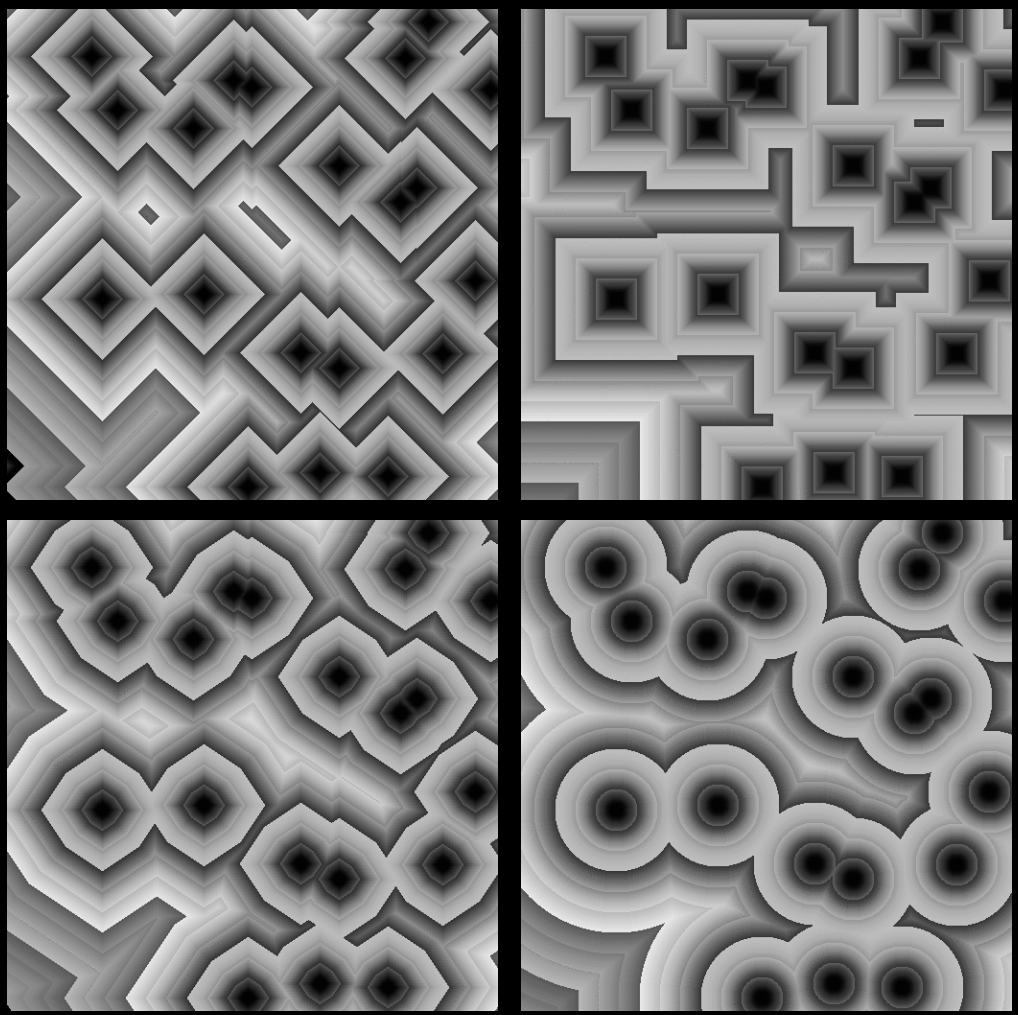
Efficient Sequential and Parallel Algorithms for Morphological Image Processing



Arnold Meijster

Efficient Sequential and Parallel Algorithms for Morphological Image Processing

Arnold Meijster

Cover: Distance transforms of 20 randomly placed foreground pixels: Manhattan distance (top left), Chessboard distance (top right), Chamfer-3-4 distance (bottom left), and Exact Euclidean distance (bottom right).

Meijster, A.

Efficient Sequential and Parallel Algorithms for Morphological Image Processing
Arnold Meijster.
Thesis Rijksuniversiteit Groningen.
ISBN 90-367-1978-x (digital version)

Printed by Universal Press (www.universalpress.nl).
ISBN 90-367-1977-1

RIJKSUNIVERSITEIT GRONINGEN

Efficient Sequential and Parallel Algorithms for Morphological Image Processing

Proefschrift

ter verkrijging van het doctoraat in de
Wiskunde en Natuurwetenschappen
aan de Rijksuniversiteit Groningen
op gezag van de
Rector Magnificus, dr. F. Zwarts,
in het openbaar te verdedigen op
vrijdag 5 maart 2004
om 14.45 uur

door

Arnold Meijster
geboren op 5 maart 1970
te Roden

Eerste promotor:	Prof. dr. J. B. T. M. Roerdink
Tweede promotor:	Prof. dr. W. H. Hesselink
Beoordelingscommissie:	Prof. dr. ir. J. F. Groote
	Prof. dr. H. A. de Raedt
	Prof. dr. P. Salembier

ISBN: 90-367-1978-x

Contents

Chapter 1

Introduction

Nowadays, many personal computers (PCs) are equipped with fancy video-cards, digital cameras, and software for editing and processing digital pictures. Most of the filter operations performed with these software packages are for esthetic purposes, i.e. they improve the visual quality of the picture. The resolution of these pictures is typically in the order of 1 million up to 4 million pixels. The filters supplied by the picture editing packages can handle these sizes at interactive rates, i.e. typically each filter takes up to a few seconds processing time.

For the creation of a few visually appealing pictures, these processing times are perfectly acceptable. However, if we want to process much larger images, or many more images (e.g. hundreds or thousands) in sequence, such processing times become unpractical. For example, a simple smoothing operation which removes noise contamination from a 5000×5000 pixel image requires over 250 million multiplies and additions, a process which would take several minutes to complete on a common PC. Such computation times are clearly not acceptable for real-time image analysis tasks. Fortunately, the computation time can be drastically reduced by using more clever (efficient) algorithms and parallel computing techniques.

In this thesis the focus is on efficient sequential and parallel algorithms for *morphological image processing*. *Mathematical Morphology* is a field of non-linear image processing, based on minimum and maximum operations. Good introductions to morphological image processing are [38, 86]. The aim of this type of image processing is to extract or enhance features from images based on shape. In this context, accuracy plays a crucial role, in contrast with picture processing where accuracy is usually traded for visually appealing results.

Many morphological operators can be computed very fast. However, most operators are hardly useful on their own. They are applied in sequence, adding up processing time. Most operators can be extended easily to 3D-image processing, requiring even more processing time. Morphological image processing based systems are typically used for real-time surveillance tasks in industrial systems, medical image processing, optical character recognition, texture analysis, etc.

Many algorithms for various morphological image operators have been published. The aim of this thesis is to speed up some of these algorithms by means of improvements in the original algorithms, or by the use of parallel computing techniques. Nowadays, many desktop workstations contain multiple CPUs, which could be exploited by parallelization of existing algorithms for morphological operators. Since these machines become more common, the focus is on par-

allelization for shared memory or SMP architectures (see section 1.2), which can be desktop workstations with multiple CPUs as well as massively parallel shared memory supercomputers.

1.1 Introduction to Mathematical Morphology

In this section the field of *mathematical morphology* is briefly introduced. For the time being, images under consideration are *binary images*, i.e. black-and-white images. In most image processing literature, images are looked upon as mappings from a set of coordinates (pixels) to a set of image values. In the case of binary images, this set contains only two values (0 and 1, or black and white). Instead of referring to the colors black and white, we prefer to use the terms *foreground* and *background*. In binary morphological image processing, operators are typically performed on the foreground or background only, and not on all pixels in the image. Therefore, it is more natural to look upon binary images as sets. The universe under consideration is the entire set of pixels, while the image itself is the set of foreground pixels (object pixels). The background of the image is obtained by taking the complement of the image with respect to the universe.

In this section the set of pixels $D \subseteq \mathbb{Z}^N$ is the universe under consideration. Underlying the set of pixels there is a neighborhood relation, denoted by a graph $G = (D, E)$, with $E \subseteq D \times D$. Two pixels p and q are said to be neighboring pixels if and only if $(p, q) \in E$. In practice, there is only a limited number of neighborhood relations used, e.g. in the case $D \subseteq \mathbb{Z}^2$, mainly three relations are used:

- 4-connectivity: neighbors north, west, east, and south on rectangular grid.
- 8-connectivity: neighbors north-west, north, north-east, west, east, south-west, south, and south-east on rectangular grid.
- 6-connectivity: hexagonal grid.

1.1.1 Operator Classification

Morphological operators are mappings from image(s) to image(s). They can be classified into three categories.

- *Point operators*: the output value of a pixel only depends on the input value of that pixel. Typical operators of this type are taking the complement, union, or intersection. These operators have a time-complexity which is linear in the number of pixels. Parallelization of point operations is trivial, since the domain of the image can be partitioned in as many (almost) equally sized subdomains as there are processors, and each processor can apply the operator to one of these subdomains without having to interact with any other processor. Thresholding is an example of this type of operator. Point operators are not discussed in the rest of this thesis.

- *Local operators*: the output value of a pixel depends on the input values in the neighborhood of that pixel. A typical operator of this type is a binary edge detector, which sets the output value of each foreground pixel (or 1-pixel) which is completely surrounded by 1-pixels to background (or 0-pixel), while 1-pixels which have a 0-pixel as neighbor remain 1-pixels (0-pixels remain 0-pixels). Local operators generally have a time-complexity which is linear in the number of pixels multiplied by the size of the neighborhood under consideration. These operators are harder to parallelize than point operators, but usually the parallelization strategy is the same. However, at the boundaries of the subdomains (partitions) processors do have to communicate and synchronize.

- *Global operators*: the output value of a pixel depends on the input values of many (possibly all) other pixels in the input image. Examples of this type of operator are connected component labeling (see chapter 3) and the watershed transform (see chapter 6). It is not possible to give a general time complexity for these operators. Global operators are hard to parallelize, and offer a real challenge for parallel program design. Most of the algorithms discussed in this thesis will be of this type.

1.1.2 Dilation and Erosion

Two basic building blocks for the construction of morphological operators are dilation and erosion. We denote the translation of a set X over a vector y by X_y, i.e. $X_y = \{x+y \mid x \in X\}$. *Minkowski addition* of two sets X and Y is defined as the piecewise vector sum of the elements of X and Y :

$$X \oplus Y = \{x+y \mid x \in X \wedge y \in Y\} = \bigcup_{x \in X} Y_x = \bigcup_{y \in Y} X_y$$

Minkowski subtraction of a set X by a set Y is defined as:

$$X \ominus Y = \bigcap_{y \in Y} X_{-y}$$

In mathematical morphology Minkowski addition and subtraction are called *dilation* and *erosion*, respectively. Although both operands X and Y are sets of the same type, it is common to look upon the first operand as the image on which the operation is applied, and the second operand is usually a much smaller set called the *structuring element*. Therefore, we prefer to use the following notation for dilation and erosion:

$$\delta_Y(X) = X \oplus Y, \quad \varepsilon_Y(X) = X \ominus Y$$

The reflection $\overset{\vee}{Y}$ of a set Y is defined as:

$$\overset{\vee}{Y} = \{-y \mid y \in Y\}$$

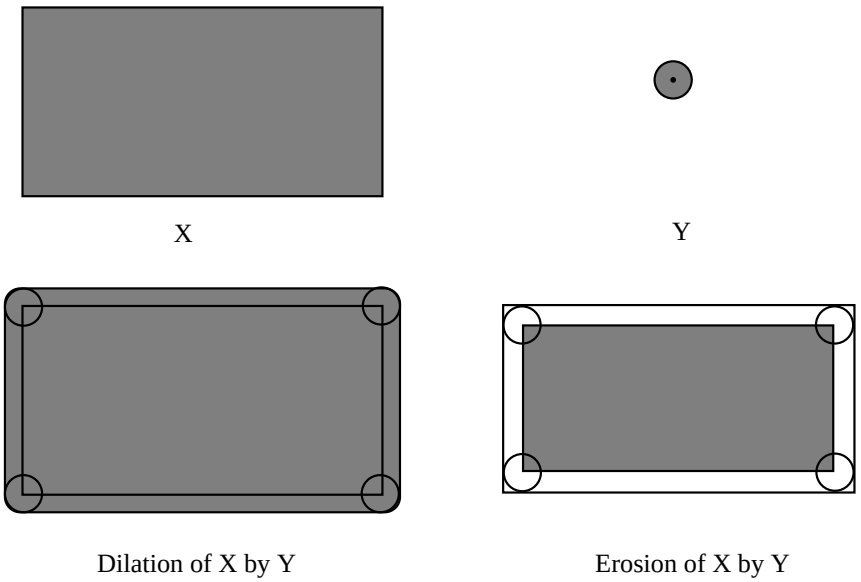


Figure 1.1. *The geometrical interpretation of dilation and erosion (in the continuous case).*

In order to avoid having to write many parentheses that make expressions hard to read, we introduce the notation

$$\overset{\vee}{Y}_h = (\overset{\vee}{Y})_h = \{h - y \,|\, y \in Y\}.$$

Dilation and erosion have a simple geometrical interpretation, which helps to understand their behavior (see figure 1.1):

$$\begin{aligned} \delta_Y(X) &= \{h \,|\, X \cap \overset{\vee}{Y}_h \neq \emptyset\} \\ \varepsilon_Y(X) &= \{h \,|\, Y_h \subseteq X\} \end{aligned}$$

In words, the dilation of X by Y is the set of points h such that the translation of the reflected set $\overset{\vee}{Y}$ over the vector h 'hits' the set X. Similarly, the erosion of X by Y is the set of points h such that after translating Y over the vector h it 'fits' in X.

Several algebraic properties of dilation and erosion exist. The most important properties are:

- Dilation and erosion are *increasing operators*, which means that if $X \subseteq Y$ we have $\delta_B(X) \subseteq \delta_B(Y)$, and $\varepsilon_B(X) \subseteq \varepsilon_B(Y)$ for any structuring element B.
- Erosion is *decreasing* in its second argument (the structuring element). If $B_1 \subseteq B_2$, then $\varepsilon_{B_2}(X) \subseteq \varepsilon_{B_1}(X)$, for every set X.
- Erosion and dilation form an *adjunction*, i.e. $\delta_B(A) \subseteq C \Leftrightarrow A \subseteq \varepsilon_B(C)$, for any sets A, B and C.
- Dilation is commutative: $\delta_B(A) = \delta_A(B)$, or equivalently $A \oplus B = B \oplus A$, for any sets A and B.

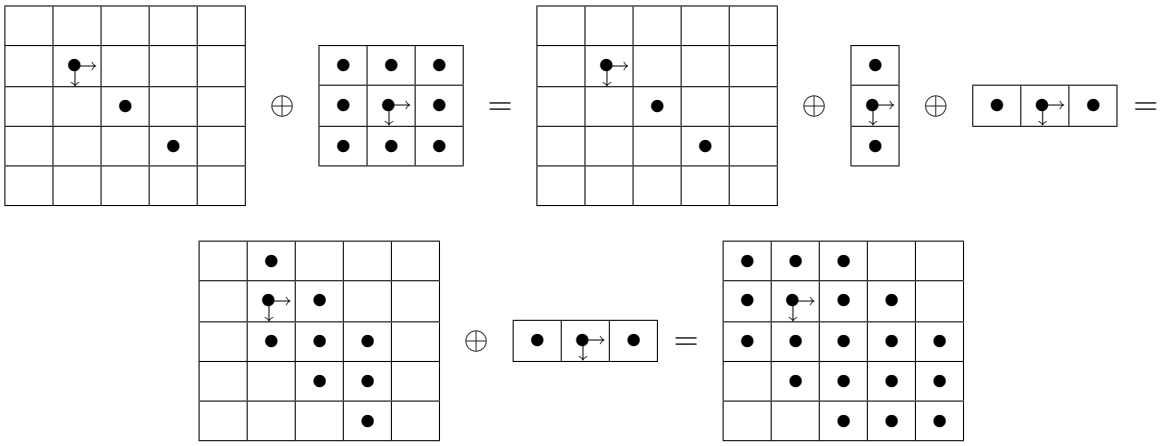


Figure 1.2. *Decomposition of the dilation by* $C_8 = \{(i, j) \mid -1 \leq i, j \leq 1\}$.

