtering phase. This synchronization is realized by means of two arrays of $K-1$ binary semaphores sa and sb. A binary semaphore s is a shared variable with values in $\{0,1\}$ and two atomic actions $V(\mathtt{s})$ and $P(\mathtt{s})$. Action $V(\mathtt{s})$ increments the value of semaphore s to 1. A process that needs to execute action $P(\mathtt{s})$ waits until $\mathtt{s}=1$ and then decrements s in one atomic action to 0. In our case, all semaphores sa[$p$] and sb[p] have the initial values 0.

**Algorithm 4**. Concurrent construction and filtering of the Max-Trees, thread $p$.

```
**process** $ccaf(p)$
  *find* xm *such that* $f(\mathtt{xm}) \leq f(x)\ \forall x \in V^p$;
  $\mathtt{hm} := f(\mathtt{xm})$;
  $\mathtt{set}[\mathtt{hm}] := W := \{\mathtt{xm}\}$;
  $\mathtt{lero}[\mathtt{hm}] := \mathtt{xm}$;
  $\mathtt{at} := 0$;
  $flood(\mathtt{hm}, \mathtt{at}, V^p)$;
  **var** $i := 1$, $q := p$;
  **while** $p+i<K \wedge q$ **mod** $2 = 0$ **do**
    $P(\mathtt{sa}[p+i])$ (* wait to glue with
                    right-hand neighbor *);
    **for all** edges $(x,y)$ between $Tree(p)$
          and $Tree(p+i)$ **do**
      $connect(x,y)$;
    **end**;
    $i := 2*i$; $q := q/2$;
  **end**;
  **if** $p=0$ **then** (* release the waiting threads *)
    **for** $i := 1$ **to** $K-1$ **do** $V(\mathtt{sb}[i])$ **end**
  **else**
    $V(\mathtt{sa}[p])$ (* signal left-hand neighbor *);
    $P(\mathtt{sb}[p])$ (* wait for thread 0 *)
  **end**;
  $filter(p, lambda)$;
**end** *ccaf*.
```

Here, $Tree(p)$ is the data structure of process $p$, which consists of the arrays par and accat restricted to the subdomain $\bigcup_{j=0}^{i-1} V^{p+j}$, which is currently the responsibility of process $p$.

Array sa of semaphores is used to guarantee that a thread only starts merging with its right-hand neighbor when the neighbor is ready for it. Array sb of semaphores is used to guarantee that the threads only start with their filtering phase after thread 0 has concluded the gluing phase.

In order to prove the absence of deadlock with the semaphores sa, we claim that every semaphore sa$[r]$ with $1 \leq r < K$ is incremented precisely once, namely, by process $r$, whereas only one process $p$ tries to decrement it, namely, $p = r - 2^a$, where $2^a$ is the highest power of 2 that divides $r$. There is no cyclic waiting, since every process only waits for a semaphore that will be released by a process with higher number.

The synchronization could also be established using the pthread primitives of mutexes and condition variables, or with Java's wait and notify primitives.

## 7 Complexity

The computational complexity of the local flooding depends on the algorithm of choice. If we build the trees with the algorithm derived from that by Salembier et al. [11], the worst-case complexity of the building phase is $O(N(C+G)/N_p)$, with $N$ being the number of voxels, $G$ being the number of gray levels, $C$ being the connectivity (4, 8, 6, or 26 in our case), and $N_p$ being the number of processors, as before. As noted in [15], this worst case occurs when the difference in gray levels between parent and child, and the number of leaves in the Max-tree are both maximal. Using a priority queue instead of a simple array to store the nodeatlevel data (equivalent to lero in Algorithm 1) yields a complexity of $O(N(C+\log G)/N_p)$ for the building phase, but is expected to be slower [15]. Alternatively, one could use the algorithm in [35]. This algorithm has complexity $O(N(C+\log G))$, yielding an overall complexity of $O(N(C+\log G)/N_p)$. This will also be tested in the following section. The algorithm by Najman and Couprie [18] is quasilinear, i.e., $O(CN \times \alpha(CN,N)/N_p)$, with $\alpha$ the inverse of the Ackermann function, which increases extremely slowly. A variant is the algorithm by Berger et al. [31], which has slightly higher time complexity, but uses far less memory. Because our main issue is with the merging stage, we did not test the latter two algorithms.

