Chapter 5

A Concurrent Algorithm for Connected Set Filtering, and its Application to Interactive Visualization

Abstract

This chapter presents a method for combined interactive filtering and visualization of volumetric data on shared memory architectures. The user can interactively set the filter parameters of a shape preserving class of morphological filters, called connected filters, and immediately see a volume rendering of the resulting filtered volume data set. The filters work by computing some attribute describing the shape or size for each connected component of the volume. The user can decide which components to preserve based on some threshold. We use a method in which the computation of attributes and connected component analysis is separated from the decision stage of the filtering process. Both stages are performed in parallel. The first stage is a kind of initialization for the (faster) decision stage, which can be run many times with different threshold values, allowing interactive filtering and visualization of the results. We have implemented the program, using Posix Threads, and ran experiments on an SGI Onyx 3400 with 16 CPUs.

5.1 Introduction

In this chapter, which is a major extension of [55], we present a method for combined filtering and visualization of volumetric data. The user can interactively set the filter parameters. An efficient parallel shared memory algorithm is used to perform the actual filtering, allowing manipulation and visualization at interactive rates.

We use a shape preserving class of morphological filters called connected filters. Connected filters have received much attention in recent years, in algorithm development [57, 83] and applications [87, 104]. Connected filters are shape preserving, because they never introduce new edges in images. A subclass of these are attribute filters, the first of which were area openings and closings, which remove image detail smaller than a particular area [21]. These in turn were extended by Breen and Jones to attribute openings which accept or reject image details based on any of a wide range of size parameters [19]. They also put forward the idea of attribute thinnings, which
allow image filtering based on shape, rather than size criteria. This idea has been formalized to so called shape filters [93], which have been applied to the problem of vessel enhancement in angiographic volume data sets [104].

In the binary case, attribute filters work by computing some parameter (or attribute) describing the shape or size for each connected component, and then deciding which component to preserve based on some threshold or window on these parameter values. An example is shown in Fig. 5.1, in which the nuts are separated from the bolts based on the number of holes. In the grey scale case, attribute filters can be implemented simply by thresholding the image at each grey level, applying a binary filter to each thresholded image and recombining them. Faster algorithms have been developed (see chapter 4, [57]), but in the case of volumetric data, filtering is still too slow to be interactive in many cases. For example, filtering of a $256^3$ data set for vessel enhancement may take 10 to 20 seconds, even on a Pentium 4 at 1.9 GHz with 800 MHz RDRAM. This is a serious drawback when optimal threshold settings are interactively determined. In this chapter, we use the Max-tree algorithm of Salembier et al. (see chapter 4, [83]), in which the computation of attributes and connected component analysis (stage 1) is separated from the decision of the filtering process (stage 2). We have shown in chapter 4 and in [57] that this algorithm is usually slower than the UNION-FIND algorithm in case we want to filter a data set with a particular parameter $\lambda$, but in this chapter we are interested in the case where we filter (interactively) with various attribute values. In this context the Max-tree algorithm is very suitable.

For both stages a parallel algorithm is presented. After performing the first stage as an initialization, we can perform the (much faster) decision stage many times with different threshold values, allowing interactive filtering and visualization of the results. For the construction of the tree, Salembier uses a region growing algorithm based on queues. We prefer to use a generalization of Tarjan’s Union-Find algorithm which does not use queues, and allows concurrent construction on shared memory systems.

The outline of this chapter is as follows. In section 5.2 we briefly recall the Max-tree data structure, and its use for attribute filtering. In section 5.3, we introduce a sequential algorithm for constructing Max-trees and in section 5.4 we introduce an algorithm that merges Max-trees

![Figure 5.1](image-url)

**Figure 5.1.** Decomposition of a binary image of nuts and bolts of different sizes into two different shape classes: (left) original image; (middle) filter with criterion "number of holes > 0"; (right) difference between (left) and (middle).
into a larger Max-tree concurrently. Section 5.5 describes an algorithm for the accumulation of attributes in a Max-tree. Section 5.6 discusses a parallel algorithm for the filtering (second) stage. In section 5.7, we show some performance results and the application of the algorithm to interactive visualization. Conclusions are drawn in section 5.8.