- Erosion is in general not commutative: $\varepsilon_B(A) \neq \varepsilon_A(B)$, or equivalently $A \ominus B \neq B \ominus A$.
- Dilation is associative: $\delta_{\delta_C(B)}(A) = \delta_C(\delta_B(A))$, or equivalently $A \oplus (B \oplus C) = (A \oplus B) \oplus C$, for any sets A, B, and C.
- Translation invariance: $\delta_B(A_x) = (\delta_B(A))_x$, and $\varepsilon_B(A_x) = (\varepsilon_B(A))_x$, or equivalently $A_x \oplus B = (A \oplus B)_x$, and $A_x \ominus B = (A \ominus B)_x$.
- Dilation and erosion are in a sense dual operators. By A^c we denote the complement of the set A (i.e. $a \in A^c \Leftrightarrow a \notin A$). Dilation of the foreground with a structuring element B is equivalent to taking the complement of the erosion of the background with the reflection of B, while erosion of the background is equivalent to the taking the complement of the dilation of the foreground with the reflection of B: $\delta_B(A) = (\varepsilon_{\check{B}}(A^c))^c$, and $\varepsilon_B(A) = (\delta_{\check{B}}(A^c))^c$.
- Dilation and erosion are not inverses of each other: $\delta_B(\varepsilon_B(A)) \neq A \neq \varepsilon_B(\delta_B(A))$.
- Dilation distributes over union: $\delta_B(\bigcup_{i=0}^n A_i) = \bigcup_{i=0}^n \delta_B(A_i)$.
- Erosion distributes over intersection: $\varepsilon_B(\bigcap_{i=0}^n A_i) = \bigcap_{i=0}^n \varepsilon_B(A_i)$.
- The erosion of a set A by the union of two sets B and C, is the same as the intersection of the erosion of A by B, and the erosion of A by C. This property can be generalized to $\varepsilon_{\cup_{i=0}^n B_i}(A) = \cap_{i=0}^n \varepsilon_{B_i}(A)$.
- Repeated erosion of a set A by sets $B_0, \ldots, B_n$, is the same as the erosion of A by the dilation of the sets $B_0, \ldots, B_n$: $\varepsilon_{B_0}(\varepsilon_{B_1}(\ldots(\varepsilon_{B_n}(A))\ldots)) = \varepsilon_{B_0 \oplus B_1 \oplus \ldots \oplus B_n}(A)$.

The last four properties are called decomposition theorems. These properties allow efficient parallel implementations of morphological operators. For example, the commonly used structuring element $C_8 = \{(i, j) \mid -1 \leq i, j \leq 1\}$, can be decomposed in $\{(0,-1),(0,0),(0,1)\} \oplus$

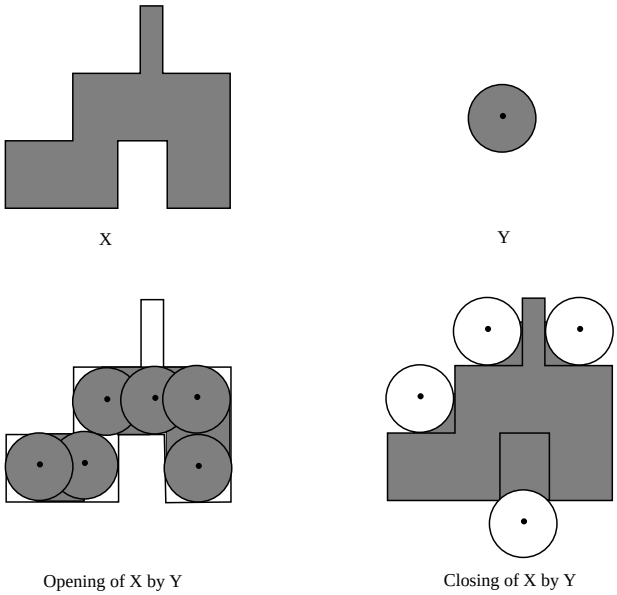


Figure 1.3. *The geometrical interpretation of opening and closing (in the continuous case). The objects are shown in grey.*

$\{(-1,0),(0,0),(1,0)\}$. An algorithm for dilating a set X directly by C_8 requires for each pixel access to all its 8-connected neighbors, which makes the algorithm not suitable for a parallel implementation. On the other hand, by decomposing the structuring element, we can implement the dilation with C_8 by two smaller dilations that can be performed in parallel. In figure 1.2 this decomposition theorem is depicted graphically. A square filled with the symbol '•' denotes a foreground pixel, while an empty square denotes a background pixel. The symbol '$\bullet\!\!\rightarrow\!\!\downarrow$' denotes the origin of the image, i.e. the pixel $(0,0)$. A parallel algorithm for dilation with C_8 is discussed in more detail in section 1.3.

The commutativity of dilation, combined with the distributivity of dilation over set union, can be used to decompose a structuring element in several smaller ones, and compute the smaller dilations in parallel on as many processors as the number of smaller structuring elements the original structuring element was decomposed in. The final result can be obtained by taking the union of the results, which can be performed in a number of steps which is logarithmic in the number of processors used.

The duality property is useful in the sense that only algorithms for dilation need to be developed, since the erosion is simply the dilation of the background. It is possible to develop algorithms for both erosion and dilation, using their specific properties, but generally this is hardly worthwhile, since inverting an image is a very fast operation.

1.1.3 Opening and Closing

We already know that erosion and dilation are not each other's inverse operations. This is easy to see if we take an object like a disc with a very small hole (let us say, only one pixel) in its interior, and we dilate the disc with a smaller disc. The result will be that the disc grows in

size, but also the hole is filled. After erosion this disc will shrink again, but the small hole has disappeared. For this reason, this sequence of operations is called a *structural closing*. The dual operator, erosion followed by dilation, is called a *structural opening*. The structural opening of a set X with a structuring element Y is denoted by $O_Y(X)$, while the corresponding closing is denoted by $C_Y(X)$. In the case of infix notation we use the symbols '$\circ$' and '$\bullet$', respectively.

$$\begin{aligned} O_Y(X) &= X \circ Y = \delta_Y(\varepsilon_Y(X)) \\ C_Y(X) &= X \bullet Y = \varepsilon_Y(\delta_Y(X)) \end{aligned}$$

Structural openings and closings, like erosion and dilation, have a geometrical interpretation:

$$O_Y(X) = \bigcup_{h, Y_h \subseteq X} Y_h, \quad C_Y(X) = \bigcap_{h, X \subseteq (\check{Y}^c)_h} (\check{Y}^c)_h$$

In words, the structural opening of X with structuring element Y is the union of translated sets Y_h that 'fit' within X. Similarly, the structural closing of X with structuring element Y is the intersection of translations of the reflection of Y in which X 'fits'. This interpretation is illustrated in Fig. 1.3.

Just like dilation and erosion, structural openings and closings are dual operators:

$$O_Y(X) = (C_{\check{Y}}(X^c))^c, \quad X \circ Y = (X^c \bullet \check{Y})^c$$

Structural opening and closing are examples of *algebraic openings and closings*, which are operators on images that satisfy the following 3 axioms:

1. *(Anti)extensive*: Opening is anti-extensive, i.e. by opening a set the result can only become smaller ($O(X) \subseteq X$). Closing is extensive, i.e. by closing a set the result can only become larger ($X \subseteq C(X)$).

2. *Idempotent*: Repeating an opening or a closing does not change the result, i.e. $O(O(X)) = O(X)$, and $C(C(X)) = C(X)$.

3. *Increasing*: If $X \subseteq Y$, then $O(X) \subseteq O(Y)$, and $C(X) \subseteq C(Y)$.

In chapter 4 algorithms for a special class, called *attribute openings and closings*, is discussed. One of these openings is called an area opening, which removes objects based upon size (i.e. number of pixels of an object), instead of using a structuring element.

1.1.4 Some Operators Based on Dilation and Erosion

Opening and closing are the most common operators which are solely based on dilations and erosions. In this section some other commonly used operators are reviewed.

The *hit-or-miss transform* is a transformation which is used for template matching. The transformation involves two template sets, B_1 and B_2, which are disjoint (see fig. 1.4). Template

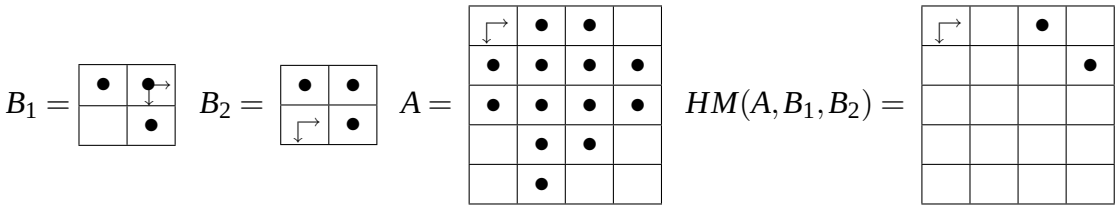


Figure 1.4. *Hit-or-miss transform.*

B_1 is used to match the foreground (objects), while B_2 is used to match the background of the image. The hit-or-miss transformation is the intersection of the erosion of the foreground with B_1 and the erosion of the background with B_2:

$$HM(A,B_1,B_2) = \varepsilon_{B_1}(A) \cap \varepsilon_{B_2}(A^c)$$

A *conditional dilation* of a set A by a structuring element B using a mask set M is a repetition of a dilation of the set A with the structuring element B followed by an intersection with the mask set M. The structuring element must contain the origin. This transformation can be used to eliminate small noise objects from a set A by applying an erosion with a structuring element which is large enough to remove the noise objects, without removing the objects of interest completely (see Fig. 1.5). The result of this operation is conditionally dilated using the original image as a mask set.

To define the conditional dilation, $\delta_{B;M}(A)$, of a set A by a structuring element B and a mask M, we first need to introduce the following recurrence:

$$\begin{aligned} A_0 &= A \\ A_n &= \delta_B(A_{n-1}) \cap M \end{aligned}$$

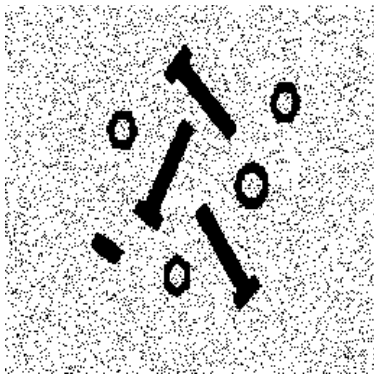
The conditional dilation $\delta_{B;M}(A)$ is now defined as $\delta_{B;M}(A) = \bigcup_{i=0}^{\infty} A_i$. For discrete sets, this means that $\delta_{B;M}(A) = A_k$, where k is the smallest integer such that $A_k = A_{k-1}$.

The *morphological edge detector* $\partial_B(A)$ of a set A with a symmetric convex structuring element B is defined as the set of pixels which are on the boundary of objects with respect to the structuring element B, i.e. $\partial_B(A) = A \backslash \varepsilon_B(A)$. The operator results in the *contours* of the object, which implies that $\partial_B(A) \subseteq A$. Several other edge detectors with different properties exist, e.g. $\partial'_B(A) = \delta_B(A) \backslash A$ so that $\partial'_B(A) \subseteq A^c$, or $\partial''_B(A) = \delta_B(A) \backslash \varepsilon_B(A)$.

1.1.5 Distance Transform

A distance transform maps a binary image to a grey-scale image. Therefore we first define the notion of grey-scale images.

A *grey-scale image* is a function $f : D \to R$, where $D \subseteq \mathbb{R}^n$, and $R \subseteq \mathbb{R}$ is a set of grey values. A *discrete (digital) grey-scale image* is a grey-scale image for which $D \subseteq \mathbb{Z}^n$ and $R \subseteq \mathbb{N}$, for which there is a neighboring relation defined on the set D, as in the case of binary images. From now on, we will omit the prefix *discrete* (or *digital*) and simply refer to grey-scale images. A



(a) noisy image

(b) filtered image using conditional dilation

Figure 1.5. *Noise removal using conditional dilation: let M be the noisy image, B a small disc, and* $A = \varepsilon_B(M)$. *The conditional dilation* $\delta_{B;M}(A)$ *is shown on the right.*

binary image can be looked upon as a grey-scale image for which $R = \{0,1\}$, although it is more common in the morphology literature to look upon binary images as sets of object pixels.

Let D be a set of pixels, and let $G = (D,E)$ (with $E \subseteq D \times D$) be the corresponding neighborhood graph (grid). Two pixels p and q are said to be neighboring pixels if and only if $(p,q) \in E$. A *path* π of length l from $p \in D$ to a point $q \in D$ is a sequence of pixels $\pi = (p_0 = p, p_1, \ldots, p_l = q)$, such that $(p_i, p_{i+1}) \in E$ for all $i \in \{0, \ldots, l-1\}$. The set of all paths from p to q is denoted by $[p \rightsquigarrow q]$, and the length of a path π is denoted by $l(\pi)$. We define $\pi = (p)$, to be the path of length 0 from p to itself.

The *(discrete) distance* $d(p,q)$ between the points p and q is defined as the minimum of the lengths of all paths from p to q, i.e.

$$d(p,q) = \min_{\pi \in [p \rightsquigarrow q]} l(\pi).$$

With some overloading of notation, the distance between a point p and a set $A \subseteq D$ is defined as

$$d(p,A) = \min_{a \in A} d(p,a).$$

In most practical cases, the grid G is simply the 4- or 8-connected grid. In the case of the 4-connected grid, the corresponding distance is called the *city-block* or *Manhattan* distance, in the case of the 8-connected grid it is called the *chess-board distance*.

Let $A \subseteq D$ be a binary image. The *distance transform* of A is the mapping $DT : D \to \mathbb{N}$ which associates to each point $p \in D$ the distance of p to the background of A (i.e. A^c),

$$DT(p) = d(p, A^c)$$

Notice that the resulting image is a grey-value image DT, for which $DT(p) = 0$, for all background pixels p. An efficient algorithm to compute any of these discrete distance transforms, as well as the (continuous) Euclidean distance transform, is discussed in chapter 2.

1.1.6 Skeletons

A skeleton is a line representation of an object. In the literature there is a large number of (different) definitions of a skeleton. A skeleton could be described as a reduced description of an object, that has some preferred properties. These properties usually contradict each other, resulting in different definitions depending on the properties selected.

Informally, one could say that a skeleton should have the following properties:

- a skeleton ought to be one pixel thick,
- a skeleton should pass through the "middle" of the object, and
- a skeleton must preserve the topology of the object.

It is clear that these requirements cannot always be met. For example, a discrete set which contains a line which is two pixels thick can not have a skeleton which passes through the middle of the object, since pixels have integer coordinates.

A simple method to obtain the skeleton of a binary object, is to compute the crest lines of its distance transform. In the continuous case, i.e. the image domain is $\mathbb{R}^2$, using the Euclidean metric this yields a connected skeleton. In the discrete case, however, this method tends to create very disconnected skeletons.

Lantuéjoul (cf. [47]) introduced an iterative scheme to construct a skeleton of a discrete set $A \subseteq D$. Let $\oplus_n B$ denote the iterated dilation of a set B, i.e. $\oplus_0 B = \{0\}$, and $\oplus_n B = (\oplus_{n-1} B) \oplus B$.