The merging phase complexity is determined by the way the volume is subdivided. The simplest way is to divide the volume into $N_p$ slices, so that each process gets assigned a contiguous block of memory to work in. If neighboring slices have $K$ connecting edges and the Max-trees have a maximal depth $G$, one merging phase requires $O(KG\log N)$ worst case, with $\log N$ as the complexity of a single merge using union find with path compression but without union by rank [16]. This assumes that, for every merge, we need to descend $O(G)$ nodes, which is not the case in practice. Only the first merge need actually descend all the way to the root; each next merge will encounter the condition that $x = y$ long before the root is encountered in most cases. Using $N_p$ parallel slices with equal boundaries, the total merging has a worst case time complexity $O(KG\log N\log P)$. Using oct-trees, however, should reduce this to $O(KG\log N)$. This is because the final merge (on one processor) will have $K$ edges to process, the preceding one $K/2$, the one before that $K/4$ per processor, etc. Unfortunately, the code needs to become much more complicated in that case.

To cast $K$ in terms of $N$, consider the following: Assume 6-connectivity in 3D. If a cubic volume of $N = M^3$ pixels is partitioned into $P$ parallel slices, the sequential preprocessing would therefore require $O((M^3G)/P)$, whereas $K = M^2 = N^{2/3}$. Merging takes less than preprocessing when $\log M \log P \leq M/P$.

In our experience, the Max-tree building phase takes more time than the Max-tree merging phase (typically, only 3-7 percent of CPU time is spent on the merging phase at 16 threads, rising to 12-21 percent for 64 threads). Therefore, efforts to improve the merging phase, e.g., by introducing oct-trees have been postponed.

The filtering phase can be performed linearly in the number of pixels [15] and should have complexity of $O(N/G)$, yielding an overall complexity of $O(G(N/P + K\log N\log P))$ or $O(G(N/P + N^{2/3}\log N\log P))$ assuming cubic volumes.

Memorywise, the algorithm is no more costly than the building algorithm itself. Using the algorithm in [11], we need a hierarchical queue (requires $O(N+G)$ storage), the original image, par array, auxiliary data, and array filt to store the output (all $O(N)$). The original sequential algorithm required the same storage plus a store for the actual tree structure ($O(N)$ worst case).

## 8 Performance Testing

The above algorithm was implemented in C in two variants: one for the volume opening in 3D and one for the more general class of antiextensive attribute filters. Wall-clock runtimes for numbers of threads equal 1, 2, 4, 6, 8, 12, 16, 32, and 64 for each of these algorithms were determined. The elongation measure $\varphi_1$ used in [14] was used in the second algorithm. This measure is a 3D equivalent of the first moment invariant by Hu [36] and is given by

$$\varphi_1(X) = \frac{\mathcal{I}(X)}{(\mathcal{V}(X))^{5/3}} \tag{8}$$

with $\mathcal{V}(X)$ the volume of set $X$ and $\mathcal{I}(X)$ the trace of the moment-of-inertia tensor of $X$ given by

$$\mathcal{I}(X) = \frac{\mathcal{V}(X)}{4} + \sum_{\vec{r}\in X}(\vec{r}-\vec{r}_{cm})^2 \tag{9}$$

with $\vec{r}_{cm}$ the center of mass location of $X$. This attribute cannot be computed incrementally, but it can be computed from an auxiliary data set containing volume, $\sum x$, $\sum y$, $\sum z$, and $\sum x^2+y^2+z^2$, which itself can be computed incrementally. A similar strategy for using auxiliary data sets was used in [5] and [11].

Timings were performed on the 16-processor Silicon Graphics Onyx 3400 shared memory parallel computer of the Centre for High Performance Computing and Visualisation, University of Groningen, and on an AMD dual-core, Opteron-based machine. Each processor of the Onyx is a