5.2 Max-trees

An efficient implementation of attribute filters relies on computing both the hierarchy of connected components in the data set, and some attribute for each component to use as a filter criterion. A Max-tree representation of the data set was introduced by Salembier et al. [83] as a versatile structure to separate the filtering process from the computation of connected components and attributes. The building of this tree structure is called the construction phase, while its use for filtering is called the filtering phase. In this section we will briefly discuss this data representation, and show how it can be used to perform filtering.

Let $V \subseteq \mathbb{Z}^n$ be some image domain ($n = 2$ for images, $n = 3$ for volumes), and $f : V \rightarrow \mathbb{R}$ the grey scale image (volume) under study. Implicitly we assume the existence of some neighborhood graph (i.e. a grid) on $V$.

A set $F \subset V$ is called a flat zone or connected component if for all $p, q \in F$ there exists a path from $p$ to $q$ along which the function value is constant, and the set $V$ is maximal in size. The threshold set $V_h(f)$ of image $f$ is the set of points that remain after thresholding at level $h$, i.e.

$$V_h(f) = \{ x \in V | f(x) \geq h \}. \quad (5.1)$$

A peak component at a grey level $h$ is a connected component of the threshold set $V_h(f)$. The number of these peak components is finite and can thus be enumerated. We introduce the notation $P^k_h$ to denote the $k$th peak component at level $h$. Max-tree nodes are peak components. In Fig. 5.2 we see a one-dimensional discrete signal, and its corresponding Max-tree. The Max-Tree is a rooted tree: each node has a pointer to its parent which has a lower grey value. The nodes corresponding to the components with the highest intensity are the leaves (see Fig. 5.2). The root node represents the set of pixels belonging to the background, that is the set of pixels with the

![Figure 5.2. Peak components of a discrete signal (left) and the corresponding Max-tree (right).](image-url)
lowest intensity in the image. Hence the name Max-tree: the leaves correspond to the regional maxima. This means that the Max-tree can be used for filters that process peak components, i.e. start from the regional maxima. Conversely, a tree in which the leaves correspond to the minima is called a Min-tree and can be used for filters that process valley components, i.e. start from the regional minima.

During the construction phase, the Max-tree is built from the flat zones of the image. After this, the tree is subjected to the filtering phase. This filtering removes flat zones based on some property. These properties are defined by an attribute value $T(P_{kh})$ of a node $P_{kh}$, from a set (usually $\mathbb{R}$ or $\mathbb{Z}$) with an order $\leq$. Given a threshold value $\lambda$ from this universe, the algorithm decides whether to preserve, or remove a node.

Salembier describes four different rules for the algorithm to filter the tree: the Min, the Max, the Viterbi, and the Direct decision. In addition, Wilkinson and Urbach [93] introduced another strategy, called the Subtractive decision. The decisions of these rules are as follows:

**Min** A node $P_{kh}$ is removed if $T(P_{kh}) < \lambda$ or if one of its ancestors is removed.

**Max** A node $P_{kh}$ is removed if $T(P_{kh}) < \lambda$ and all of its descendant nodes are removed as well.

**Viterbi** The removal and preservation of nodes is considered as an optimization problem. For each leaf node the path with the lowest cost to the root node is taken, where a cost is assigned to each transition. In this chapter we do not consider this rule. For details see [83].

**Direct** A node $P_{kh}$ is removed if $T(P_{kh}) < \lambda$; its pixels are lowered in grey level to the highest ancestor which meets the criterion, its descendants are unaffected.

**Subtractive** As above, but the descendants are lowered by the same amount as $P_{kh}$ itself.

Figure 5.2 shows the peak components of a 1-D discrete signal, and the corresponding Max-tree. The results of applying the Min, Max, Direct and Subtractive methods on this image with $\lambda = 10$ are shown in Fig. 5.3. Which of these rules is the most appropriate depends mainly on the application.