The k-th *skeleton subset* $S_{B,k}(A)$ of a set A using a structuring element B is defined as

$$S_{B,k}(A) = \varepsilon_{\oplus_k B}(A) \backslash O_B(\varepsilon_{\oplus_k B}(A)) = (A \ominus (\oplus_k B)) \backslash ((A \ominus (\oplus_k B)) \circ B), \quad k \in \mathbb{N}.$$

The structuring element B must be chosen such that it approximates a circular disc, i.e. it must be convex, bounded and symmetric. Let K be the smallest integer k such that $S_{B,k} = \emptyset$. The *morphological skeleton* $S_B(A)$ of the set A using the structuring element B is defined as:

$$S_B(A) = \bigcup_{k=0}^{K} S_{B,k}(A)$$

When all the skeleton subsets are known, the original set A can be reconstructed using

$$A = \bigcup_{k=0}^{K} S_{B,k}(A) \oplus (\oplus_k B)$$

A compact representation of the skeleton subsets can be obtained using a grey-scale image $sk : D \to \mathbb{N}$, defined as

$$sk(p) = \begin{cases} k+1 & if \quad p \in S_{B,k}(A) \\ 0 & otherwise \end{cases}$$

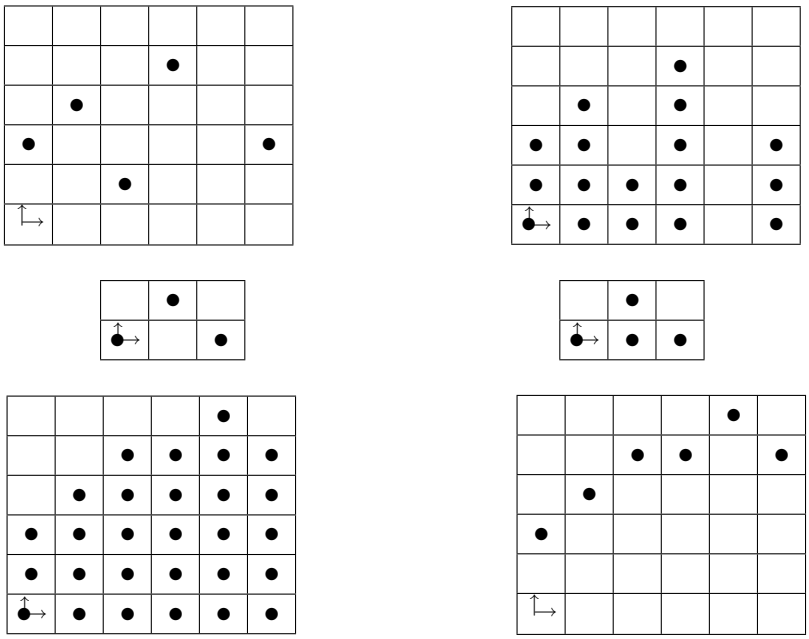


Figure 1.6. *Top: a one-dimensional function f (left), and its umbra $U(f)$ (right). Middle: a structuring element g (left), and its umbra $U(g)$ (right). Bottom: binary dilation $U(f) \oplus U(g)$ (left), and the grey-value dilation $T(U(f) \oplus U(g))$ (right).*

Note that the skeleton sets are disjoint, therefore *sk* is a proper function. A useful property of the skeleton sets $S_{B,k}(A)$ is that we can obtain the opening of A with structuring element $\oplus_n B$ by partial reconstruction, by omitting the first n skeleton sets, i.e.

$$O_{\oplus_n B}(A) = \bigcup_{k=n}^{K} S_{B,k}(A) \oplus (\oplus_k B)$$

1.1.7 Grey Scale Morphology

Thus far, all operators took their arguments from the set of binary images. In this subsection, the domain of the operators is extended to allow grey-scale images as well. The approach followed is to transform grey-scale images into sets, apply the operators from binary morphology and transform the result back into a grey-scale image. This process is purely of theoretical value, since in practice grey-scale morphological operators are implemented directly, and not via this indirect process. Besides, in the case of grey-scale images with a continuous domain and range, this approach invokes mathematical difficulties.

Let $f : D \to R$ be a grey-scale image. The *umbra set* $U(f)$ (umbra is Latin for shadow) of the function f is defined as

$$U(f) = \{(p,r) \in D \times R \,|\, r \leq f(p)\}$$

A set $U \subseteq D \times R$ is called an umbra if and only if $\forall((p,r) \in U, r' \in R : r' \leq r \Rightarrow (p,r') \in U)$.

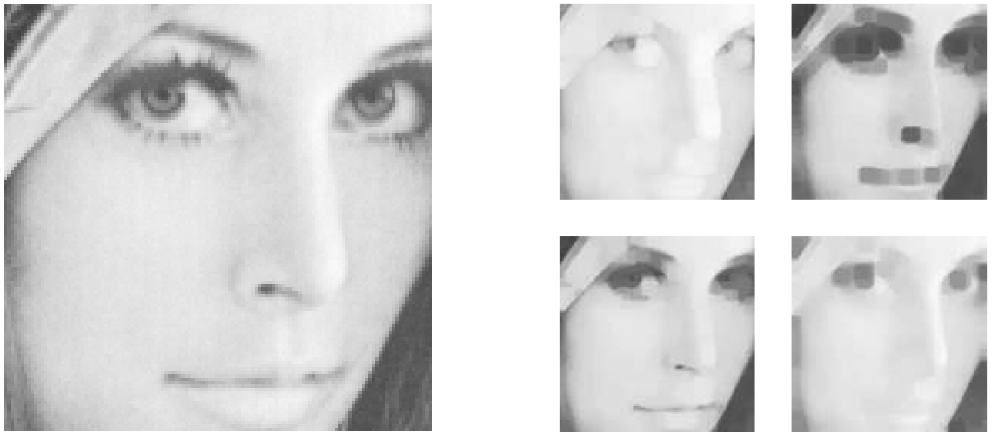


Figure 1.7. *Left: grey scale image. Right: upper left: grey scale dilation; upper right: grey scale erosion; lower left: grey scale opening; lower right: grey scale closing.*

The *top* function is the inverse of the umbra function. For an umbra $U \subseteq D \times R$, the top function $T : D \times R \to (D \to R)$ is defined as

$$(T(U))(p) = \max_{(p,r)\in U} r$$

Using the top and umbra function we can define grey value dilation and erosion. First we associate an umbra to the input image and the structuring element. Then we apply the binary dilation (or erosion) on the umbra. The result again will be an umbra, on which we apply the top function to transform the result back into a grey-scale image.

In order to avoid confusion with the binary operators, we use the symbols $\overline{\oplus}$ and $\overline{\ominus}$ for the grey-value dilation and erosion respectively. The grey-value dilation and erosion are defined as:

$$\begin{aligned} f\overline{\oplus}g &= T(U(f)\oplus U(g)) \\ f\overline{\ominus}g &= T(U(f)\ominus U(g)) \end{aligned}$$

In practice, the umbra and top functions are not used. These functions are only useful to show that properties of the binary operators are inherited by the grey-scale equivalents. For the grey-scale dilation and erosion of two functions $f, g : D \to R$, one can show that

$$\begin{aligned} (f\overline{\oplus}g)(p) &= \max\{f(p-d)+g(d)\,|\,d \in D,\, p-d \in D\}, \\ (f\overline{\ominus}g)(p) &= \min\{f(p+d)-g(d)\,|\,d \in D,\, p+d \in D\}. \end{aligned}$$

The *reflection* $\overset{\vee}{f}$ of a grey-value function f is defined as $\overset{\vee}{f}(p) = -f(-p)$. Using the umbra, top, and reflection operator, all properties of binary dilation and erosion are inherited by the grey-scale dilation and erosion. In the same way as in the binary case, other grey-scale operators like opening and closing are built upon the grey-scale erosion and dilation. The operators set-intersection ($\cap$) and set-union ($\cup$) have been replaced by the operators min and max respectively.

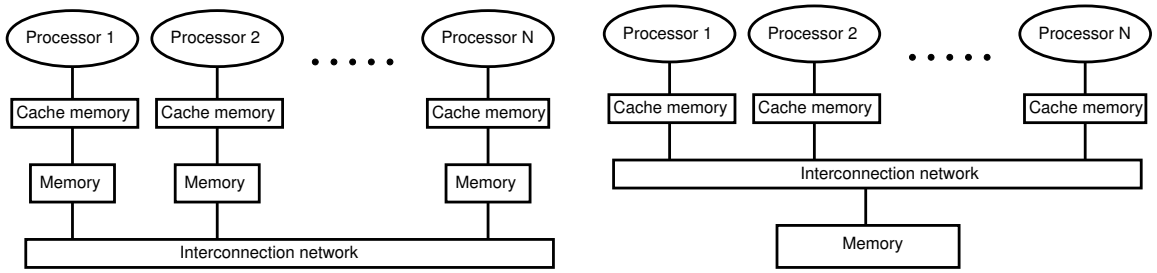


Figure 1.8. *Left: distributed memory architecture. Right: shared memory architecture.*

For example, the grey-scale opening ($\overline{\circ}$) and closing ($\overline{\bullet}$) are defined as (see Fig. 1.7):

$$\begin{aligned} f\overline{\circ}g &= (f\overline{\ominus}g)\overline{\oplus}g \\ f\overline{\bullet}g &= (f\overline{\oplus}g)\overline{\ominus}g \end{aligned}$$

Since most properties of binary operators are inherited by the corresponding grey-scale operator, there is no need to use explicitly a different symbol for grey-scale morphological operators. From the type of the arguments of the operator, it is clear whether we mean an operator which is applied to binary arguments, or it is an operator which is applied to grey-scale arguments.

1.2 Introduction to Parallel Computing

The evident goal of the use of parallel computers is to speed up computations by using multiple CPUs, or to perform larger computations which are not possible on a single processor machine. In general we can divide parallel computing architectures in two main classes with respect to memory layout. The first is the class of *distributed memory machines*. The second class is that of *shared memory machines* (see fig. 1.8).

1.2.1 Distributed Memory Machines

In distributed memory machines, each processor has its own local memory and memory cache. All variables of a program are *private*, i.e. they live in the local memory (or cache) of a processor. If one processor requires data contained in another processor's memory, messages must be passed between them through some interconnection network. This network can be a LAN (Local Area Network) in the case of a cluster of workstations, as well as a dedicated network inside a massively parallel system. For communication between processors a message passing library is used. Two commonly used libraries are MPI (Message Passing Interface, see [34, 70]) or PVM (Parallel Virtual Machine, see [77]). Typically, a single program is run on each processor. Each process owns a private integer variable, which designates its process number. Using this identification, a process can decide which actions to take. For example, if a process has identification number 0, it may behave as a master process, while all other processes will act as a slave process. The programming model in which each processor runs the same program and acts on its own local memory is sometimes called *SPMD*, which is an abbreviation for *Single Program Multiple*

Data. Although some message passing programs are discussed in this thesis, the main focus is on shared memory architectures.

1.2.2 Shared Memory Architectures

In *shared memory machines*, every processor has access to all of the memory, i.e. , there is a shared address space. In this configuration, it must be assured that processors do not simultaneously access regions of memory in such a way that errors would occur.

Symmetric Multiprocessing (SMP) refers to the paradigm used in programming shared memory systems. In SMP-programs all processes have full and equal functionality. The term SMP is often used synonymously with ”shared memory”.

Concurrency is the word used to describe independence between a number of actions, such as the execution of a number of instructions simultaneously. Usually in this context a process is called a *thread* or a *light weight process*. It is light weight in the sense that it shares its dataspace as well as its program code with the process that created it. Therefore an operating system (like UNIX) does not have to create (or replicate) a data space for a thread at initialization, since it is inherited from the process that created the thread. A commonly used function library that provides the programmer with tools to create and synchronize threads is the POSIX pthreads library (see [20, 41, 43]).

With a program composed of many concurrent threads, we lose the convenience offered by the determinism of sequential programs. The result of the following program fragment in which two threads P and Q run concurrently cannot be simply deduced from reading it. Variables which are declared outside the scope of a thread are *shared variables*, while variables declared within the scope of a thread are *private variables*. All threads can access all shared variables, while each thread can only access its own private variables.

var x : *integer;* $(*$ *initially* $x = 1$ $*)$

thread *P;*
  var y : *integer;*
  $y := x; y := y + 1; x := y;$
end;

thread *Q;*
  var y : *integer;*
  $y := x; y := 3 * y; x := y;$
end;

Let us assume that each assignment is an *atomic* (or *indivisible*) operation, i.e. it does not interfere with any other statement. This way, a concurrent execution of the program can be regarded as a sequential interleaving of the statements. If we label the three assignments of thread P with a, b, and c, and those of thread Q with d, e, and f, then each interleaving of these letters in which a precedes b, b precedes c, d precedes e, and e precedes f, is a possible execution of the

program. Such a possible execution is called a *trace*. The number of possible traces increases very rapidly with the number of statements of a program. This simple example consisting of six assignments gives rise to no less than 20 traces, yielding four possible outcomes. After execution of the program the value of the shared variable x can be 2, 3, 4, or 6, depending on the order of execution of each thread, as can be seen in the following table:

trace	x	trace	x	trace	x	trace	x	trace	x
abcdef	6	adbcef	3	dabecf	3	abdefc	2	adefbc	2
defabc	4	adbecf	3	daebcf	3	daefbc	2	adbefc	2
abdecf	3	adebcf	3	deabcf	3	dabefc	2	adebfc	2
abdcef	3	dabcef	3	deabfc	2	daebfc	2	deafbc	2

Besides the fact that assignments are in practice not atomic actions, it is clearly not feasible to verify all possible traces of larger concurrent programs. Fortunately, primitives are available that allow to make regions of code atomic. These regions are usually called *critical sections*. These sections are areas of code that access shared variables, and must therefore be performed under *mutual exclusion*, i.e. no two threads are allowed to be in the same critical section simultaneously.

1.2.2.1 Mutexes

Critical sections can be made using a synchronization primitive, called a *mutex*. A mutex serializes the execution of a program. It can be regarded as a shared integer variable, which is either -1, or a positive number indicating a thread number. Mutexes have two basic operations, **lock** and **unlock**. A mutex is unlocked if it has the value -1. If a thread calls **lock** on an unlocked mutex, the mutex locks (is set to the identification number of the thread) and the thread continues. If however the mutex is locked, the thread blocks until the thread 'owning' the lock calls **unlock**. It is regarded an error if a thread that does not own a mutex performs an **unlock** on it. The operations **lock** and **unlock** are atomic operations on mutexes, and are provided by the operating system.

1.2.2.2 Condition Variables

Besides a primitive for guarding critical sections, we usually also need some mechanism to suspend execution until some predicate holds. For example, it might be necessary to suspend the execution of a thread that tries to retrieve data from an empty queue, until some other thread inserts data and signals it to continue. This type of synchronization can be implemented by means of *condition variables*.

Two operations are allowed on a condition variable: **wait** and **signal**. The **wait** operation suspends the calling thread, while a **signal** operation wakes a thread which is suspended on the condition variable. A condition variable is associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it. A thread that owns the mutex (i.e. it has locked the mutex), can decide upon inspection of shared data that it cannot continue. In this case, it suspends itself by calling **wait(**c, m**)**, where c denotes the condition variable, and m the associated mutex.

```
procedure barrier (self : integer; N : integer);
    lock (mut);
      cnt := cnt+1;
      if cnt = N then
        cnt := 0;  (* reset counter for re-use *)
        forall t ∈ [0..N)\{self} do
          signal(cv[t]);
        end forall;
      else
        wait(cv[self], mut);
      end if;
    unlock (mut);
end;
```

Figure 1.9. *Implementation of a barrier using condition variables. The number of threads is denoted by N, while self denotes the identification number of each thread (*$0 \leq self < N$*).*

A thread is only allowed to call **wait** on a condition variable when it owns the associated mutex. Upon suspension of a thread, the mutex *m* is unlocked. Unlocking the mutex and suspending on the condition variable is done atomically. This guarantees that the condition cannot be signaled between unlocking the mutex and starting to wait on the condition variable.

Any other thread can wake up the suspended thread by calling **signal(*c*)**. This is typically the case if the waking thread modified shared data, like inserting data in a shared queue. A thread that is signalled, atomically (re)locks the associated mutex, and can resume execution. When two or more threads are suspended on a condition variable, only one thread is woken up. When no threads are suspended on the condition variable, the signal is ignored.

The POSIX implementation of condition variables allow a waiting thread to wake up without any thread signaling it. These cases are supposed to be rare, and are called *spurious wake-ups*. Spurious wake-ups are allowed for reasons of efficiency. Building a fully safe signal mechanism is argued to be too expensive in practice, while signaled threads should verify if the condition they were waiting for still holds once they are woken up anyway. If the condition does not hold, the thread should reenter a wait state. In practice this means that after each **wait** a thread should retest, and wait again if necessary. In the remainder of this thesis we assume that the combination of **wait** and **signal** does not suffer from spurious wake-ups. The reader should however realize that in actual implementations of the pseudo-codes presented here retesting is necessary.

1.2.2.3 Barriers

Using mutexes and condition variables it is easy to build another very useful synchronization primitive, called a *barrier*. A barrier is a so-called collective operation, i.e. all threads take part in it. A barrier is a routine which gets called by all threads. Each thread, except the last, gets suspended. When the last thread arrives, all other threads are released. A barrier is typically a

useful primitive in algorithms that consist of a sequence of stages. Each stage can be executed concurrently, however a next stage should not be started until all threads completed the current one. A typical example of a situation where a barrier can be useful is found in chapter 2, where an algorithm for distance transforms is discussed. The algorithm consists of two phases, that need to be separated by a barrier.

A barrier can be implemented by means of a shared variable *cnt* that acts as a counter, which reflects the number of threads that are suspended on the barrier. Hence, *cnt* is initialized as 0. The counter is protected by a mutex *mut*. A thread that reaches the barrier locks the mutex, and increments *cnt*. Let N be the number of threads. If *cnt* has not reached N the thread suspends itself by waiting on a condition variable. For this purpose, we introduce an array $cv[0..N)$ of condition variables. Thread i can suspend itself by calling **wait(**$cv[i], mut$**)**. The last thread that arrives at the barrier detects that (after increment) *cnt* has reached N, and wakes all other threads by signaling their condition variables. Figure 1.9 shows an implementation of this procedure.