TABLE 1
Volume Data Sets Used in Performance Testing

| Large volumes | | | | |
|---|---|---|---|---|
| Name | size | bits | type | source |
| backpack | $512^2 \times 373$ | 8, 12 | backpack CT | Kevin Kreeger, Viatronix Inc., USA. |
| prone | $512^2 \times 463$ | 8, 12 | colon CT | Walter Reed Army Medical Center, USA. |
| supine | $512^2 \times 426$ | 8, 12 | colon CT | Walter Reed Army Medical Center, USA. |
| vertebra | $512^3$ | 8, 12 | CT-angiogram | Michael Meißner, Viatronix Inc., USA. |
| XMastree | $512^2 \times 499$ | 8, 12 | tree | University of Vienna and Vienna University of Technology [37]. |
| **Small volumes** | | | | |
| Name | size | bits | type | source |
| mrt_angio2 | $256 \times 320 \times 128$ | 8, 10 | MR-angiogram | Özlem Gürvit, Institute for Neuroradiology, Frankfurt, Germany. |
| angio | $256^2 \times 124$ | 8, 12 | MR-angiogram | |
| fullhead | $256^2 \times 176$ | 8, 12 | head CT | Visualization Toolkit |
| xmas | $256^2 \times 249$ | 8, 12 | tree | University of Vienna and Vienna University of Technology [37]. |

500-MHz MIPS R14000 processor, and the machine had 20 Gbytes of RAM. The Opteron-based machine has two dual-core Opteron 280 processors at 2.4 GHz, giving a total of four processor cores and 8 Gbytes of memory (4 Gbytes per processor socket). The minimum value of 10 timings was taken to be most indicative of the speed of the algorithm. The timings were done on five publicly available volume data sets that were processed in two versions: 10-bit or 12-bit original resolution and down sampled to 8 bits per voxel. All large volume data sets were rotational b-plane CT scans, and their sources are given in Table 1. The timing results are shown in Fig. 5. Four other rather smaller data sets (approx $256^3$ voxels) from a variety of sources, including magnetic resonance images besides CT-scans, were also tested in two gray-level resolutions each. The timings of these sets follow the same pattern as the large data sets in Table 1 (data not shown).

On the Onyx 3400, wall-clock computing time for the area openings drop from an average of 241 s for a single thread down to 20.1 s at 64 threads for 8-bit data and from 321 s down to 34.8 s for 12-bit data. For attribute thinning using elongation criteria, these times are 326 s and 29.1 s for 8-bit data and 429 s and 47.7 s for 12-bit data, respectively. As expected, the speed-up for 8-bit/voxel volumes is higher than for 12-bit/voxel images due to the increased height and complexity of the trees, both of which influence the performance, though on the Opteron-based machine, the difference is slight (4.59 versus 4.63 maximum speedup).

A striking result, as shown in Fig. 5, is the fact that the speed-up increases as we increase the number of threads beyond the number of processors in our machine (16). For volume openings on the Onyx, the speedup at 16 threads is $7.68 \pm 0.55$ and $6.91 \pm 0.86$ for 8-bit and 12-bit data, respectively. However, at 64 threads, this has risen to $12.0 \pm 1.6$, and $9.2 \pm 1.6$, respectively. This is probably due to the fact that, as the slices become smaller, the flood-filling process needed to build the trees for each slice becomes more cache friendly. This apparently more than compensates the penalty for doing more work merging the trees together (and the overhead of multithreading). This result also means that even more speed-up could be attained using more processors. For elongation filtering, the results are similar, if slightly worse, attaining a speed-up of $11.3 \pm 1.2$ for 8-bit data and $9.0 \pm 1.0$ for 12-bit data at 64 threads.

On the Opteron-based machine, the situation is slightly different, showing a slightly higher than linear speed-up on 2 and 4 threads for the largest data sets (2.17 and 4.11, respectively, for 12-bit data for elongation filtering and 2.12 and 4.1 for volume openings). This is probably caused by the fact that more than 4 Gbytes of memory is needed in this case, which means that, when using a single thread, data from the other processor's memory bank is needed from time to time, increasing average memory latency. When using multiple threads, this problem is reduced. Using more threads than CPU cores further increases speed-up to $5.3 \pm 0.5$ at 64 threads for volume openings and $4.6 \pm 0.4$ at 16 threads for elongation filtering (both 12 bit/voxel).

Intrigued by this result, we tested the performance of our algorithm on a single CPU (Intel P4-520, 3.0 GHz). The speed-up as a function of number of threads for smaller data sets in Table 1 is shown in Fig. 6. As can be seen, the performance on 8-bit/voxel data improves by between 49 percent to 72 percent at 32 or 64 threads in three cases. The xmas8 volume shows a different pattern, increasing steadily to a 61 percent increase in speed at 128 threads. We also measured cache hit/miss statistics using valgrind (http://valgrind.org/) and computed the expected changes in computing time assuming latency ratio's of L1:L2:Memory = 1:9:115. The results of these measurements are also shown in Fig. 6.