### 5.3 Construction of a Max-tree

In this section we will discuss an efficient sequential algorithm for constructing Max-trees. The data sets we are interested in usually consist of a large set of pixels or voxels together with some function $f$ on them that represents their grey values. Pixels have a natural neighborhood relationship, four-connectivity or eight-connectivity. A similar connectivity relation can be defined in 3D volumes. For our purposes it is convenient to regard the pixels or voxels as the vertices of an undirected graph with edges defined by the neighborhood relationship (underlying grid). This way, the algorithm becomes applicable to data sets of arbitrary dimensions.

We regard the pixels or voxels as the vertices of the undirected graph $G = (V, E)$, where $V$ is the set of vertices, and $E$ the set of edges. Let $N$ be the number of vertices of the graph, i.e. the size of $V$. We assume that pixels are numbered consecutively, starting from 0.
Recall that, for \( h \in \mathbb{R} \), the \( h \)-threshold set is the set \( V_h = \{ x \in V \mid f(x) \geq h \} \). The connected components of threshold set \( V_h \) are the components of the subgraph \((V_h, E_h)\) where \( E_h = E \cap (V_h \times V_h)\). These connected components are the peak components that constitute the nodes of the Max-tree. In this section, our aim is to compute these connected components for all levels \( h \), as well as the resulting Max-tree in a single computation.

Inspired by Tarjan’s union-find algorithm [88], which we used in a similar way in chapters 3, 4 and in [40, 57], we represent the connected components of the relations \( E_h \) by a forest structure induced by pointers to parent nodes. These pointers are collected in an array \( \text{par} \) which can be regarded as a (modifiable) function from nodes to nodes. We call a tree which is encoded in this \( \text{par} \) array, a \( \text{par} \)-tree.

We define \( \text{par}^n[x] \) by repeated application of \( \text{par} \), i.e. \( \text{par}^0[x] = x \) and \( \text{par}^{n+1}[x] = \text{par}^n[\text{par}[x]] \). A node \( x \in V \) is called a root of a \( \text{par} \)-tree if \( \text{par}[x] = x \). For each non-root, we can find its ancestors by repeated application of function \( \text{par} \). Since we are building Max-trees, i.e. the leaf nodes of the tree have highest grey levels, the grey levels of the nodes that are reached by repeated application of \( \text{par} \) are monotonically descending. In other words, we construct \( \text{par} \) such that for all \( x \in V \), we have \( f(\text{par}[x]) \leq f(x) \).

During the construction of the \( \text{par} \)-tree, we make sure that we do not introduce any cycles in the \( \text{par} \)-tree. For any vertex \( x \), we define the “oldest ancestor” up to \( h \) by means of the recursive function \( \text{anc} \) given by

\[
\text{anc}(x, h) = \begin{cases} x \vee f(\text{par}[x]) < h & \text{then } x \text{ else } \text{anc}(\text{par}[x], h). 
\end{cases}
\]

Since \( \text{par} \) is acyclic, this recursion always terminates.

In Fig. 5.4, an example is shown of a small artificial image and a possible \( \text{par} \)-tree representation. The pixels are numbered from 0 to 8, and between parenthesis their function value are denoted. We use 4-connectivity. Consider pixel \( p = 5 \) with \( f(5) = 4 \). This pixel is a local...
maximum, and therefore becomes a leaf node of the \( \text{par} \)-tree. It has neighboring pixels, which we will denote with \( n, s, w \) (north, south, west). From the figure we can see that \( n = 2, w = 4, \) and \( s = 8 \), with function values \( f(2) = 3, f(4) = 1, f(8) = 0 \). Since pixel \( n = 2 \) has the highest value, \( \text{par}[5] \) is set to 2. Furthermore, since pixel \( w \) is reachable from 5, via a descending path, we decide to set \( \text{par}[2] \) to 4. The interpretation is as follows. When we threshold the image at level 3, we find a connected component containing pixels 2 and 5. When we threshold at level 1, the component is expanded with (among others) pixel 4. Since pixel \( s \) is also a neighbor of 5, their must also be a \( \text{par} \)-path from 4 to 8. Pixel 8 has the lowest function value of all pixels, and therefore becomes the root of the entire \( \text{par} \)-tree. Verifying the remaining parent-pointers is left as an exercise for the reader. Note, that the property \( f(\text{par}[x]) \leq f(x) \) allows more than one possible tree: for example we could set \( \text{par}[6] = 4 \) instead of \( \text{par}[6] = 7 \).