1.2.2.4 Semaphores

Another useful synchronization primitive is the *semaphore*. A semaphore is a shared integer variable that is set upon initialization. After that, it can never be accessed directly. However, there are two operations to increment or decrement the value of the semaphore by one. Decrementing is a (possibly) blocking function. If the resulting semaphore value is negative, the calling thread is blocked, and cannot continue until some other thread increments it. Incrementing the semaphore when it is negative causes one (and only one) of the threads blocked by this semaphore to become unblocked and runnable. Semaphores were invented by the Dutch professor Edsger W. Dijkstra (see [28]). He dubbed the two operations P and V. These names come from the Dutch words 'Proberen' (to test) and 'Verhogen' (to increment).

Semaphores can easily be implemented by means of mutexes and condition variables. We implement a semaphore by means of a structure consisting of an integer variable *val*, a mutex *mut*, and a condition variable *cv*. The operation *P(s)* decrements *s.val*. When *s.val* has becomes less than zero, the thread that called $P(s)$ will get blocked by waiting on the condition variable, and will remain so until another thread unblocks it by signaling the condition variable. This is all done atomically. On the other hand, $V(s)$ increments *s.val*, and if this is less than or equal to zero, then there is at least one other thread that is blocked on s. Exactly one of these threads gets unblocked by signaling the condition variable *cv*. This implementation of semaphores and their corresponding operations by means of mutexes and condition variables is shown in figure 1.10.

There is a variant of the semaphore called a *binary semaphore*. A binary semaphore is much like a normal semaphore except that the integer can only assume the values of 0 and 1. They are usually implemented so that threads attempting to lock a semaphore whose value is zero simply block until the value is 1, then they unblock and set it to zero. A binary semaphore resembles a mutex, however it can be unlocked by any thread. Just like signaling a condition variable on which no threads are suspended, a V-operation is ignored if the value of the semaphore is 1.

```
type semaphore = record
                   val : integer;
                   mut : mutex;   (* initially unlocked *)
                   cv : condition variable;
                 end;

procedure initSemaphore(val : integer; var s : semaphore);
    s.val := val;
end;

procedure P(var s : semaphore);
    lock (s.mut);
    s.val := s.val - 1;
    if s.val < 0 then wait(s.cv, s.mut);
    unlock (s.mut);
end;

procedure V(var s : semaphore);
    lock (s.mut);
    s.val := s.val + 1;
    if s.val <= 0 then signal(s.cv);
    unlock (s.mut);
end;
```

Figure 1.10. *Implementation of semaphores and the corresponding operations P and V.*

1.2.3 Speedup and Efficiency

The quality of an algorithm is hard to define. It is evident that an algorithm must be *correct*, i.e. it meets its specification. Besides this requirement, one can come up with several other requirements, like efficiency, portability, scalability, elegance, and modularity. In this thesis we are mainly concerned with the correctness and efficiency of parallel algorithms. An important measure for the quality of a parallel program is its *speedup*. The speedup of an algorithm using p processors is the ratio of the time needed to run the algorithm on one processor (T_1) and the time needed to run the algorithm using p processors (T_p), i.e.

$$S_p = T_1/T_p.$$

Note that the times T_i are so called *wall clock* times in seconds. Wall clock time is simply the execution time of the program measured using e.g. a stopwatch. Note that this differs from what is generally called *cpu-time*, which is the accumulation of execution time over all processors. A concurrent algorithm that keeps 2 CPUs active during execution has a cpu-time time that is twice as large as its wall clock time.

Ideally, the speedup obtained by using p CPUs is p. This is called *linear speedup*. In practice, linear speedup is hardly ever reached, since parallel programs spend time on synchronization, communication, or contain sequential sections.

Another frequently used measure of performance is the *efficiency* of a parallel program. The efficiency of a program yields the same information as the speedup of a program, since it is computed as

$$E_p = (S_p/p) \times 100\%.$$

1.3 An Example: Concurrent dilation

In this section a simple example of a concurrent algorithm is presented. Its main purpose is to get a better understanding of the concepts introduced in this chapter. The example chosen is an algorithm for dilation of a binary image with the structuring element $C_8 = \{(i,j) \mid -1 \leq i,j \leq 1\}$ (see sect. 1.1.2, fig. 1.2). We represent the binary image by a shared array of integers, of which the values are only 0 or 1.

var *im*[0..*M*-1,0..*N*-1] **of** *integer*;

Here, M and N denote the width and height of the image, respectively. A parallel algorithm that addresses for each foreground pixel (i,j) all its 8-connected neighbors, and sets them to foreground as well, requires a second output image, where a mutex is necessary to protect each pixel to be set by two or more threads at the same time. Such a direct implementation would yield a fully serialized program where all threads are competing for the mutex. The execution time would at best be the same as execution on a single CPU, but is likely to be much worse.

A better approach is to use the decomposition theorem from section 1.1.2, which states that

$$X \oplus \{(i,j) \mid -1 \leq i,j \leq 1\} = X \oplus \{(i,0) \mid -1 \leq i \leq 1\} \oplus \{(0,j) \mid -1 \leq j \leq 1\}.$$

In words, we can first perform a dilation in a row-wise fashion, followed by a dilation which is performed column-wise. In both stages the dilation per row (column) is independent of the dilation of any other row (column). This is the key to an effective concurrent algorithm. We introduce two shared integer variables *row* and *col* which are both initialized to 0. They denote (in the corresponding phases), which row or column to process next. A mutex *mut* is introduced to protect these two variables. In the first phase (row-wise dilation), each thread tries to lock the mutex. When it succeeds, it copies the current value of *row* in a private (i.e. non-shared) variable *r*, increments *row*, and unlocks the mutex. After this, the thread inspects the value of the private variable *r*. If $r < M$ the thread performs a horizontal dilation of row *r*, after which it will request for a new row to process. If $r \geq M$ all rows have been allotted to threads, and thus the thread waits until all other threads have finished processing their last row, by calling a barrier. When all threads are released from the barrier, a similar procedure for the columns is started. This analysis yields the concurrent algorithm presented in fig. 1.11.

Just like in the naive algorithm a mutex is used. However this mutex is not used to guard accesses to the pixels of the image, but guards the variables *row* and *col*. These variables are

```
var row, col;  (* shared, initially 0 *);

function getTask (var s : intger) : integer;
var t : integer;
    lock (mut);
    t := s;
    s := s + 1;
    unlock (mut);
    return t;
end;

procedure dilate8×8 (var im : array [0..M-1,0..N-1] of integer);
var r, c : integer;
    (* horizontal phase *)
    r := getTask (row);
    while r < M do
        for c := 0 to N-2 do
          if im[r,c] = 0 ∧ im[r,c+1] = 1 then im[r,c] = 1;
        endfor;
        for c := N-1 downto 1 do
          if im[r,c] = 0 ∧ im[r,c-1] = 1 then im[r,c] = 1;
        endfor;
        r := getTask (row);
    endwhile;
    (* all threads need to finish before we start a vertical phase *)
    barrier (self, nthreads);
    (* vertical phase *)
    c := getTask (column);
    while c < N do
        for r := 0 to M-2 do
          if im[r,c] = 0 ∧ im[r+1,c] = 1 then im[r,c] = 1;
        endfor;
        for r := M-1 downto 1 do
          if im[r,c] = 0 ∧ im[r-1,c] = 1 then im[r,c] = 1;
        endfor;
        c := getTask (column);
    endwhile;
(* wait for all threads to finish *)
barrier (self, nthreads);
end;
```

Figure 1.11. *Code fragment of a concurrent algorithm implementing a binary dilation with structuring element* $C_8 = \{(i, j) | -1 \leq i, j \leq 1\}$.

much less frequently accessed, so the chance of several threads accessing the same mutex at the same time has reduced significantly. Besides, the critical section guarded by the mutex is very small. It consists of only two assignments, while processing of 8 neighboring pixels in a critical section takes more time. The smaller a critical section, the less time a thread spends in it. Hence, smaller critical sections lead to a smaller chance on contention.

Note that the decision which thread should process which row (column) is decided at run-time, i.e. the data distribution is *dynamic*. We could easily do without the mutex, and use a fixed distribution. Let T be the number of threads. We could decide that thread i ($0 \leq i < T$) processes rows from the interval $[i * \frac{M}{T}, (i+1) * \frac{M}{T})$, and columns from the interval $[i * \frac{N}{T}, (i+1) * \frac{N}{T})$. This distribution is called *static*. Using a static distribution we do not need the mutex, but still need the two barriers. Both distributions have their advantages. The static distribution does not need the mutex, and thus contention on it is not possible. However, if the amount of work per domain is not about equal, *load imbalance* might occur, and thus the program is slowed down by the thread taking the longest time to finish its task. This load imbalance is automatically solved in the case of a dynamic distribution, at the price of possible contention on the mutex. Especially if the amount of work per task gets larger, dynamic distribution pays off.

In this particular example, the amount of work per row (column) is linear in the number of pixels, and hardly depends on the content of the image. Therefore, in this case a static distribution would suffice, but for demonstration purposes a dynamic distribution was chosen.

1.4 Thesis organization

This thesis deals with algorithms for morphological image processing. The focus is on design of efficient sequential algorithms, and their parallelization for concurrent execution on shared memory architectures. The chapters 2, 3, 4, and 6, consist of published refereed papers ([40, 54, 57, 80]). I have decided to leave them as they are, even though this results in some overlap between the chapters. These papers have been published in cooperation with J.B.T.M. Roerdink, W.H. Hesselink, M.H.F. Wilkinson, and C. Bron.

Chapter 2 discusses a general algorithm for computing distance transforms in linear time. The algorithm is general in the sense that several metrics can be used. The metrics discussed in this chapter are the Manhattan metric, the chess board metric, and the Euclidean metric. The algorithm consists of two phases, each of which consists of two scans. In the first phase the pixels are processed column-wise, while in the second phase pixels are processed row-wise. The computation per row (column) is independent on the computation in any other row (column), yielding a very efficient parallel algorithm that runs in linear time.

Chapter 3 discusses the design of a parallel algorithm for the determination of the connected components of an image. Usually, connected components are determined using a region-growing algorithm that uses a FIFO-queue. Such region-growing algorithms are hard to parallelize. Therefore, an algorithm based on Tarjan's UNION-FIND algorithm (cf. [88]) is used, which is easier to parallelize. After presentation of the sequential algorithm, it is distributed using the message passing paradigm. Next, the message passing is eliminated and replaced by shared memory constructs. Especially for the processing of large images and 3D data sets, the parallel

algorithm shows nearly linear speedup.

In chapter 4 a comparison is made between three algorithms for connected set opening and closing. One of the simplest examples of this class of operators is the area opening, which is the example used throughout the chapter. An area opening removes objects which have an area smaller than a filter parameter λ. The grey value of each pixel p is lowered to the highest value h such that the connected component of the binary image obtained by thresholding at level h has an area at least λ. Two known algorithms from the literature and a new algorithm are compared. The first is a priority queue based algorithm published by Vincent (cf. [98,99]). The second algorithm published by Salembier et al. (cf. [83]) uses a hierarchical queue to build a tree structure called a Max-tree. Once this tree is constructed, filtering can be performed on the tree structure. The third algorithm by Meijster and Wilkinson (cf. [57]) uses a variation of Tarjan's UNION-FIND algorithm, in which connected components are represented using disjoint sets, while attributes (like area) are maintained on the fly. From experiments we can conclude that the latter algorithm is the fastest in the case we need to perform an opening (or closing) with a given λ. However, if we wish to filter the dataset using several values for the parameter λ, the Max-tree approach is more suitable, since it allows to compute a Max-tree only once, while the filtering can be performed as many times as we wish. Besides, the filtering phase takes much less time than the construction of the tree, yielding the possibility to do interactive filtering and visualization. This is the topic discussed in chapter 5.

In chapter 6 a critical review of several definitions of the watershed transform is presented. None of the definitions turn out to be equivalent with any of the other definitions. Several implementations of sequential and parallel watershed algorithms are discussed, including the ones proposed by Meijster and Roerdink (cf. [51–53]). Several authors have published algorithms for which they claim that they implement one of the definitions, but some of these algorithms turn out to be not fully correct. This is usually due to scanning order, i.e. the order in which pixels are processed. For example, rotating and image by 180 degrees, computing its watershed, and rotate the result back, does not always yield the same result as computing the watershed of the image directly, as it ought to be. This scanning order problem is even more severe if we try to implement concurrent algorithms for the watershed transform. Besides, the watershed is a so called global operation, i.e. the output value of a pixel depends on the input values of several (possibly all) pixels in the input image, which makes concurrent computation even harder. Therefore, several authors (including ourselves) have introduced new watershed definitions that can be computed in parallel.

Chapter 7 gives some concluding remarks and a summary of the contents of this thesis.

This chapter has been published as: A. Meijster, J. B. T. M. Roerdink and W. H. Hesselink, A general algorithm for computing distance transforms in linear time. In: *Mathematical Morphology and its Applications to Image and Signal Processing*, Kluwer Acad. Publ., 2000, pp. 331-340.

Chapter 2

A General Algorithm for Computing Distance Transforms in Linear Time

Abstract

A general algorithm for computing distance transforms of digital images is presented. The algorithm consists of two phases. Both phases consist of two scans, a forward and a backward scan. The first phase scans the image column-wise, while the second phase scans the image row-wise. Since the computation per row (column) is independent of the computation of other rows (columns), the algorithm can be easily parallelized on shared memory computers. The algorithm can be used for the computation of the exact Euclidean, Manhattan (L_1 norm), and chessboard distance (L_∞ norm) transforms.

2.1 Introduction

Distance transforms play an important role in many morphological image processing applications. They have been extensively studied and used in computational geometry, image processing, computer graphics and pattern recognition, e.g., [17, 18, 25, 81]. The two-dimensional distance transform can be described as follows. Let B be a set of grid points taken from a rectangular grid of size $m \times n$. The problem is to assign to every grid point (x,y) the distance to the nearest point in B. If we use the Euclidean metric for computing distances, and represent B by a boolean array $b[\cdot,\cdot]$, we thus want to compute the two dimensional array $dt[x,y] = \sqrt{EDT(x,y)}$, where

$$EDT(x,y) = \texttt{MIN}(i,j : 0 \leq i < m \wedge 0 \leq j < n \wedge b[i,j] : (x-i)^2 + (y-j)^2).$$

Here we use the notation $\texttt{MIN}(k : P(k) : f(k))$ for the minimal value of $f(k)$ where k ranges over all values that satisfy $P(k)$.

Since the exact Euclidean distance transform is often regarded as too computationally intensive, several algorithms have been proposed that use some mask which is swept over the image in two scans, to compute approximations like the Manhattan (city-block) distance, the chessboard distance, or chamfer distances (see [17, 18, 25, 81]). The time complexity is linear in the number of pixels of the image (i.e. $O(m \times n)$), but it does not yield the exact Euclidean distance, which

is required for some applications. Another drawback of these algorithms is that they are hard to parallelize for execution on parallel computers since previously computed results are propagated during the computation, making the process highly sequential. A recursive algorithm of order $mn \log m$ for the exact *EDT* is given in [45]. In [75] a recursive algorithm of order mn for the exact *EDT* is given by reducing the problem to a matrix search algorithm.

In this chapter, which is based upon [54], we present an algorithm that also computes distance transforms in linear time, is simpler and more efficient than [75], and is easy to parallelize. It can compute the Euclidean (*EDT*), the Manhattan (*MDT*), and the chessboard distance (*CDT*) transform, defined by

$$\begin{aligned} EDT(x,y) &= \texttt{MIN}(i,j : 0 \le i < m \wedge 0 \le j < n \wedge b[i,j] : (x-i)^2 + (y-j)^2), \\ MDT(x,y) &= \texttt{MIN}(i,j : 0 \le i < m \wedge 0 \le j < n \wedge b[i,j] : |x-i| + |y-j|), \\ CDT(x,y) &= \texttt{MIN}(i,j : 0 \le i < m \wedge 0 \le j < n \wedge b[i,j] : |x-i| \ \mathbf{max} \ |y-j|). \end{aligned}$$

If we define the minimum of the empty set to be ∞, and use the rule $z + \infty = \infty$ for all z, we find with some calculation

$$\begin{aligned} EDT(x,y) &= \texttt{MIN}(i : 0 \le i < m : (x-i)^2 + G(i,y)^2), \\ MDT(x,y) &= \texttt{MIN}(i : 0 \le i < m : |x-i| + G(i,y)), \\ CDT(x,y) &= \texttt{MIN}(i : 0 \le i < m : |x-i| \ \mathbf{max} \ G(i,y)), \end{aligned}$$

where $G(i,y) = \texttt{MIN}(j : 0 \le j < n \wedge b[i,j] : |y-j|)$.