The 12-bit/voxel data (not shown) yield a slightly different pattern, peaking between 16 and 32 threads, with the exception of xmas12, which peaks at 64. Performance gains are also more modest, between 18 percent and 44 percent. These differences can be explained by the increased cost of merging the trees.

We also tested the algorithm in [35] to build the trees for each strip instead of that in [11]. However, in the data sets tested, the former turns out to be consistently 20-30 percent slower than the latter algorithm.

The difference in speed-up between the 12-bit and 8-bit data sets can easily be understood from the increased cost of merging, given the higher value of $G$. To assess the average merge cost, we measured the number of nodes merged on the average per edge processed during the merging phase, and the average depth of recursion of the `levroot` function. All measurements were done using 64 threads. It was found that, for 8-bit data, the average merge depth is $0.64 \pm 0.31$ and, for 12-bit data, $4.3 \pm 2.2$. In

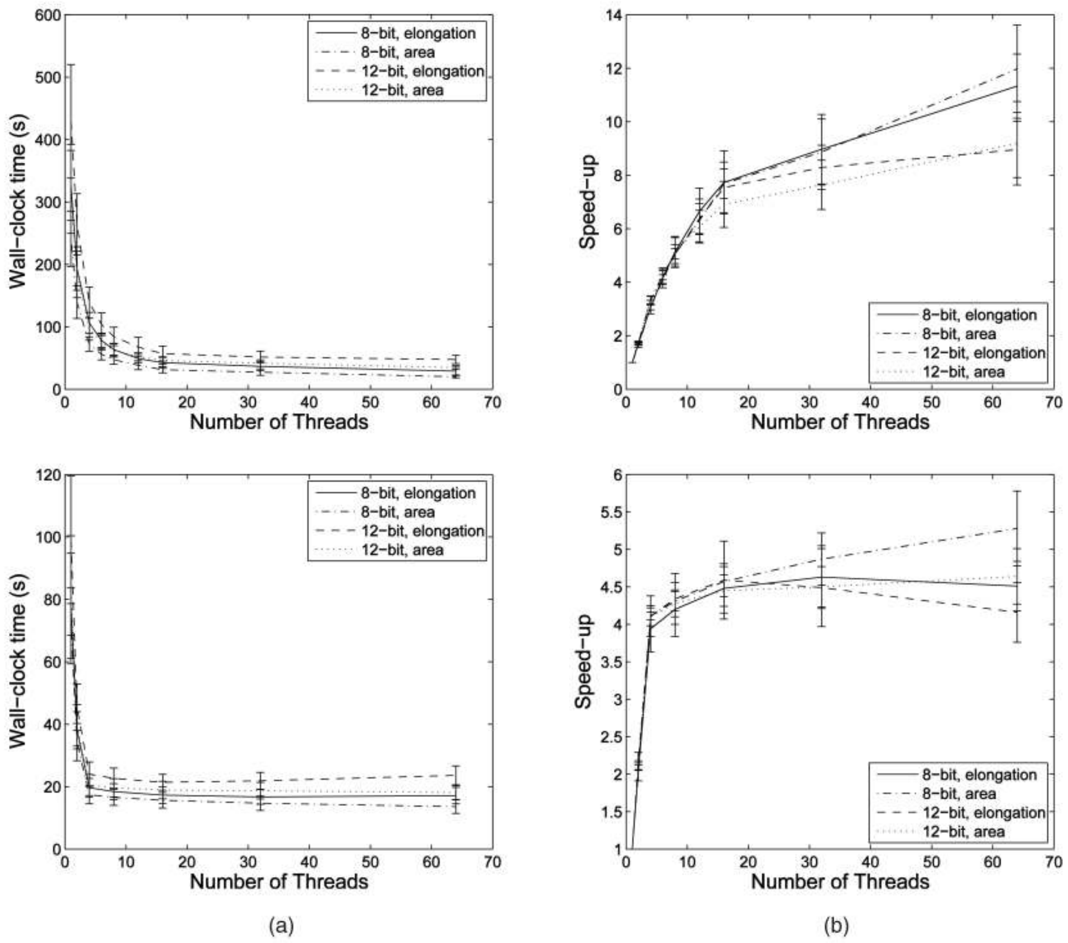

Fig. 5. (a) Timings and (b) speed-up of the parallel algorithm for Max-Tree building and filtering as a function of number of threads for attribute filtering. The top row shows the results for the Onyx 3400, the bottom for the Opteron-based machine (two dual-core chips). In either case, there is an increase in speed after the number of threads exceeds the number of processor cores.