As we can see in Fig. 5.4, following a \( \text{par} \)-pointer does not always yield a parent node with a lower function-value, e.g. \( f(0) = f(1) = f(4) = f(7) = 1 \). This is the result of the fact that these pixels are part of the same flat zone of the image, and we encode this connectedness in the \( \text{par} \)-tree. We define a vertex \( x \) to be a level root if \( x \) is the root of the tree (i.e. \( \text{par}[x] = x \)) or \( f(\text{par}[x]) < f(x) \). It is easy to see that \( h \leq f(x) \) implies that \( x' = \text{anc}(x, h) \) satisfies \( f(\text{par}[x']) < h \leq f(x') \) and is therefore a level root.

We can determine whether two pixels \( p \) and \( q \) with the same function value are in the same peak component by computing their level roots. If the pixels have the same level root, they are in the same peak component, otherwise they belong to different peak components.

We now start with the construction of an iterative algorithm for building the \( \text{par} \)-tree. We start with a forest of singleton-trees, by setting \( \text{par}[x] = x \), for each \( x \in V \). We aim now at an iterative algorithm in which at each iteration an edge \( (p, q) \) is removed from \( E \), and the \( \text{par} \)-structure gets updated by merging the trees to which \( p \) and \( q \) belong. This process stops when

![Figure 5.4](https://example.com/image.png)

**Figure 5.4.** An image (left) and a possible \( \text{par} \)-tree structure (right). Nodes are numbered \( p(q) \), where \( p \) is the node number, and \( q = f(p) \).
Figure 5.5. Pair replacement.

We therefore consider some \((p, q) \in E\). Let \(s\) be the minimum of the function values of \(p\) and \(q\), i.e. \(s = \min(f(p), f(q))\) (see Fig. 5.5). Let us assume that \(s = f(q)\), and that \(p\) has an ancestor \(r\) (which is also a level root) with \(f(\text{par}[r]) < s \leq f(r) \leq f(q)\). In this case we can replace \(p\) by \(r\), since \(p\) and \(r\) are in the same peak component, and so are all nodes on the \(\text{par}\)-path from \(p\) to \(r\). Using this argument, we replace the pair \((p, q)\) by the pair \((x, y)\), where \(x = \text{anc}(p, s)\) and \(y = \text{anc}(q, s)\) (see Fig. 5.5).

We now would like to set \(\text{par}[x]\) to \(y\). However, if we do this directly, we disconnect the ancestors of \(x\) from \(x\) and its descendant. Therefore, we store \(\text{par}[x]\) in a temporary variable \(z\), before making \(y\) the parent of \(x\).

We are now confronted with a new pair \((y, z)\) with \(f(z) < f(y)\). For this pair we repeat the process of merging by setting \(x\) to \(y\), and \(y\) to \(z\). This process terminates when \(x = y\), i.e. the trees containing \(x\) and \(y\) have already been merged before, or when \(\text{par}[x] = x\). In the latter case we have to make \(y\) the parent of \(x\), if \(x \neq y\). This analysis yields the following program fragment:

```
procedure merge(x, y) =
  s := \min(f(x), f(y)) ; x := \text{anc}(x, s) ; y := \text{anc}(y, s) ;
  if f(x) < f(y) then swap(x, y) end ; (* s = f(y) \leq f(x) *)
  while x \neq y \land \text{par}[x] \neq x do
    z := \text{par}[x] ; \text{par}[x] := y ;
    x := y ; y := \text{anc}(z, f(z)) ;
  end ;
```
5.3.1 Some optimizations

Determination of level roots can be made more efficient by shortening the parent paths whenever ancestors are determined. This is the classical method of path compression in Tarjan’s union-find algorithm [88]. In comparison with the classical case, however, we have to be careful that the parent path of any node must not loose the level root at the current level, which is defined by

\[ \text{levroot}(x) = \text{anc}(x, f(x)). \]

Path compression can be implemented by replacing function \text{levroot} by the following recursive procedure with side effect:

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

Path compression is incorporated in function \text{anc} by means of \text{levroot} via

```plaintext
procedure anc(x, s) returns V =
  while \( \text{par}[x] \neq x \land s \leq f(\text{par}[x]) \) do := \text{levroot}(\text{par}[x]) end ;
  \text{return} x ;
end .
```

Another important technique to speed up the construction algorithm, is to sort the nodes of \( V \) based on grey-level. This can be done in linear time using counting sort. In the program fragment shown above, we choose to remove the pair \((p, q)\) from \( E \) such that \( p \) is the unprocessed node with the highest grey level. When we construct the \text{par}-tree in this order, we are sure that \( p \) becomes the root of a newly merged tree, and thus we do not traverse a possibly long \text{par}-path starting at \( p \).

5.4 Concurrent Merging of the trees

Now that we have developed a procedure \text{merge} for merging two Max-trees, we can construct the Max-tree of an image by means of a simple concurrent algorithm.

We use \text{nthreads} processes that are numbered consecutively from 1 to \text{nthreads}. Each processor has a private integer variable \text{self} \( (1 \leq \text{self} \leq \text{nthreads}) \), with which it can identify itself. The set of vertices \( V = [0, N) \) is split in consecutive sub-domains \( V_p = [lwb(p), upb(p)) \), where \( lwb(p) = (p - 1) * N / \text{nthreads} \), and \( upb(p) = p * N / \text{nthreads} \). When each processor has computed a Max-tree for its own domain, it is time to merge the resulting trees in a global Max-tree.
We decide that the process with number $p$ ($p < \text{ntthreads}$) merges the local trees of domain $V_p$ and $V_{p+1}$. In order for process $p$ to start merging, it needs to know if it can safely start, i.e. the tree in which nodes from domain $V_{p+1}$ are incorporated is not being modified by any process. This can be achieved by merging in a binary tree fashion.

Consider the example shown in Fig. 5.6, where $\text{ntthreads} = 15$. Each node of the binary search tree represents a process number. All leaf nodes of the tree are odd numbered processes, while internal nodes are numbered even. When an even numbered process $p$ is ready constructing its local Max-tree, it signals leaf node $p - 1$ that it can merge the trees from domain $V_{p-1}$ and $V_p$. This signaling is implemented by means of an array of binary semaphores $\text{sem}$, which is indexed by process numbers. All semaphores are initialized with the value 0. A semaphore $s$ gets atomically incremented by the operation $V(s)$, while it gets decremented (acquired) by the operation $P(s)$. After signaling the leaf node, $p$ waits its turn by calling $P(\text{sem}[sel f])$. Process $p$ will get signaled at least once, since each node has a left child. When a right child exists as well, it will have to wait for a second signal before merging the Max-trees corresponding with domains $V_p$ and $V_{p+1}$. Since the binary tree is a search tree, the test $sel f + 1 \leq \text{ntthreads}$ suffices to determine whether a right child exists. When process $p$ gets woken up by its child(ren), it calls the routine $\text{glue}(sel f)$, which is a routine that calls $\text{merge}(x, y)$ for all $x \in V_{sel f}$ and $y \in V_{sel f+1}$. If $p$ is not the root of the binary tree, it signals its ancestor that it has finished merging. As we can see from Fig. 5.6, computing the ancestor of a process number $p$ involves computing the binary representation of $p$. The bit pattern of the direct ancestor of $p$ can be obtained from $p$ its bit-pattern, by replacing its least significant 1-bit by a 0-bit, and setting the preceding bit to 1. For example, process 2 has bit pattern 0010, and process 6 has bit pattern 0110. Both processes have process 4 (bit pattern 0100) as their parent. The computation of the ancestor of $p$ is performed by the routine $\text{ancestor}(p)$. The leaf nodes of the tree start the merging process. They act analogous to internal nodes, except that they do not have children that will signal them. Leaf nodes with number $p$ need only wait for a single signal (from process $p + 1$), to start merging.