The algorithm can be summarized as follows. In a first phase each column C_x (defined by points (x,y) with x fixed) is separately scanned. For each point (x,y) on C_x, the distance $G(x,y)$ of (x,y) to the nearest points of $C_x \cap B$ is determined. In a second phase each row R_y (defined by points (x,y) with y fixed) is separately scanned, and for each point (x,y) on R_y the minimum of $(x-x')^2 + G(x',y)^2$ for *EDT*, $|x-x'| + G(x',y)$ for *MDT*, and $|x-x'| \ \mathbf{max} \ G(x',y)$ for *CDT* is determined, where (x',y) ranges over row R_y.

2.2 The first phase

The object of the first phase is to determine the function G. We first observe that we can split G into two functions GT (top) and GB (bottom), such that $G(i,y) = GT(i,y) \ \mathbf{min} \ GB(i,y)$, where

$$\begin{aligned} GT(i,y) &= \texttt{MIN}(j : 0 \le j \le y \wedge b[i,j] : y-j) \\ GB(i,y) &= \texttt{MIN}(j : y \le j < n \wedge b[i,j] : j-y) \end{aligned}$$

We start with the computation of GT by introducing an array g to store its values. It is easy to see that $GT(i,y) = 0$ if $b[i,y]$ holds, and that, otherwise, $GT(i,y) = \mathrm{G}T(i,y-1) + 1$ (or ∞ if $y = 0$). We can therefore compute $g[x,y] := GT(x,y)$ using only $g[x,y-1]$ in a simple column scan from top to bottom. Similarly, we find $GB(i,y) = \mathrm{G}B(i,y+1) + 1$. The second scan runs from bottom to top, and computes $G(x,y)$ directly, using GT from the previous scan, and GB from the current

```
forall x ∈ [0..m−1] do
  (* scan 1 *)
  if b[x,0] then
    g[x,0] := 0
  else
    g[x,0] := ∞;
  endif
  for y := 1 to n−1 do
    if b[x,y] then
      g[x,y] := 0
    else
      g[x,y] := 1+g[x,y−1];
    endif
  (* scan 2 *)
  for y := n−2 downto 0 do
    if g[x,y+1] < g[x,y] then
      g[x,y] := (1+g[x,y+1])
    endif
end forall
```

Figure 2.1. *Program fragment for the first phase.*

one. After some simplification, this results in the code fragment given in Fig. 2.1. Clearly, the time complexity is linear in the number of pixels (i.e. $O(m \times n)$). In actual implementations it is convenient to replace ∞ by $m+n$, since all distances in the images are less than $m+n$ if the set B is non-empty.

2.3 The second phase

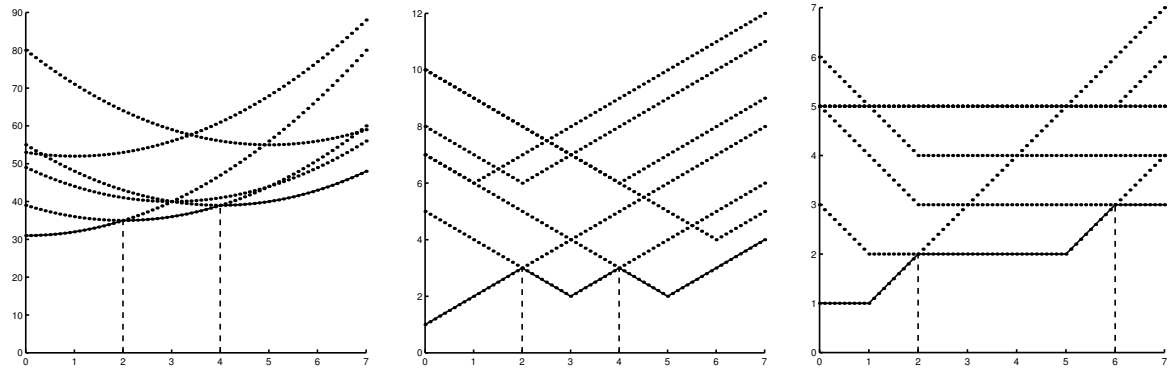
In the second phase we want to compute *EDT*, *MDT*, or *CDT* row by row, i.e. for all x with fixed y. Therefore, in this section we regard y as a constant and omit it as a parameter in auxiliary functions, and introduce $g(i) = G(i,y)$. Instead of developing an algorithm for each metric separately, we aim at a more general algorithm for

$$DT(x,y) = \texttt{MIN}(i : 0 \leq i < m : f(x,i)). \tag{2.1}$$

The choice of the function f depends on the metric we wish to use, i.e.

$$f(x,i) = \begin{cases} (x-i)^2 + g(i)^2 & \text{for } EDT, \\ |x-i| + g(i) & \text{for } MDT, \\ |x-i| \ \textbf{max}\ g(i) & \text{for } CDT. \end{cases}$$

It is helpful to introduce a geometrical interpretation of the minimization problem of Eq. 2.1. For any i with $0 \leq i < m$, denote by F_i the function $x \mapsto f(x,i)$ on the real interval $[0, m-1]$.



(a) *EDT* (b) *MDT* (c) *CDT*

Figure 2.2. DT *as the lower envelope (solid line) of curves F_i, $0 \leq i < m$ (dotted lines). The dashed vertical lines indicate the transitions between regions.*

We call i the *index* of F_i. In the case of *EDT*, the graph of F_i is a parabola with vertex at $(i, g(i))$. In the case of *MDT* the parabolas are replaced by V-shaped approximations, while in the case of *CDT* we deal with 'topped off' V-shaped approximations (see Fig. 2.2). We can interpret *DT* geometrically as the *lower envelope* of the collection $\{F_i | 0 \leq i < m\}$ evaluated at integer coordinates, cf. Fig. 2.2. The lower envelopes consist of a number of consecutive curve segments, whose index we denote by $s[0], s[1], \ldots, s[q]$ counting from left to right. The projections of the segments on the x-axis are called *regions*, and form a partition of the interval $[0, m)$ by consecutive segments. The computation of *DT* now consists of two scans. In a forward (left-to-right) scan the set of regions is determined using an incremental algorithm. In a backward (right-to-left) scan the values $DT(x,y)$ are trivially computed for all x.

We start by replacing the upper bound m in (2.1) by a variable u and define

$$FL(x,u) = \texttt{MIN}(i : 0 \leq i < u : f(x,i)).$$

The geometric interpretation is that we restrict the set B to the half plane to the left of u. Clearly, $DT(x,y) = FL(x,m)$.

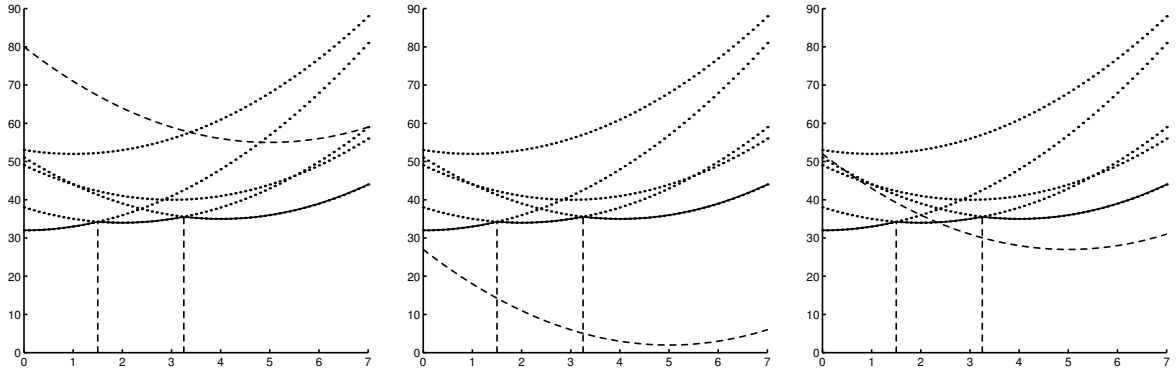
For given upper bound $u > 0$, we define an index h to be a *minimizer* at x if, in the expression for $FL(x,u)$, the minimal value of $f(x,i)$ occurs at h. In general, x may have more than one minimizer. The *least minimizer* $H(x,u)$ of x w.r.t. u is defined as the least index h with $0 \leq h < u$ such that $f(x,h) \leq f(x,i)$ for all i in the same range, i.e.

$$H(x,u) = \texttt{MIN}(h : 0 \leq h < u \wedge \forall (i : 0 \leq i < u : f(x,h) \leq f(x,i)) : h). \tag{2.2}$$

We clearly have $FL(x,u) = f(x,H(x,u))$, hence $DT(x,y) = f(x,H(x,m))$. Therefore, the problem reduces to the computation of $H(x,m)$.

We consider the sets $S(u)$ of the least minimizers that occur during the scan from left to right, and the sets $T(h,u)$ of points with the same least minimizer h. We thus define

$$\begin{aligned} S(u) &= \{H(x,u) \,|\, 0 \leq x < m\}\,, \\ T(h,u) &= \{x \,|\, 0 \leq x < m \wedge H(x,u) = h\} \text{ if } 0 \leq h < u. \end{aligned} \tag{2.3}$$



(a) above (b) below (c) intersection

Figure 2.3. *Location of F_u (dashed curve) w.r.t. the lower envelope (solid line).*

Clearly, $S(u)$ is a nonempty subset of $[0,u)$, and $S(u) = \{h \,|\, T(h,u) \neq \emptyset\}$. We define the *regions* for u to be the sets $T(h,u)$ that are nonempty. It is easy to see that the regions for u form a partition of $[0,m)$.

The aim is the case where $u = m$. Indeed, for $x \in T(h,m)$, we have $H(x,m) = h$ and hence $DT(x,y) = f(x,h)$. The second phase of the algorithm therefore consists of two scans: scan 3 computes the partition of $[0,m)$ that consists of the regions for m and scan 4 uses these regions to compute DT. For given u, only the curves with indices from 0 to $u-1$ are taken into account. The minimizer of x corresponds to the index of the curve segment whose projection on the horizontal axis contains x. Let the current lower envelope consist of $q+1$ segments, i.e. $S(u) = \{s[0], s[1], \ldots, s[q]\}$, with $s[\ell]$ the index of the ℓ-th segment. Consider what happens when F_u is added. Three situations may occur:

(a) F_u is *above* the current lower envelope on $[0,m-1]$, cf. Fig. 2.3(a). Then $S(u+1) = S(u)$, since the set $T(u,u+1)$ is empty.

(b) F_u is *below* the current lower envelope on $[0,m-1]$, cf. Fig. 2.3(b). Then $S(u+1) = \{u\}$, i.e., all old regions have disappeared, and there is one new region $T(u,u+1) = [0,m)$.

(c) F_u *intersects* the current lower envelope on $[0,m-1]$, cf. Fig. 2.3(c). The current regions will either shrink or disappear, and there is one new region $T(u,u+1)$.

We start searching from right to left for the current region which is intersected by F_u. This can be determined by comparing the values of F_u and F_ℓ at the begin point $t[\ell]$ of each current region $\ell = q, q-1, \ldots$, until we find the first $\ell = \ell^*$ such that $F_u(t[\ell^*]) \geq F_{s[\ell^*]}(t[\ell^*])$. Then F_u is not the least minimizer at $t[\ell^*]$, and there must be an intersection of F_u with F_{ℓ^*} in region ℓ^*. Let x^* be the horizontal coordinate of the intersection. If $\ell^* = q$ and $x^* \geq m$ we have case (a); if $\ell^* < 0$ we have case (b); otherwise case (c) pertains.

To find x^*, we introduce a function *Sep*, where $Sep(i,u)$ is the first integer larger or equal than the horizontal coordinate of the intersection point of F_u and F_i with $i < u$, i.e.

$$F_i(x) \leq F_u(x) \quad \Leftrightarrow \quad x \leq Sep(i,u). \tag{2.4}$$

```
forall y ∈ [0..n−1] do
  q := 0; s[0] := 0; t[0] := 0;
  for u := 1 to m−1 do (∗ scan 3 ∗)
    while q ≥ 0 ∧ f(t[q],s[q]) > f(t[q],u) do
      q := q−1;
    if q < 0 then
      q := 0; s[0] := u
    else
      w := 1+Sep(s[q],u);
      if w < m then
        q := q+1; s[q] := u; t[q] := w
      end if
    end if
  end for
  for u := m−1 downto 0 do (∗ scan 4 ∗)
    dt[u,y] := f(u,s[q]);
    if u = t[q] then q := q−1
  end for
end forall
```

Figure 2.4. *Program fragments for the second phase.*

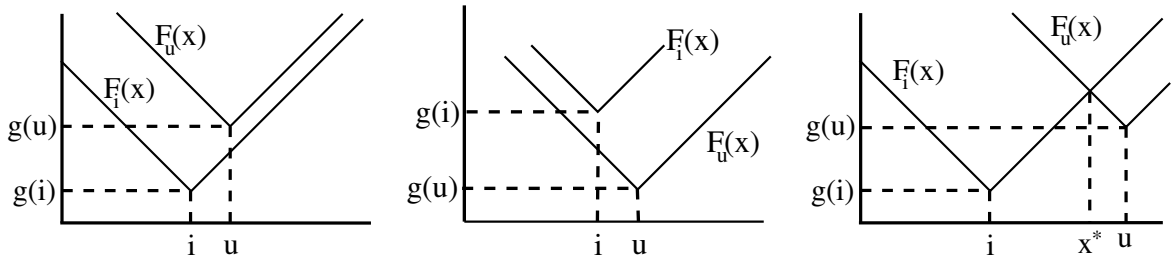
We thus have $x^* = Sep(s[l^*],u)$. Clearly, the function *Sep* is dependent on which distance transform we want to compute. In the next section we will derive the expressions for the function *Sep*, but in the remainder of this section we simply assume that *Sep* is available.

We introduce an integer program variable u. It is convenient to represent $S(u)$ by an increasing sequence of elements. Since the regions form a partition of $[0,m)$ by consecutive segments, we can represent them by the sequence of their least elements. According to the case analysis above, the regions are to be adapted at their end. We can therefore implement these sequences in two integer arrays, s and t, with an integer variable q as index of the end point.

We start with the forward scan, see scan 3 in Fig. 2.4. We have $S(1) = \{0\}$, and $T(0,1) = [0,m)$, and thus start with $q = 0$, $s[0] = 0$, and $t[0] = 0$. In a loop, variable u is incremented, and thus the representations of S and T must be updated by means of the case analysis above.

To investigate the complexity of the forward scan, we consider the expression $q + 2(m-u)$, which is initially $2m$. In every execution of the body of the outer loop (scan 3 in Fig. 2.4), and also in every execution of the body of its inner loop, the value of the expression decreases. This implies that the time complexity of the scan is linear in m. Note that, the *average* number of iterations of the inner loop is at most two. The algorithm uses less than $2m$ comparisons of f values, and function *Sep* is evaluated less than m times.

When the forward scan is finished, we have completely determined the partition of $[0,m)$ in regions. Given these regions, we can trivially compute dt-values in a simple backward scan (see scan 4 in Fig. 2.4).



(a) $g(u) \geq g(i)+u-i$ (b) $g(i) > g(u)+u-i$ (c) otherwise

Figure 2.5. *Cases for finding Sep for MDT.*

2.4 Derivation of the function Sep

The derivation in the previous section was independent of the actual metric used. The functions dependent on the metric are f and *Sep*. In this section we compute expressions for *Sep* for *EDT*, *MDT*, and *CDT*. The easiest is *EDT*. We find for $i < u$

$$\begin{aligned}
& F_i(x) \leq F_u(x) \\
\Leftrightarrow & \{\text{definition of } F_i, F_u\} \\
& (x-i)^2 + g(i)^2 \leq (x-u)^2 + g(u)^2 \\
\Leftrightarrow & \{\text{calculus; } i < u\text{; } x \text{ is an integer}\} \\
& x \leq (u^2 - i^2 + g(u)^2 - g(i)^2) \textbf{ div } (2(u-i)).
\end{aligned}$$

Here, we denote integer division with rounding off towards zero by **div**. Thus, we find for *EDT* that

$$Sep(i,u) = (u^2 - i^2 + g(u)^2 - g(i)^2) \textbf{ div } (2(u-i)).$$

If we use the Manhattan metric, the analysis is slightly more complicated. Since we have to deal with absolute values in the expressions, awkward case analysis is necessary if we want to compute *Sep* analytically. Therefore we prefer a geometric argument. We have to consider three cases (see Fig. 2.5).