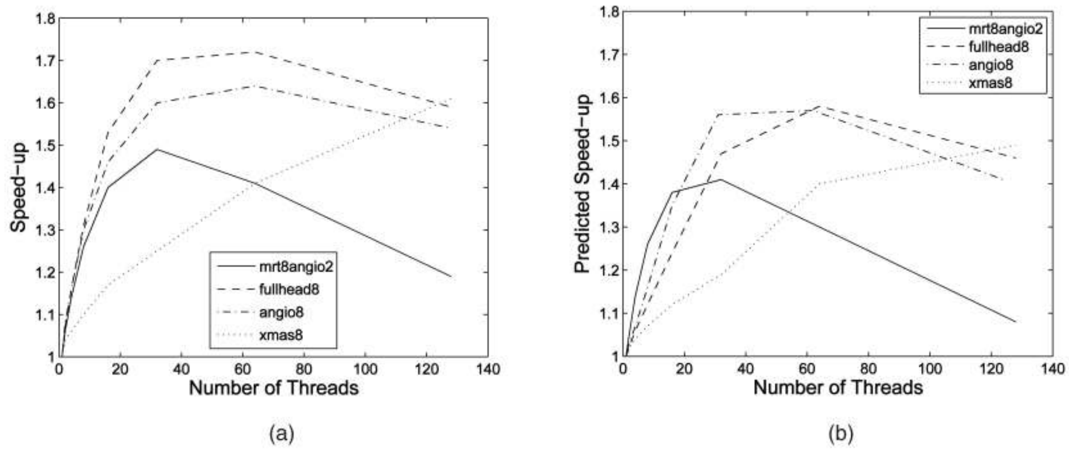

Fig. 6. Speed-up on a single core CPU (P4-520, 3.0 GHz) of the parallel algorithm for Max-Tree building and filtering as a function of number of threads, for smaller volume data sets in Table 1: (a) 8-bit data, (b) predicted from cache hit/miss data.

both cases, this is far smaller than $G$ (256 and 4,096, respectively). The mean recursion depth of the `levroot` function is $1.14 \pm 0.11$ for 8-bit data, and $0.81 \pm 0.07$ for 12-bit data; again, much lower than the worst case. The difference is due to the fact that flat zones decrease on the average as $G$ increases. Unsurprisingly, data sets with many high-contrast features had comparatively large merge depths (5-8 at 12 bit/voxel typically for CT-scans with bony structures), whereas sparser or lower contrast images volumes yielded lower numbers of merges per edge (1.6-3.5 at 12 bit/voxel typically for MR-angiograms).

## 9 Conclusions

The proposed algorithm shows a good degree of speed-up (between 7 and 14 on 16 processors of the Onyx and between 3.6 and 5.7 on the four processor cores of the Opteron-based machine), even on complicated volume data sets with high gray-level resolution. If the algorithm is run with the number of threads set to one, it is identical in speed to the original sequential algorithm [11]. Therefore, the speed-up is with respect to the sequential algorithm. Any other sequential algorithm for the Max-tree that can be adapted to, or actually uses, the union-find approach to voxel labeling can be inserted into this framework, and similar speed-ups are to be expected, *provided* that the same perfect path-compression per node is maintained in each slice, i.e., all members *at level h* of $D_h^k$ point directly to their level root. Examples of such algorithms are given in [18], [31], and [35]. The latter two have been implemented for floating point data as well, which is important in certain application domains. However, higher values of $G$ mean higher cost of merging, so the speed-up may be lower in the floating-point case. In that case, decreasing the cost of merging by using oct-trees might be advisable.

In future work, we will study the parallel computation of the topological watershed [25], [26]. A variant of the Max-Tree that implements attribute filters with so-called second-generation connectivities [28], called the Dual-Input Max-Tree [38], has already been parallellized using our approach [39].