The root of the binary tree is the process number which is the largest power of two contained
in the tree. Once a non-root process has signaled its ancestor, it will wait until it receives a signal from the root process, which is used as a kind of barrier. When the root process has finished its merging of Max-trees, it calls the routine `accumulate`, which computes attribute values for all nodes in the Max-tree. The algorithm for accumulating attributes is discussed in section 5.5. When all attributes have been computed, all processes are released from the barrier.

This analysis yields the following program fragment:

```plaintext
procedure globalmerge (self) =
  if nthreads = 1 then return ;
  if self mod 2 = 0 then (* internal nodes *)
    V(sem[sel f − 1]); P(sem[sel f]);
    if self+1 ≤ nthreads then
      P(sem[sel f]); glue(self);
    end ;
  if self ≠ root then
    V(sem[ancestor(self)]);
    P(sem[sel f]); (* enter barrier *)
  end else
  accumulate;
  for p := 1 to nthreads do
    if p ≠ self then V(sem[p]); (* release barrier *)
  end else (* leaf nodes *)
  if self+1 ≤ nthreads then
    P(sem[sel f]); glue(self);
  end :
  V(sem[ancestor(self)]);
  P(sem[sel f]); (* enter barrier *)
end ;
end.
```

## 5.5 Accumulating attributes

In this section we discuss a sequential algorithm for some kind of accumulation of attributes over the peak components, for all levels $h$. Currently, the accumulation of attributes is performed by a single process, i.e. the root process of the binary merge tree (cf. section 5.4). We are in the process of developing a concurrent version of the algorithm.

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. Using this specific attribute yields a filter called an area opening (in 2D) or a volume opening (in 3D). In this section we will use this special case as an example. The implementer of another attribute filter is responsible for implementing the correct accumulation procedure.

We introduce an array `accat` to store accumulated attributes. For any $x \in V$ the set of its descendants in the `par`-tree is given by
A Concurrent Algorithm for Connected Set Filtering, and its Application to Interactive Visualization

\[ D(x) = \{ p \in V \mid (\exists n \in \mathbb{N} : \text{par}^n[p] = x) \} . \]

In the case of an area (or volume) opening, we aim at the computation of

\[ \text{accat}[v] = \bigoplus_{w \in D(v)} 1, \text{ for all level roots } v \in V, \]

where \( \bigoplus \) denotes summing of attributes. Obviously, in this simple case \( \text{accat}[v] \) is the number of descendants of \( v \). However, we still prefer this more complicated expression, since it expresses the need for some addition operator on attributes. In this case, the operator is simply summing on integers, but it could very well be another operator, like taking minimum or maximum.

Initially, we set \( \text{accat}[v] = 0 \), for all \( v \in V \). Accumulating attributes simply boils down to traversing \( \text{par} \)-paths from each node of the Max-tree all the way down to the root of the tree. Consider some \( v \in V \). First, we add the unit-attribute (in this case 1) to \( \text{accat}[v] \). If \( v \) is not the root of the Max-tree, we traverse down the \( \text{par} \)-path of \( v \) until we reached the root. During this traversing we add the unit-attribute to \( \text{accat}[p] \) for all \( p \) visited along the \( \text{par} \)-path.

Several optimizations of this algorithm are possible. Before starting the accumulation process it is a good idea to perform perfect path-compression by performing \( \text{levroot}(v) \) for all \( v \in V \). This results in a tree in which each node is either a level root, or points directly to its level root. This optimization also facilitates fast filtering in stage 2. It is also possible to sort the nodes of \( V \) on grey-value, and start accumulating from the highest grey-value, in which it is not necessary to traverse \( \text{par} \)-paths all the way down to the root of the tree, yielding an algorithm with linear time complexity.

### 5.6 Filtering phase

When the global Max-tree has been computed, we can perform filtering very efficiently. During this stage, the Max-tree is not modified, therefore each process is allowed to read the data structure simultaneously without using semaphores.

We consider the case of direct filtering, i.e. each pixel is lowered in grey level to the level of the ancestor in the Max-tree with highest grey-level that satisfies the filter criterion. We introduce an array \( \text{filt} \) in which we store the filtered data set. The only thing a process \( p \) has to do, is to follow \( \text{par} \)-pointers for each point \( v \) of its private domain \( [\text{lwb}(p), \text{upb}(p)) \), until it reaches an ancestor \( w \) of \( v \) which satisfies the filter criterion. When \( w \) is found, \( \text{filt}[v] \) is set to \( f(w) \).