If $g(u) \geq g(i)+u-i$, the graph of F_u lies entirely above the graph of F_i for all x, thus we choose $Sep(i,u) = \infty$. If $g(i) > g(u)+u-i$, the graph of F_i lies entirely above the graph of F_u, so $F_i(x) \leq F_u(x)$ for no x at all. Thus, we must choose $Sep(i,u) = -\infty$ to satisfy (2.4). In all other cases, F_u intersects F_i at $x^* = (g(u) - g(i) + u + i)/2$. So, if we want to compute *MDT* we use

$$Sep(i,u) = \begin{cases} \infty & \text{if } g(u) \geq g(i)+u-i, \\ -\infty & \text{if } g(i) > g(u)+u-i, \\ (g(u) - g(i) + u + i) \textbf{ div } 2 & \text{otherwise.} \end{cases}$$

For the case of *CDT* we have $|x-i| \textbf{ max } g(i) \leq |x-u| \textbf{ max } g(u)$. We consider two main cases, which each can be split up in three sub-cases. First we consider the case $g(i) \leq g(u)$. From Fig. 2.6(a)-(c), we see that the increasing segment of F_i ($y = x-i$) intersects the decreasing part of F_u ($y = u-x$), or the constant part ($y = g(u)$). Let γ be the vertical coordinate corresponding

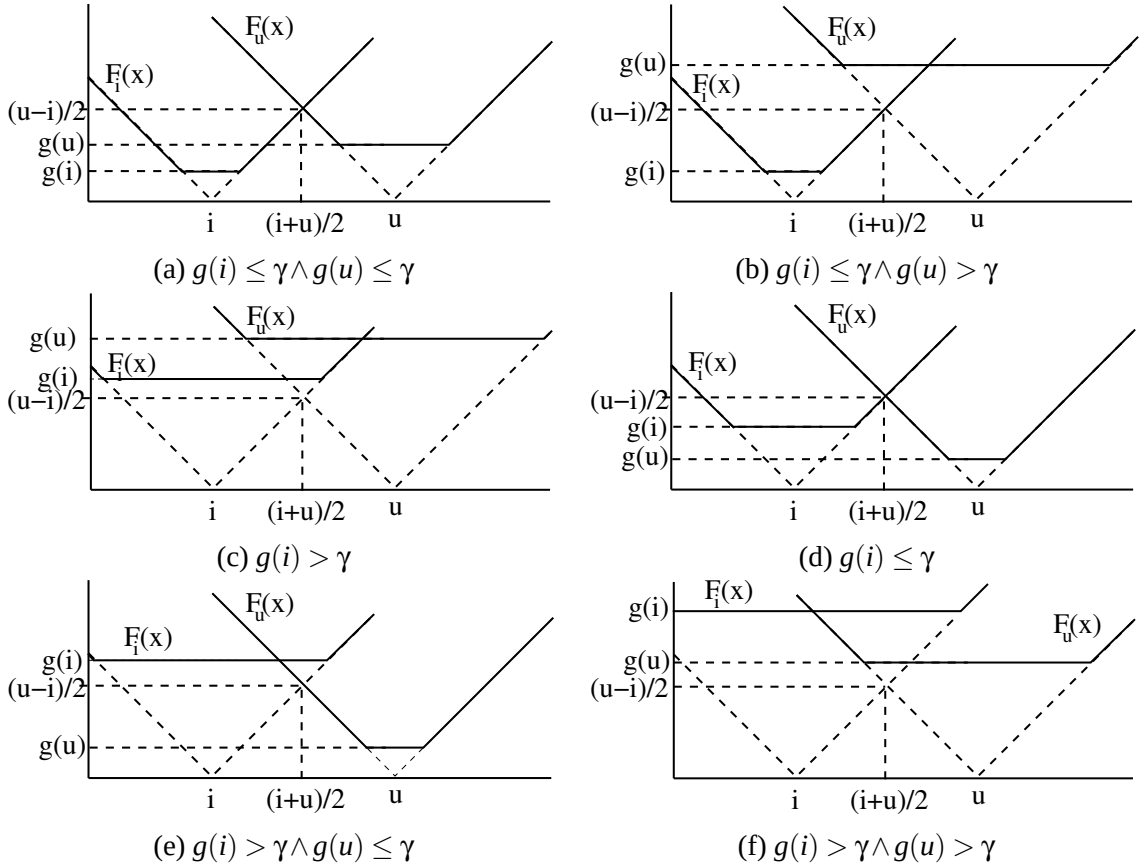


Figure 2.6. *Cases for finding Sep for CDT, where* $\gamma = (u-i)/2$. *Cases (a)-(c):* $g(i) \leq g(u)$. *Cases (d)-(f):* $g(i) > g(u)$.

with the middle of i and u ($x = (i+u)/2$), i.e. $\gamma = (u-i)/2$. From Fig. 2.6(a), we see that if $g(i) \leq \gamma \wedge g(u) \leq \gamma$, we have $F_i(x) \leq F_u(x)$ if $x \leq (i+u)/2$. From Fig. 2.6(b)-(c), we see that the increasing part of F_i intersects the constant segment of F_u at $i+g(u)$, and thus we have $F_i(x) \leq F_u(x)$ if $x \leq i+g(u)$. Putting the three cases together, we can conclude

$$g(i) \leq g(u) \Rightarrow \big(F_i(x) \leq F_u(x) \Leftrightarrow x \leq \frac{i+u}{2} \ \mathbf{max}\ (i+g(u))\big).$$

The other main case is $g(i) > g(u)$. Again, in Fig. 2.6(d), we see that if $g(i) \leq \gamma$, the intersection at $(i+u)/2$ is the separator. If $g(i) > \gamma$ (see Fig. 2.6(e)-(f)), the horizontal segment of F_i intersects the decreasing part of F_u at $x = u - g(i)$. Just like in the previous case, we can put these cases together. This results in the following expression for *Sep*:

$$Sep(i,u) = \begin{cases} (i+g(u)) \ \mathbf{max}\ ((i+u) \ \mathbf{div}\ 2) & \text{if } g(i) \leq g(u), \\ (u-g(i)) \ \mathbf{min}\ ((i+u) \ \mathbf{div}\ 2) & \text{otherwise.} \end{cases}$$

Table 2.1. *Timing results in ms. From left to right: EDT, MDT, and CDT.*

size	p=1	p=2	p=3	p=4	p=1	p=2	p=3	p=4	p=1	p=2	p=3	p=4
256	12	7	5	4	11	6	4	3	12	6	4	3
512	69	35	25	19	63	34	24	17	67	35	25	18
1024	307	156	104	79	281	147	97	74	298	152	101	77
2048	1542	780	517	389	1407	709	476	357	1501	753	506	381
4096	6251	3137	2098	1577	5753	2886	1929	1451	6073	3053	2041	1530

2.5 Parallelization, timing results and conclusions

Since the computation per row (column) is independent of the computation on any other row (column), the algorithm is well suited for parallelization on a shared memory machine. In the first (second) phase, the columns (rows) are distributed over the processors. The two phases must be separated by a *barrier*, which assures that all processors have completed the first phase before any of them starts with the second phase. The theoretical time complexity of the parallel algorithm for p processors (where $p \leq m$ **min** n) is $O(mn/p)$.

We ran experiments on an Intel Pentium III based shared memory parallel computer with 4 cpu's, running at a 550MHz clock frequency. We performed time measurements using several binary images, and found that the execution time is almost independent of image content, and scales well w.r.t. the number of processors. This is as expected, since the amount of work per row and column is almost the same. In table 2.1 the timings for square images are given for $p = 1$ to $p = 4$ processors. Note that the computation of *MDT* and *CDT* is only slightly faster than the exact *EDT*. We also implemented the sequential algorithm of [81] for *CDT*, and found that our algorithm is less than a factor of 2 slower, which can easily be overcome by parallel processing.

The algorithm can be easily extended to d-dimensional distance transforms by separating the problem into d phases, each solving a one-dimensional problem, as carried out above for the case $d = 2$.

This chapter has been published as: W. H. Hesselink, A. Meijster and C. Bron, Concurrent Determination of Connected Components, *Sci. Comp. Programming*, **41**, pp. 173-194, 2001.

Chapter 3

Concurrent Determination of Connected Components

Abstract

The design is described of a parallel version of Tarjan's algorithm for the determination of equivalence classes in graphs that represent images. Distribution of the vertices of the graph over a number of processes leads to a message passing algorithm. The algorithm is mapped to a shared-memory architecture by means of POSIX threads. It is applied to the determination of connected components in image processing. Experiments show a satisfactory speedup for sufficiently large images.

3.1 Introduction

In many image processing applications, one of the first steps is to compute the connected components of the image. For this purpose one usually takes the simple breadth first scanning algorithm, which stems from the corresponding problem in graph theory. This algorithm has the disadvantages that it requires a FIFO-queue the size of which is a priori unknown, and that it is hard to parallelize. The number of pixels involved is often large, say more than a million, and for real-time applications often several images must be processed per second. It is therefore important to have an efficient parallel algorithm for this task. This is confirmed by the fact that there are many articles on parallel image component labelling. Most of these articles aim at distributed memory architectures, e.g., cf. [2, 22, 36].

Two classical sequential algorithms that explicitly use the fact that the graph is an image, are given in [48, 82]. The main drawback of these algorithms is the use of a large equivalence table. Inspired by these two algorithms and Tarjan's disjoint set algorithm, [89], we here present an algorithm that does not need such a large table, and that can elegantly be parallelized. The sequential algorithm on itself is not new ([32]), but as far as we know there does not exist a parallel implementation of this algorithm, which is the main focus of this chapter. The algorithm can be implemented on distributed as well as shared memory machines.

The algorithm determines a directed spanning forest for an undirected graph by placing links that are not necessarily along the edges of the graph. It is meant for large graphs, the nodes

of which are distributed over a relatively small number of processes, preferably in such a way that most of the edges belong to only one process. In this respect, the situation differs from settings as investigated in [49, 91], where the processes are in one-to-one correspondence with the nodes. Indeed, a typical setting for our algorithm could be a medical application used by medical specialists to analyse 3-D CT-volumes or MRI-volumes. In that case, the graph may have in the order of 10^8 points and the computation can be distributed over, say, four up to sixteen processors.

Although we are especially interested in the application to images, we present the algorithm for general undirected graphs. The design goes through four stages. We first give a version of Tarjan's sequential algorithm, then distribute this over several processes with message passing. This design is then mapped to a shared memory architecture by means of mutual exclusion and synchronization. Finally, the mutual exclusion and synchronization are implemented by means of POSIX thread primitives.

The resulting algorithm is a concurrent one in which the amount of communication is decided at runtime. Such algorithms are very error prone. Our presentation may seem to focus on logic, but that is not the case. Since we want a working algorithm, our focus is on correctness, i.e., preservation of invariants, avoidance of deadlock, and guarantee of progress. Logical formulae are the only way to unambiguously express the properties needed.

Since we want to avoid unnecessary communication, we use no path compression beyond the parts of the graph under control of a single process. If the vertices of the graph are distributed randomly over the processes, this leads to bad worst case performance (i.e. quadratic in the lengths of the paths). In practice, however, there is often a natural way to distribute the nodes over the processes such that most edges adjacent to a node belong to only one process. In that case, the performance of the algorithm is quite good.

We finally describe the application to the determination of connected components in images. Since images are usually more or less constant locally, we sketch an optimization that can reduce the number of communications needed significantly. The results show that the algorithm makes distribution quite effective.

Overview. In Section 3.2 we give the abstract problem and develop a sequential solution. In Section 3.3, the algorithm is distributed over several processes in an asynchronous way. In Section 3.4, we specialize to a shared memory architecture in bounded space with atomicity brackets and **await** statements. In Section 3.5, these constructs are implemented by means of POSIX thread primitives. Section 3.6 describes the finalization of the algorithm. Section 3.7 contains the application to image processing. We draw conclusions in Section 3.8.

3.2 The problem and a sequential solution

In the image processing context, points of an image are regarded as directly connected if they are neighbours and have (nearly) the same colour or grey-value. The problem is to determine the connected components of the image. Since images contain many points, and since one may want to process many subsequent images in real time, there is reason to consider distributed solutions. Graph theory is the proper abstract setting for any discussion of connected components.

We therefore let the image be represented by an undirected graph. The aim is to determine its connected components by means of a distributed algorithm. Our first step is the design of a sequential algorithm, which is a variation of Tarjan's algorithm, cf. [89], chapter 2, [94] 12.3.

Let (V,E) be an undirected graph. We regard E as a (symmetric binary) relation. The connected components of the graph are the equivalence classes of the reflexive transitive closure E^* of E. The idea is to represent the components by rooted trees by means of an array variable

$$\texttt{par} : \textbf{array}\ V\ \textbf{of}\ V\ ,$$

which stands for "parent". We define function $root : V \to V$ by

$$root(n) \quad = \quad \textbf{if}\ \texttt{par}[n] = n\ \textbf{then}\ n\ \textbf{else}\ root(\texttt{par}[n])\ \textbf{fi}\ .$$

Since V is finite, function *root* is well-defined if and only if the directed graph induced by the arrows of par has no cycles of length > 1. We want to establish the postcondition that function *root* is well-defined and satisfies

$$Q: \quad (\forall\, m,n ::\ (m,n) \in E^* \quad \equiv \quad root(m) = root(n))\ .$$

In order to establish Q, we introduce the equivalence relation *Con* given by

$$(m,n) \in Con \quad \equiv \quad root(m) = root(n)\ .$$

Now Q is equivalent to $E^* = Con$.

We assume that the initialization establishes $\texttt{par}[n] = n$ for all $n \in V$. Then function *root* is well-defined and relation *Con* is equal to the identity. We shall modify array par in such a way that function *root* remains well-defined and that relation *Con* is only extended. We therefore only modify par by assignments of the form $\texttt{par}[x] := y$ under one of the preconditions

$$\begin{array}{ll} P0(x,y): & (\exists\, k : k \geq 1 : \texttt{par}^k[x] = y)\ , \\ P1(x,y): & \texttt{par}[x] = x \quad \wedge \quad root(y) \neq x\ . \end{array}$$

Here, $\texttt{par}^k[x]$ is obtained by k subsequent applications of par on index x. In the case of $P0(x,y)$, node y is an ancestor of x and the assignment $\texttt{par}[x] := y$ does not modify relation *Con*. Such an assignment is called *path compression*, cf. [1]. In case of $P1(x,y)$, node x is a root and not an ancestor of y. Since y becomes the parent of x, relation *Con* is strictly extended.

We now come to the edge relation E of the graph. Since we do not want to store every unordered pair twice, we assume that relation E is represented by a set *edlis* of pairs of nodes via the initial relation $E = sym(edlis)$ where function *sym* is defined by

$$(m,n) \in sym(R) \quad \equiv \quad (m,n) \in R \quad \vee \quad (n,m) \in R\ .$$

We take *edlis* to be a program variable and introduce the loop invariant

$$J0: \quad E^* \quad = \quad (Con \cup sym(edlis))^*\ .$$

Predicate $J0$ holds initially, since then $sym(edlis) = E$ and *Con* is the identity. If *edlis* is empty, $J0$ implies predicate Q since *Con* is an equivalence relation. We therefore take $edlis \neq empty$ as the guard of the loop.

Now the abstract sequential algorithm is

A: **while** $edlis \neq empty$ **do**
  fetch (u,v) *from edlis* ;
  Extend
od ,

where command *Extend* has to restore predicate $J0$ if it is falsified by the removal of (u,v) from *edlis*. Restoration can be done by placing a `par` pointer between the components of u and v. We therefore search for elements x, y, connected to u and v, that satisfy $P1(x,y)$. We thus introduce the predicate

$$JE: \quad (u,x),(v,y) \in Con \quad \vee \quad (u,y),(v,x) \in Con \ ,$$

and describe *Extend* by

Extend: **if** $(u,v) \notin Con$ **then**
  choose x,y *with* $P1(x,y) \wedge JE$;
  $\texttt{par}[x] := y$
fi .

It is easy to see that in this way $J0$ is preserved and that, consequently, algorithm A is correct. So it remains to implement *Extend*. Since relation *Con* is not directly available, we implement *Extend* by means of a loop with JE as invariant. Since *Con* is an equivalence relation, $JE \wedge x = y$ implies $(u,v) \in Con$. We can therefore refine *Extend* as follows.

Extend: $x := u$; $y := v$ $\{JE\}$;
while $x \neq y \wedge \neg P1(x,y)$ **do**
  modify x,y *while preserving* JE
od ;
if $x \neq y$ **then** $\texttt{par}[x] := y$ **fi** .