## Appendix A

The following list summarizes the notational conventions used in this paper:

- $B$, $C$, and $D$ are arbitrary sets subsets of $V$.
- $Anc(x)$ is a set of ancestors of $x$.
- $anc(x, h)$ is a function returning the oldest ancestor of $x$ down to level $h$.
- $\mathcal{A}$ is an attribute function.
- $D(x)$ is a set of descendants of $x$.
- $D_h^i$, $D_h^k$ is the peak component or dome, a connected component of threshold set $V_h$, with indices $i$ and $k$ from some index set.
- $\mathcal{E}$ is a universal set ($\mathbb{R}^n$ or $\mathbb{Z}^n$).
- $E$ is a set of edges.
- $E_h$ is a set of edges, restricted to nodes in threshold set $V_h$.
- $E_h^*$ is a reflexive transitive closure of $E_h$.
- $f$ is a gray-scale image.
- $\varphi_1$ is an elongation measure.
- $G$ is a number of gray levels.
- $\Gamma_T$ is a binary trivial filter.
- $\Gamma^T$ is a binary attribute filter.
- $\gamma^T$ is a gray-scale attribute filter.
- $\Gamma_x$ is a connected opening at point $x \in \mathcal{E}$.
- $h$ is a threshold level.
- $\mathcal{I}(X)$ is a trace of the moment-of-inertia tensor of $X$.
- $K$ is a number of threads.
- *LeRo* is a set of level roots.
- $levroot(x)$ is a function returning level root of $x$.
- $\lambda$ is an attribute threshold.
- $M$ is a commutative monoid (for attribute computation).
- $N$ is a number of pixels (voxels).
- $N_p$ is a number of processors.
- $\mathcal{P}(\mathcal{E})$ is a power set (set of all subsets) of $\mathcal{E}$.
- $P$ is a binary relation corresponding to array `par`.
- $Par(x)$ returns the level root of $\texttt{par}[x]$ unless $\texttt{par}[x] = \perp$.
- `par` is an array containing parent pointers for all vertices $x \in V$.
- $R$ is an arbitrary binary relation on $V$.
- $R_h$ is a restriction of $R$ to $V_h$.
- $R_h^\sharp$ is a symmetric reflexive transitive closure of $R_h$.
- $T$ is a filter criterion.
- $V$ is an image domain or (equivalently) a set of nodes.
- $V_h$ is a threshold set at level $h$ (subset of $V$).
- $V^p$ is a subdomain of $V$ assigned to processor $p$.
- $\mathcal{V}(X)$ is a volume of $X$ (in 3D).
- $X$ is a binary image.
- $\perp$ is a `null` pointer.
- $A \bowtie B$ shows symmetrical difference: $A \bowtie B = (A \setminus B) \cup (B \setminus A)$.
- $\widehat{+}$ is a binary operator of commutative monoid $M$.
- $\widehat{\sum}$ is a "summation" operator associated with $\widehat{+}$.
- $\widehat{0}$ is a neutral element of $\widehat{+}$.

## References

[1] F.J. Seinstra, D. Koelma, and J.M. Geusebroek, "A Software Architecture for User Transparent Parallel Image Processings," *Parallel Computing,* vol. 28, pp. 967-993, 2002.

[2] T. Hirata, "A Unified Linear-Time Algorithm for Computing Distance Maps," *Information Processing Letters,* vol. 58, pp. 129-133, 1996.

[3] A. Meijster, J.B.T.M. Roerdink, and W.H. Hesselink, "A General Algorithm for Computing Distance Transforms in Linear Time," *Proc. Int'l Symp. Math. Morphology (ISMM '00),* pp. 331-340, June 2000.

[4] P. Salembier and J. Serra, "Flat Zones Filtering, Connected Operators, and Filters by Reconstruction," *IEEE Trans. Image Processing,* vol. 4, pp. 1153-1160, 1995.

[5] E.J. Breen and R. Jones, "Attribute Openings, Thinnings and Granulometries," *Computer Visualization and Image Understanding,* vol. 64, no. 3, pp. 377-389, 1996.

[6] H.J.A.M. Heijmans, "Connected Morphological Operators for Binary Images," *Computer Visualization and Image Understanding,* vol. 73, pp. 99-120, 1999.

[7] J.C. Klein, "Conception et Réalisation d'une Unité Logique Pour l'analyse Quantitative d'images," PhD dissertation, Nancy Univ., 1976.

[8] L. Vincent, "Morphological Grayscale Reconstruction in Image Analysis: Application and Efficient Algorithm," *IEEE Trans. Image Processing,* vol. 2, pp. 176-201, 1993.