A simple optimization is possible. For all nodes \( u \) on the \( \text{par} \)-path from \( v \) to \( w \), we could set \( \text{filt}[u] \) to \( \text{filt}[w] \). This observation leads to the following procedure. We initialize \( \text{filt} \) such that it is \(-1\) everywhere, and use \( \text{filt}[p] \neq -1 \) to test whether \( \text{filt}[p] \) has already been determined. When we want to compute the filtered value for some \( v \) that finds \( q \) on its \( \text{par} \)-path, we can simply set \( \text{filt}[v] \) to \( \text{filt}[q] \). This yields the following code fragment.

```python
procedure directfilter (self, lambda) =
    for v := lwb(self) to upb(self) do
        if filt[v] = -1 then
```
5.7 Performance results and application to visualization

Once we have computed a Max-tree representation of a volume data set, including its attributes, we can filter the data set for different values of the threshold parameter $\lambda$ without recomputing a new Max-tree. This allows us to implement an interactive program, in which the user repeatedly filters the data set at different threshold levels, while looking at a volume rendering of the filtered result. We ran some tests on a $256 \times 256 \times 256$ magnetic resonance angiography data set, using an SGI Onyx 3400 system, with 16 CPUs and 20 Gb shared memory. This system is used for driving a large visualization facility at the University of Groningen, consisting of a reality theatre and a CAVE. We ran the program where a user stands in the CAVE, and looks at a volume rendering of the filtered data set. This rendering can interactively be manipulated by the user. The user can scale, rotate, and translate the data set, and can also change colors by manipulating a color lookup table. These manipulations are quite interactive, performing at typical frame rates of 10 to 20 frames per second. A 3D-mouse is used to set the filter parameter interactively. When the user chooses to change the filter parameter, a filtered data set is computed, which takes typically less than a second (using more than 4 CPUs). A radiologist from the university hospital steered
the application, and seemed quite comfortable with the interactivity supplied.

In order to get also some objective indications of the performance, we ran some timing experiments. With large data sets, cache misses obstruct reproducible behavior. We therefore decided to use relatively small data sets, for which a single image slice fits in the caches of the CPUs. It turns out that the load balancing of the algorithm is quite good under these circumstances, while the balancing becomes unpredictable when a processor runs frequently into cache miss-hits.

We ran the algorithm on the angiography data set a 1000 times, for different \( \lambda \) values which were chosen randomly from the domain \([0..4)\). Timing results were obtained for the first stage (constructing the Max-tree) and for the second stage (filtering). For each separate case we averaged the results over the 1000 runs. The results are shown in table 5.1.

The algorithm scales quite well. The build phase has an efficiency of more than 75%, while the filter phase has an efficiency of about 50%. Even though we have the availability of 16 processors, the run using 32 threads is slightly better than the run using 16 threads. This is explained by load imbalance, which can be better resolved by the operating system when we use more threads.

### 5.8 Conclusions

In this chapter, we have proposed a method for interactive filtering of volume data sets based on a class of shape preserving filters. We have briefly introduced such filters and have described how they can be implemented efficiently using Max-trees. The Max-tree approach splits the filtering task in two stages. The first stage is a construction of a tree, while the second stage performs actual filtering using this tree. We have presented a concurrent algorithm for constructing Max-trees and filtering data sets using these trees. The performance of the program scales quite well in the number of threads, allowing interactive visualization and manipulation of the filtered data set. Experiments have shown that users can manipulate the visualization interactively, at satisfactory frame rates.
Figure 5.7. Magnetic resonance angiography volume data set (size $256^3$) filtered interactively with an attribute thinning as shape filter. The attribute used was $I/V^{5/3}$, with $I$ the moment of inertia, and $V$ the volume of a peak component; the top left-hand image is the original, in the others the attribute threshold was 0.5, 1.0, 1.5, 2.0, 2.5, 3.0 and 4.0, respectively. This attribute is a shape dependent number that expresses elongation. Visualization was done by maximum intensity projection.