For the ease of distributed verification of the inequality $root(y) \neq x$ in $P1(x,y)$, we assume that the set V has a total order $\leq$ and introduce the additional invariant (compare [94] p. 261):

$$J1: \quad \texttt{par}[n] \leq n \ .$$

Here and henceforth, we use the convention that all invariants are universally quantified over the free variables they contain (here n).

We now decide that the loop in *Extend* preserves the invariant $JE \wedge x \geq y$. We therefore assume that the pairs in *edlis* are ordered by

$$J2: \quad (m,n) \in edlis \quad \Rightarrow \quad m > n \ .$$

In the body, we replace x by $\texttt{par}[x]$ and, if necessary, restore $x \geq y$ by swapping. Now the guard of the loop can be simplified since $J1 \wedge x \geq y$ implies

$$P1(x,y) \quad \equiv \quad \texttt{par}[x] = x \quad \wedge \quad x \neq y\,.$$

It follows that

$$x \neq y \wedge \neg P1(x,y) \quad \equiv \quad x \neq y \wedge \texttt{par}[x] \neq x,$$

and thus we obtain

Extend: $x := u\,;\quad y := v\,;$
while $x \neq y \wedge \texttt{par}[x] \neq x$ **do**
$x := \texttt{par}[x]\,;$
if $x < y$ **then** $x,y := y,x$ **fi**
od ;
if $x \neq y$ **then** $\texttt{par}[x] := y$ **fi** .

Since V is finite, it is easy to see that the loop terminates.

The efficiency of algorithm A can be improved considerably by path compression, i.e., by extending the final **then** branch of *Extend* with assignments $\texttt{par}[z] := y$ for all nodes z on the par paths of u and v. This optimization preserves all invariants. A simple version of it only adds $\texttt{par}[u] := y$ and $\texttt{par}[v] := y$. In our application this seems to be just as effective.

3.3 Distribution

In this section, we distribute algorithm A over a system of sequential processes that communicate by message passing. We use the following convention with respect to private variables. If x is a private variable of process p, we refer to it as x in the code and as $x.p$ if p is not obvious from the context. Let *Process* be the set of processes. We assume that the set V is distributed over the processes by means of a function $owner : V \to Process$. We assume that process p is allowed to inspect and modify $\texttt{par}[x]$ if and only if $p = owner(x)$.

We assume that *edlis* is distributed over the processes as well. So, each process p has its own set $edlis(p)$ and we regard *edlis* as an alias for the union of the sets $edlis(p)$. We introduce the invariant

$$J3: \quad (m,n) \in edlis(q) \quad \Rightarrow \quad owner(m) = q\,.$$

Since the loop in *Extend* can only be executed by process p as long as $owner(x) = p$, we introduce the local search command

Search: $x := u\,;\quad y := v\,;$
while $owner(x) = self \wedge x \neq y \wedge \texttt{par}[x] \neq x$ **do**
$x := \texttt{par}[x]\,;$
if $x < y$ **then** $x,y := y,x$ **fi**
od ,

where *self* stands for the executing process. Since the guards are evaluated from left to right, $\mathtt{par}[x]$ is not inspected if $owner(x) \neq self$. Execution of *Search* establishes the postcondition

$$owner(x) \neq self \quad \vee \quad x = y \quad \vee \quad \mathtt{par}[x] = x\,.$$

It is now clear that each process should repeatedly execute

fetch (u,v) *from* $edlis(self)$;
Search ;
if $x \neq y$ **then**
  if $owner(x) = self$ **then** $\mathtt{par}[x] := y$
  else *put* (x,y) *into* $edlis(owner(x))$ **fi**
fi .

This program fragment preserves $J0 \wedge J1 \wedge J2 \wedge J3$, i.e., indeed, $J0$, $J1$, $J2$, $J3$ are invariants. It terminates for the same reason as in the case of the sequential algorithm.

In this way, the sets $edlis(p)$ become buffers with one consumer and many producers. Process p fetches elements from $edlis(p)$ and other processes may put elements into it. These actions therefore require communication: the last line of this fragment can be read as "send (x,y) to the process that owns x".

Since communication is expensive in performance, we partition the set $edlis(p)$ into two parts $edlis0(p)$ and $edlis1(p)$, and assume the invariant $edlis(p) = edlis0(p) \cup edlis1(p)$ with initially

$$edlis0(p) = \{(u,v) \in edlis(p) \,|\, owner(v) = p\}\ ;$$
$$edlis1(p) = \{(u,v) \in edlis(p) \,|\, owner(v) \neq p\}\ .$$

We can therefore treat $edlis0(p)$ in an initial program fragment A0, obtained from A by substituting $edlis0(p)$ for $edlis$.

A0: **while** $edlis0(self) \neq empty$ **do**
  fetch (u,v) *from* $edlis0(self)$;
  Extend
od ,

Since initially $\mathtt{par}[z] = z$ for all z, fragment A0 preserves the invariant that $owner(\mathtt{par}[z]) = p$ for all z with $owner(z) = p$.

During the treatment of $edlis1(p)$, process p must be able to put elements into $edlis1(q)$ where q is some process with $q \neq p$. As a consequence, process p must not stop when its set $edlis1(p)$ is empty since other processes may insert new elements in $edlis1(p)$. We declare for each process a private variable *continue* to indicate that new pairs yet may arrive.

A1: **while** *continue* **do**
  fetch (u,v) *from* $edlis1(self)$;
  Search ;
  if $x \neq y$ **then**

```
            if owner(x) = self then par[x] := y
            else put (x,y) into edlis1(owner(x)) fi
        fi
    od .
```

The program for process p now becomes the composition of the two parts A0 and A1. Part A0 needs no further refinement. Part A1 primarily requires termination detection: how to give the boolean variables *continue* the adequate values?

We assume that process p starts up with initial values for $edlis0(p)$ and $edlis1(p)$. The size of the union of the sets $edlis1(p)$ only shrinks. Every process can terminate when all sets $edlis(p)$ are empty and each process has finished its local computation, but not earlier. To keep track of the edges that have yet to be treated, we attach a unique token t to each edge (u,v) in $edlis1(p)$. This token serves to indicate the originator of the pair (u,v) for the sake of termination detection. It is sent unmodified with the changing edge (u,v) as a message $edge(u,v,t)$. When no triple is sent, the token t is destroyed.

Each token shall belong to the process that creates it. We represent the assignment of tokens to processes by a function $origin : Token \to Process$. Each process gets a private integer variable *ctok* to count its number of outstanding tokens. Whenever a token is destroyed, a message *down* is sent to its origin. A process decrements *ctok* when it receives a message *down*. We thus have the invariant that *ctok* of process q is the number of messages $edge(u,v,t)$ in transit with $origin(t) = q$ plus the number of *down* messages in transit to q. This can be expressed by

$$J4: \quad ctok.q = \#\{edge(u,v,t) \mid origin(t) = q\} + transit(down, q) \ ,$$

where we use $transit(m,q)$ to denote the number of messages m in transit to q, and $\#A$ to denote the number of elements of the set A.

We introduce a message *stop* to signal termination. Indeed, when all tokens of all processes have been destroyed, all buffers are empty and every process may terminate.

In order to decide that all tokens of all processes have been destroyed, we introduce a global counter gc for the number of processes that are initializing or have $ctok > 0$. We give one process, say *adm*, the additional task to administrate the value of gc, which initially equals $\#Process$. A process that reaches $ctok = 0$, sends a *gcdown* message to *adm*. We postulate the invariant that gc equals the number of processes q with $ctok.q > 0$ plus the number of *gcdown* messages in transit, i.e.

$$J5: \quad \mathtt{gc} = \#\{q \mid ctok.q > 0\} + transit(gcdown, adm) \ .$$

When process *adm* receives the message *gcdown* it decrements gc and, if gc becomes 0, it sends messages *stop* to all processes, as expressed in command *GcDown*:

```
GcDown: gc := gc − 1 ;
        if gc = 0 then
            for all q ∈ Process do send stop to q od
        fi .
```

A process that receives *stop*, sets *continue* to false. This leads to the invariants

$J6:$ $\quad continue.q \quad \equiv \quad \mathtt{gc} > 0 \quad \vee \quad transit(stop,q) > 0$;
$J7:$ $\quad transit(stop,q) > 0 \quad \Rightarrow \quad \mathtt{gc} = 0$.

Fragment A1 is now replaced by

```
A1:   Init1 ;
      while continue do
         in edge(u,v,t)  →
               Search ;
               if x ≠ y ∧ owner(x) ≠ self then
                  send edge(x,y,t) to owner(x)
               else
                  if x ≠ y then par[x] := y fi ;
                  send down to origin(t) ;
               fi
         [] down   →
               ctok := ctok − 1 ;
               if ctok = 0 then send gcdown to adm fi
         [] gcdown  →  GcDown
         [] stop   →  continue := false
         ni
      od .
```

The auxiliary command *Init1* is treated below. The rest of A1 is a repetition that consists of reception and treatment of messages. For this purpose, we use a variation of the **in...ni** construct of the language SR of [4]. It involves waiting for the next message to arrive, the choice according to the arriving message, and it introduces formal parameters for the arguments of the message, if any. Note that this code implies that a process may send asynchronous messages to itself. Such messages can easily be eliminated. We have not done so for the sake of uniformity.

After the treatment of $edge(u,v,t)$, the process may perform path compression along the two paths it has investigated in its own part of the graph. In view of the communication overhead, we decided not to consider more extensive forms of path compression.

For the sake of uniformity, the initialization of A1 translates the edges in *edlis1* into *edge* messages from the process to itself. A1 is thus initialized by

```
Init1:  ctok := 0 ;
        continue := true ;
        for all (x,y) ∈ edlis1(self) do
           create a token t with origin(t) = self ;
           ctok := ctok + 1 ;
           send edge(x,y,t) to self
        od ;
        if ctok = 0 then send gcdown to adm fi .
```

In order to verify the invariants, we first need to describe the execution model. Processes are concurrently allowed to receive and execute messages. Since the effect of execution of a message only depends on the message and the precondition of the accepting process, we may (for the sake of the correctness proof) assume that the messages are accepted by the processes in some linear order and that a message is accepted only when the command associated to the previous message has been executed completely by the previous accepting process. In other words, in our model, the acceptance of a message includes atomic execution of the associated command. The invariants are predicates that are supposed to hold before and after each complete acceptance of a message.

It is now straightforward to verify the invariants $J4$, $J5$, $J6$, and $J7$. Indeed, each of these predicates holds when all processes have completed *Init1*. Acceptance of a message *edge* leads to re-sending of a message *edge* or *down*. Therefore, $J4$ is preserved. Acceptance of *down* by process p preserves $J4$ since $ctok.p$ is decremented. It also preserves $J5$, since *gcdown* is sent if $ctok.p$ reaches 0. Acceptance of *gcdown* by process *adm* preserves $J5$ since $\mathtt{gc}$ is decremented. It also preserves $J6$ and $J7$ since *stop* is sent if and only if $\mathtt{gc}$ reaches 0. Acceptance of *stop* by process p preserves $J6$ since $continue.p$ is set to false and $J7$ implies $\mathtt{gc} = 0$.

It follows from $J4 \wedge J5$ that, while there are edges to be processed, we have $\mathtt{gc} > 0$, so that $J6$ implies that all processes have not yet terminated. On the other hand, when there are no messages in transit, then $J4 \wedge J5$ implies that $\mathtt{gc} = 0$, so that $J6$ implies $\neg\, continue.q$ for all processes q. So, then, all processes have terminated.

3.4 Bounded shared memory

We now assume that the processes communicate by means of shared memory, and that the size of this memory is bounded. We use the convention that shared variables are in typewriter font. In this section we specify the requirement on atomicity and synchronization by means of atomicity brackets and **await** statements. The next section is devoted to the implementation of these constructs by means of the POSIX thread primitives.

We eliminate the messages *edge*, *down* and *stop*, and replace them by procedures *PutEdge*, *Down*, and *Stop*. The edge triples that are to be communicated between the processes will be placed somewhere in the shared memory. Each process is equipped with a private list of such triples and has a private variable *head0* that serves as the head of this list. The private list is empty iff $head0 = nil$. Procedure *GetEdge* fetches a triple from the private list.

We introduce a procedure *AwaitEdge*, the postcondition of which implies that $head0 \neq nil$ or *Stop* has been called. Then program fragment A1 is replaced by

```
A2:   Init2 () ;
      loop
          AwaitEdge () ;
          if head0 = nil then exitloop fi ;
          GetEdge (u,v,t) ;
          Search ;
```

```
    if x ≠ y ∧ owner(x) ≠ self then
        PutEdge (owner(x), x, y, t)
    else
        if x ≠ y then par[x] := y fi ;
        Down (origin(t))
    fi
endloop .
```

In order to replace the messages *down* by a procedure *Down*, we replace the private variables *ctok* by a shared variable

```
ctok : array Process of Integer ,
```

and we define

```
procedure Down (q : Process) =
    var b : Boolean ;
    ⟨ ctok[q] := ctok[q] − 1 ;  b := (ctok[q] = 0) ⟩ ;
    if b then GcDown () fi
end .
```

Here, atomicity brackets ⟨ ⟩ are used to specify that the command enclosed by them shall be executed without interference. Now *GcDown* is a procedure given by

```
procedure GcDown () =
    var b : Boolean ;
    ⟨ gc := gc − 1 ;  b := (gc = 0) ⟩ ;
    if b then Stop () fi
end .
```

We replace the private variables *continue* by a shared array

```
cntu : array Process of Boolean ;
```

with initially cntu[*q*] = *true* for all processes *q*. We then define procedure *Stop* by

```
procedure Stop () =
    for all q ∈ Process do ⟨ cntu[q] := false ⟩ od
end .
```

Note that, in this way, the special process *adm* is eliminated.

Remark. One could of course replace the array cntu by a single boolean variable. This would cause memory contention, however, when many processes try to access it concurrently. We therefore prefer to use an array. □

We finally come to the central problem of a shared data structure where the processes can deposit the edges destined for other processes. For this purpose, we assume that there is a constant M such that $\#edlis1(p) \leq M$ for all processes p. Let N be the number of processes. It then follows that we need at most $N * M$ tokens. We thus define the type $Token = [0\,..\,N*M-1]$ and use this type as the index set for the messages. We decide to store the triple (x,y,t) always at index t by means of the shared variable

$\texttt{pair} : \textbf{array}\ Token\ \textbf{of}\ V \times V\ .$

The message buffers are constructed as lists of pairs. For this purpose, we introduce a value $nil \notin Token$ to designate the empty list and declare the shared variables

$\texttt{next} : \textbf{array}\ Token\ \textbf{of}\ Token \cup \{nil\}\ ;$
$\texttt{head} : \textbf{array}\ Process\ \textbf{of}\ Token \cup \{nil\}\ ;$

with initially $\texttt{head}[q] = nil$ for all q. We use $\texttt{head}[q]$ as the head of the list for process q where other processes can write. Now procedure *PutEdge* is given by

procedure *PutEdge* $(q : Process;\ x,y : V;\ t : Token) =$
  $\texttt{pair}[t] := (x,y)\ ;$
  $\langle\ \texttt{next}[t] := \texttt{head}[q]\ ;$
    $\texttt{head}[q] := t\ \rangle$
end .

Reading and writing of $\texttt{head}[q]$ must be done under mutual exclusion. The assignment to $\texttt{pair}[t]$ is not threatened by interference, however, since we preserve the invariant that there is always at most one process that holds token t.

Recall that every process also has a private variable *head0* as the head of a private list of tokens. A process fetches an element from its private list by the simple procedure

procedure *GetEdge* $(\textbf{var}\ x,y : V;\ \textbf{var}\ t : Token) =$
  $t := head0\ ;$
  $head0 := \texttt{next}[t]\ ;$
  $(x,y) := \texttt{pair}[t]$
end .

In procedure *AwaitEdge*, the two lists of a process are swapped whenever the private list is empty and the public one is not:

procedure *AwaitEdge* $() =$
  if $head0 = nil$ **then**
    $\langle\ \textbf{await}\ \texttt{head}[self] \neq nil \ \vee\ \neg\texttt{cntu}[self]\ \textbf{then}$
      $head0 := \texttt{head}[self]\ ;$
      $\texttt{head}[self] := nil\ \rangle$
  fi
end .

Here we use an atomic **await** statement as described in (e.g.) [3,5]. Note that, indeed, *AwaitEdge* has the postcondition that $head0 \neq nil$ if $\texttt{cntu}[self]$ holds.