[9] F. Cheng and A.N. Venetsanopoulos, "An Adaptive Morphological Filter for Image Processing," *IEEE Trans. Image Processing,* vol. 1, pp. 533-539, 1992.

[10] L. Vincent, "Morphological Area Openings and Closings for Grey-Scale Images," *Proc. NATO Shape in Picture Workshop: Math. Description of Shape in Grey-Level Images,* Y.-L. O, A. Toet, D. Foster, H.J.A.M. Heijmans, and P. Meer, eds., pp. 197-208, 1993.

[11] P. Salembier, A. Oliveras, and L. Garrido, "Anti-Extensive Connected Operators for Image and Sequence Processing," *IEEE Trans. Image Processing,* vol. 7, pp. 555-570, 1998.

[12] E.R. Urbach and M.H.F. Wilkinson, "Shape-Only Granulometries and Grey-Scale Shape Filters," *Proc. Int'l Symp. Math. Morphology (ISMM '02),* pp. 305-314, 2002.

[13] E.R. Urbach, J.B.T.M. Roerdink, and M.H.F. Wilkinson, "Connected Shape-Size Pattern Spectra for Rotation and Scale-Invariant Classification of Gray-Scale Images," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 29, pp. 272-285, 2007.

[14] M.H.F. Wilkinson and M.A. Westenberg, "Shape Preserving Filament Enhancement Filtering," *Proc. Medical Image Computing and Computer-Assisted Intervention (MICCAI '01),* W.J. Niessen and M.A. Viergever, eds., pp. 770-777, 2001.

[15] A. Meijster and M.H.F. Wilkinson, "A Comparison of Algorithms for Connected Set Openings and Closings," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 24, no. 4, pp. 484-494, Apr. 2002.

[16] R.E. Tarjan, "Efficiency of a Good But Not Linear Set Union Algorithm," *J. ACM,* vol. 22, pp. 215-225, 1975.

[17] R. Jones, "Connected Filtering and Segmentation Using Component Trees," *Computer Visualization and Image Understanding,* vol. 75, pp. 215-228, 1999.

[18] L. Najman and M. Couprie, "Building the Component Tree in Quasi-Linear Time," *IEEE Trans. Image Processing,* vol. 15, pp. 3531-3539, 2006.

[19] S. Morse, "Concepts of Use in Computer Map Processing," *Comm. ACM,* vol. 12, pp. 147-152, 1969.

[20] J. Roubal and T.K. Peucker, "Automated Contour Labeling and the Contour Tree," *Proc. Int'l Symp. Computer-Assisted Cartography (AUTOCARTO '85),* pp. 472-481, 1985.

[21] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore, "Contour Trees and Small Seed Sets for Iso-Surface Traversal," *Proc. 13th Ann. Symp. Computational Geometry,* pp. 212-220, 1997.

[22] H. Carr, J. Snoeyink, and U. Axen, "Computing Contour Trees in All Dimensions," *Computational Geometry,* vol. 24, pp. 75-94, 2003.

[23] V. Pascucci and K. Cole-McLaughlin, "Efficient Computation of the Topology of Level Sets," *Proc. IEEE Visualization,* pp. 187-194, 2002.

[24] P. Monasse and F. Guichard, "Fast Computation of a Contrast Invariant Image Representation," *IEEE Trans. Image Processing,* vol. 9, pp. 860-872, 2000.

[25] G. Bertrand, "On Topological Watersheds," *J. Math. Imaging and Vision,* vol. 22, pp. 217-230, 2005.

[26] M. Couprie, L. Najman, and G. Bertrand, "Quasi-Linear Algorithms for the Topological Watershed," *J. Math. Imaging and Vision,* vol. 22, pp. 231-249, 2005.

[27] U. Braga-Neto and J. Goutsias, "A Theoretical Tour of Connectivity in Image Processing and Analysis," *J. Math. Imaging and Vision,* vol. 19, pp. 5-31, 2003.

[28] J. Serra, "Connectivity on Complete Lattices," *J. Math. Imaging and Vision,* vol. 9, pp. 231-251, 1998.

[29] P. Maragos and R.D. Ziff, "Threshold Decomposition in Morphological Image Analysis," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 12, no. 5, pp. 498-504, May 1990.

[30] W.H. Hesselink, A. Meijster, and C. Bron, "Concurrent Determination of Connected Components," *Science of Computer Programming,* vol. 41, pp. 173-194, 2001.

[31] C. Berger, T. Geraud, R. Levillain, N. Widynski, A. Baillard, and E. Bertin, "Effective Component Tree Computation with Application to Pattern Recognition in Astronomical Imaging," *Proc. Int'l Conf. Image Processing,* Sept. 2007.

[32] C. Fiorio and J. Gustedt, "Two Linear Time Union-Find Strategies for Image Processing," *Theoretical Computer Science (A),* vol. 154, pp. 165-181, 1996.

[33] B.A. Galler and M.J. Fischer, "An Improved Equivalence Algorithm," *Comm. ACM,* vol. 7, pp. 301-303, 1964.

[34] S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs," *Acta Informatica,* vol. 6, pp. 319-340, 1976.

[35] W.H. Hesselink, "Salembier's Min-Tree Algorithm Turned into Breadth First Search," *Information Processing Letters,* vol. 88, no. 5, pp. 225-229, 2003.

[36] M.K. Hu, "Visual Pattern Recognition by Moment Invariants," *IRE Trans. Information Theory,* vol. 8, pp. 179-187, 1962.

[37] A. Kanitsar, T. Theussl, L. Mroz, M. Sramek, A.V. Bartoli, B. Csebfalvi, J. Hladuvka, D. Fleischmann, M. Knapp, R. Wegenkittl, P. Felkel, S. Roettger, W.P. Stefan Guthe, and M.E. Groeller, "Christmas Tree Case Study: Computed Tomography as a Tool for Mastering Complex Real World Objects with Applications in Computer Graphics," *Proc. Visualization,* pp. 489-492, 2002.

[38] G.K. Ouzounis and M.H.F. Wilkinson, "Mask-Based Second Generation Connectivity and Attribute Filters," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 29, pp. 990-1004, 2007.

[39] G.K. Ouzounis and M.H.F. Wilkinson, "A Parallel Dual-Input Max-Tree Algorithm for Shared Memory Machines," *Proc. Int'l Symp. Math. Morphology (ISMM '07),* pp. 449-460, 2007.

**Michael H.F. Wilkinson** received the MSc degree in astronomy from the Kapteyn Laboratory, University of Groningen, in 1993 and the PhD degree from the Institute of Mathematics and Computing Science (IWI), University of Groningen, in 1995. He was appointed as a researcher at the Centre for High Performance Computing, University of Groningen, working on simulating the intestinal microbial ecosystem on parallel computers. He is currently a lecturer at IWI. His research interests include image analysis, connected morphology, deformable models, and visualization, especially in biomedical applications. He is a senior member of the IEEE and the IEEE Computer Society.

**Hui Gao** received the PhD degree in computing science from the University of Groningen in 2005. He is an associate professor of algorithm design and analysis at the School of Computer Science and Engineering, University of Electronic Science and Technology of China. Since July 2007, the WSNMPO project on Complex network analysis, supported by the National High-Tech Research and Development Plan of China, has become the main area of his work.

**Wim H. Hesselink** received the PhD degree in mathematics from the University of Utrecht in 1975. He is a professor of program correctness. After 10 years of research in algebraic groups and Lie algebras, he moved on to computing science. In 1986/1987, he was on sabbatical leave with the University of Texas, Austin. He is currently an editor of the journal *Science of Computer Programming*. His research interests include the development and verification of concurrent programs, application of mechanical theorem provers, algorithms for image processing, knowledge-based programs, and molecular computing by polycyclic hydrocarbons.

**Jan-Eppo Jonker** received the MSc degree in theoretical physics and the MSc degree in computer science from the University of Groningen in 1968 and 1989, respectively. From 1968 to 1976, he studied the electronic impurities in iron. From 1976 to 1989, he studied medical aspects of road accidents. He was an appointed lecturer at the Institute for Mathematics and Computing Science, University of Groningen. He has recently retired as a lecturer in computer science. His research interests include programming methodology, parallel computations, and delay-insensitive circuits.

**Arnold Meijster** received the PhD degree in computing science with a focus on parallel algorithms for morphological image processing from the Institute of Mathematics and Computing Science, University of Groningen, The Netherlands, in 2004. He is currently working at the Centre for High Performance Computing and Visualization, University of Groningen, working on the development of parallel and distributed algorithms for various high-performance computing applications.