We assume that processes are numbered from $Process = [0\,..\,N-1]$. We distribute the tokens according to the rule that process p gets the tokens t with $p*M \leq t < (p+1)*M$. It follows that function *origin* is given by $origin(t) = t$ **div** M. Then the initialization is given by

procedure *Init2* $()=$
  var $t := self*M$;
  $head0 := nil$;
  for all $(x,y) \in edlis1(self)$ **do**
    $\texttt{next}[t] := head0$;
    $head0 := t$;
    $\texttt{pair}[t] := (x,y)$;
    $t := t+1$
  od ;
  $\texttt{ctok}[self] := t - self*M$;
  if $\texttt{ctok}[self] = 0$ **then** *GcDown* $()$ **fi**
end .

Here the assignments to `ctok` are not threatened by interference with *Down*, since the tokens from process q are not yet available to other processes. The use of two lists for every process enables us to treat the initialization of the processes as a private activity.

3.5 Using mutexes and condition variables

In this section, we implement the atomicity brackets and the **await** statement introduced in the previous section by means of mutexes and condition variables as specified in the POSIX thread standard, cf. [6, 43].

Mutexes serve to implement the atomicity brackets $\langle\,\rangle$. A mutex can be regarded as a record with a single field *owner* of type *Process*; $m.owner = \bot$ means that the mutex is free. The commands to lock and unlock a mutex m are given by

lock (m) : $\quad\langle$ **await** $m.owner = \bot$ **then** $m.owner := self\,\rangle$;
unlock (m) : $\quad\langle$ **await** $m.owner = self$ **then** $m.owner := \bot\,\rangle$.

The description of **unlock** may be slightly surprising: it enforces that only the owner of the lock is able to unlock it. A thread that tries to unlock a mutex it does not own, has to wait indefinitely. For every mutex m, we use the initialization $m.owner = \bot$. The commands **lock** and **unlock** are abbreviations of the `pthread_mutex_lock` and `pthread_mutex_unlock`, which are POSIX primitives.

After this preparation we come back to the synchronization of the program of the previous section. In order to allow maximal concurrency, we introduce several mutexes and arrays of mutexes for the protection of specific atomic regions. We introduce a mutex $\texttt{mtok}[q]$ to protect $\texttt{ctok}[q]$ and a mutex `mgc` to protect `gc`. We thus declare

```
mtok : array Process of Mutex ;
mgc : Mutex .
```

The procedures *Down* and *GcDown* become

```
procedure Down (q : Process) =
    var b : Boolean ;
    lock (mtok[q]) ;
    ctok[q] := ctok[q] − 1 ;   b := (ctok[q] = 0) ;
    unlock (mtok[q]) ;
    if b then GcDown () fi
end .

procedure GcDown () =
    var b : Boolean ;
    lock (mgc) ;
    gc := gc − 1 ;   b := (gc = 0) ;
    unlock (mgc) ;
    if b then Stop () fi
end .
```

We use condition variables for the implementation of the **await** construct in *AwaitEdge*. A variable v of type *Condition* is the name of a list $Q(v)$ of threads that are waiting for a signal. We only use the POSIX primitives `pthread_cond_wait` and `pthread_cond_signal`, abbreviated by **wait** and **signal**. These primitives are expressed by

```
wait (v, m) :
    ⟨ unlock (m); insert self in Q(v) ⟩ ;
    lock (m) .
```

Command **wait** consists of two atomic commands: to start waiting and to lock when released. Note that a thread must own the mutex to execute **wait**.

Command **signal** (v) is equivalent to *skip* if $Q(v)$ is empty. Otherwise, it releases at least one thread waiting at $Q(v)$. This is expressed in

```
signal (v) :
    ⟨ if not isEmpty(Q(v)) then release some threads from Q(v) fi ⟩ .
```

Notice that, when some thread signals v and thus releases a waiting thread, the latter need not be able to (immediately) lock the mutex. Some other thread may obtain the mutex first.

Back to the program. We introduce a mutex `gate`$[q]$ to protect `head`$[q]$ and `cntu`$[q]$ in the procedures *AwaitEdge*, *PutEdge*, and *Stop*. We introduce a condition variable `cv`$[q]$ to signal process q that the condition it may be waiting for has been established. We thus declare

```
gate : array Process of Mutex ;
cv : array Process of Condition .
```

The procedures *PutEdge* and *Stop* are translations of their counterparts in Section 3.4 extended with signals to the possible waiting processes.

```
procedure PutEdge (q : Process; x,y : V; t : Token) =
    pair[t] := (x,y) ;
    lock (gate[q]) ;
    next[t] := head[q] ;
    head[q] := t ;
    signal (cv[q]) ;
    unlock (gate[q])
end .

procedure Stop () =
    for all q ∈ Process do
        lock (gate[q]) ;
        cntu[q] := false ;
        signal (cv[q]) ;
        unlock (gate[q])
    od
end .
```

AwaitEdge is implemented in

```
procedure AwaitEdge () =
    if head0 = nil then
        lock (gate[self]) ;
        if head[self] = nil ∧ cntu[self] then
            wait(cv[self],gate[self])
        fi ;
        head0 := head[self] ;
        head[self] := nil ;
        unlock (gate[self])
    fi
end .
```

Note that, here, at most one process can be waiting at any condition variable. So, there is no danger that a signal releases more than one thread. On the other hand, since the waiting process is the only process that can invalidate it, the wait condition need not be tested again.

Remarks. If one removes the **lock** and **unlock** in *Stop*, the program becomes incorrect, since then a process, say q, may observe that the guard in *AwaitEdge* holds true and another process may falsify `cntu`$[q]$ and signal `cv`$[q]$ before q starts waiting.

It is also possible to implement the **await** construct in *AwaitEdge* by means of a split binary semaphore, see e.g. [3]. □

3.6 Harvest

After execution of algorithm A or its shared memory version, the connected components of the graph are determined by the function *root*. We collect this result in a separate array

$$\texttt{root} : \textbf{array}\ V\ \textbf{of}\ V\ .$$

In view of invariant $J1$, a sequential algorithm to do this is

B: **for all** $n \in V$ **do in increasing order**
  if $\texttt{par}[n] = n$ **then**
    $\texttt{root}[n] := n$;
  else $\texttt{root}[n] := \texttt{root}[\texttt{par}[n]]$ **fi**
od .

In this way, the connected components of the graph are characterized by a unique representing element, the root of the par tree. Loop B is very efficient, of order $O(\#V)$.

When the graph is very large, distributed harvesting may be indicated. To enable this, we decide that in harvest time all processes are allowed to inspect array par, but inspections and updates of $\texttt{root}[n]$ are only allowed for the owner of node n. We therefore write $V(p)$ to denote the set of nodes $n \in V$ with $owner(n) = p$ and we let the processes perform

C: **for all** $n \in V(self)$ **do in increasing order**
  if $\texttt{par}[n] \in V(self)$ **then**
    if $\texttt{par}[n] = n$ **then** $\texttt{root}[n] := n$
    else $\texttt{root}[n] := \texttt{root}[\texttt{par}[n]]$ **fi**
  else
    $r := \texttt{par}[n]$;
    while $r \neq \texttt{par}[r]$ **do** $r := \texttt{par}[r]$ **od** ;
    $\texttt{root}[n] := r$
  fi
od .

Fragment C has the inefficiency that root paths that leave $V(p)$ may be traversed repeatedly. We therefore introduce the following optimization. For each process, we declare a private variable *outList* of the type list of nodes with the invariant

$$J8: \quad x \in V(p) \quad \wedge \quad \texttt{par}[x] \notin V(p) \quad \Rightarrow \quad x \in outList.p\ .$$

We take $outList.p$ to be empty initially. Predicate $J8$ is preserved by program fragment A0. In order to preserve $J8$ during A1 and A2, we now let the assignments $\texttt{par}[x] := y$ in A1 and A2 be accompanied by the instruction to add x to the private *outList*.

We now first set all values of root to some reserved value $\perp$ and then determine the roots of the elements of *outList*.

D: **for all** $n \in V(self)$ **do** $\texttt{root}[n] := \bot$ **od** ;
for all $n \in outList$ **do**
$r := n$;
while $r \neq \texttt{par}[r]$ **do** $r := \texttt{par}[r]$ **od** ;
$\texttt{root}[n] := r$;
od .

After this loop, all points $x \in V(p)$ with $\texttt{par}[x] \notin V(p)$ have the correct value for $\texttt{root}[x]$, while the other points $x \in V(p)$ have $\texttt{root}[x] := \bot$. These remaining points of $V(p)$ are treated in the following loop.

E: **for all** $n \in V(self)$ **do in increasing order**
if $\texttt{root}[n] = \bot$ **then**
if $\texttt{par}[n] = n$ **then** $\texttt{root}[n] := n$
else $\texttt{root}[n] := \texttt{root}[\texttt{par}[n]]$ **fi**
fi
od .

Here, we use the invariant $J1$. Since the points are treated in increasing order, we have the invariant that **root**$[x]$ has the final value for all $x < n$. This property is preserved by the body of loop E because of $J1$. The composition D ; E is our final version of the harvest. This version is more efficient than version C, since only the root paths of points in *outList* are followed completely.

The list *outList* can be implemented most easily as a stack with maximal size equal to the number of boundary points of the set $V(p)$. Every element of $outList.p$ is an ancestor of a point of the boundary of $V(p)$, with all intermediate points within $V(p)$. This implies that $\#outList.p$ is bounded by the number of elements of the boundary of $V(p)$.

3.7 Application to Image Processing

In this section we focus on the application to image processing. We first consider a grey-scale image represented by a two-dimensional integer-valued array $im[H,W]$ (later we will consider 3-dimensional 'images' as well), where H and W are the height and the width of the image, respectively. The first coordinate (x) denotes the row number (scan line), while the second co-ordinate (y) denotes the number of the column. Since grey-scale images are discretizations of real black-and-white photographs there is an implicit underlying grid. We consider the case of 4-connectivity, meaning that pixels (except for boundary pixels) have four neighbours (north, east, south, west). Two neighbouring pixels that have the same image value, are considered to be in the same connected component. So, the graph considered has the pixels as nodes, and two pixels are connected iff they are neighbours *and* have the same image value.

Since the graph under consideration is derived from a rectangular grid, we can distribute it over the N processes by splitting it in equally sized slices. We have decided to distribute the image on the last coordinate of a pixel. It follows that we split the image in (almost) equally

sized vertical slices. The test $(x,y) \in V(p)$ becomes $lwb(p) \leq y < lwb(p+1)$, where *lwb* is given by

$$lwb(p) = (p \cdot W) \textbf{ div } N \ .$$

It follows that the corresponding function *owner* satisfies

$$owner(y) = ((N \cdot (y+1)) - 1) \textbf{ div } W \ .$$

The parallel algorithm consists of three phases. In the first phase, algorithm A0 is applied on the image slices. This is performed in a scan-line fashion, in which for pixel (x,y) only the pixels $(x-1,y)$ (north) and $(x,y-1)$ (west) need to be inspected.

In the middle phase, algorithm A2 is applied to edges of the graph which cross the boundaries of the distribution. In view of invariant *J*2, the list $edlis1(p)$ must contain the pixel pairs (x,y), $(x,y-1)$ with $y = lwb(p)$ for which $im[x,y] = im[x,y-1]$. It follows that H is an upper-bound for the length of the edge lists $edlis1(p)$. We can therefore take M of Section 3.4 to be equal to H.

Since we deal with images, a very effective optimization can be used to reduce the sizes of the buffers $edlis1(p)$. Indeed, we need not insert the pair (x,y), $(x,y-1)$ into $edlis1(p)$, if this list already contains the pair $(x-1,y)$, $(x-1,y-1)$ while also $im[x,y] = im[x-1,y]$. Indeed, if this is the case, the pair consists of pixels connected already, and we can therefore disregard the new edge. Experiments have shown that for camera made images this optimization often reduces the size of the buffers significantly. The optimization is used in the initialization *Init2*, while the remainder of A2 is left unmodified. In the final phase, we use the harvesting routine (D;E) to compute the output image.

The algorithm is easily adapted to 'images' of higher dimensionality. Apart from choosing another distribution, indexing, and the corresponding functions *lwb*, *owner* and *origin*, no modifications are necessary. We applied the algorithm to a 3-dimensional CT-scan data set $im[D,H,W]$, where D is the number of 2-D image slices (depth) of the data set. We used the same functions *lwb* and *owner*. In this case, the bound M of the sizes of the lists *edlis1* is $D \cdot H$.

Practical Results.

We applied the shared memory version of the algorithm on a set of seven 2-D test images. We had the availability of two shared memory architectures, namely a Cray J90 vector computer consisting of 32 processors and 4 Gb shared memory, and a Compaq ES40 with 4 Alpha-processors and 1 Gb shared memory. The processors of the Cray J90 computer are shared with other users, and the scheduling of these is done by the operating system, without any control to the user. It is therefore almost impossible to acquire 32 processors without interference by other users. For this reason, we decided to do time-measurements up to 16 processors, which turned out to be reasonably available. Each measurement was performed a hundred times, of which the 25 best and the 25 worst measurements were discarded. The remaining 50 measurements were averaged. This way we hope to get a reasonable measurement. On the Compaq ES40, measurements were performed simply 50 times and averaged immediately, since we were the only user on the system. The absolute timings on a single CPU are shown in Table 3.1. Note that the ES40 performs

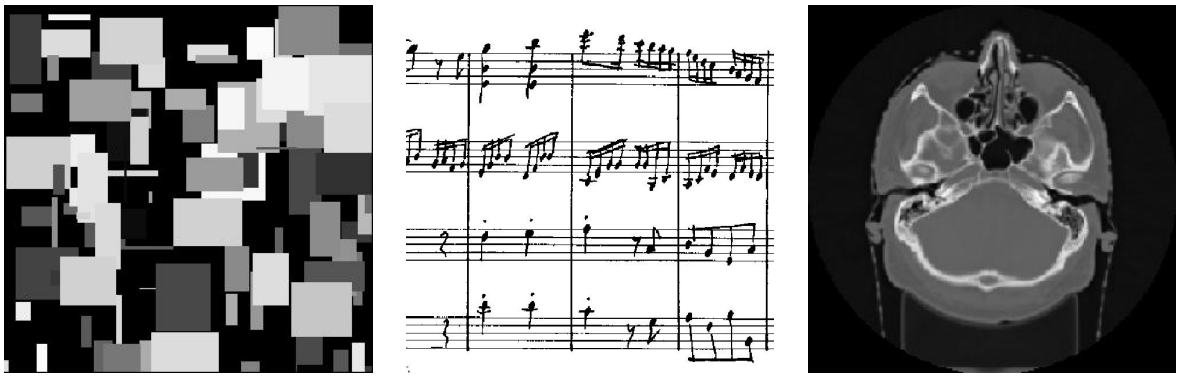


Figure 3.1. *Test images: (a) squares (b) music (c) CT*

	ES40			CRAY J90		
image	256	512	1024	256	512	1024
empty	10	42	172	381	1558	6260
vline	7	30	123	283	1145	4610
hline	6	28	114	287	1163	4801
comb	7	30	123	279	1139	4615
squares	10	42	173	374	1539	6228
music	10	42	172	361	1488	6080
CT	9	42	181	297	1370	5808

Table 3.1. *Absolute timings in milliseconds on a single CPU for different images sizes*

much better than the Cray. One may realize that the design of the Cray is more than 5 years older than that of the ES40, and that the Cray is a typical vector processor, which is not of any use in our algorithm. Besides, the Compaq has a cache memory on each processor of 512 kB, while the Cray has no cache whatsoever.

The image named *empty* is a trivial image of which all pixels have the same grey-value. The image *vline* is an image for which pixels $im[x,y] = 1$ if y is even, and $im[x,y] = 0$ if y is odd. The image *hline* is the image *vline* rotated over 90 degrees. The image *comb* is similar to the image *vline* except that the pixels on the last scan-line have grey-value 1, i.e. $im[H-1,y] = 1$. Clearly, these images are artificial images. We also used some more realistic images, which are shown in Fig. 3.1. The first image consists of 50 squares of random sizes, located at random positions. Each square has a unique grey-value. The second image is a camera-made image of handwritten music. The third image is slice 50 of a $93 \times 256 \times 256$ CT-scan of a head. The number of grey-values is reduced from 256 to 32 to reduce the influence of noise.

For the artificial images, path compression is extremely effective. For the more realistic images path compression is worthwhile, but is less effective. For these images, it turns out that the running time of the algorithm is hardly dependent on the content of the image. For most camera made images, the algorithm runs in approximately the same time.

In Tables 3.2 and 3.3, we see the speedup using more than one processor. The measurements

	256×256			512×512			1024×1024		
image	S_2	S_3	S_4	S_2	S_3	S_4	S_2	S_3	S_4
empty	1.7	2.0	2.3	1.8	2.3	2.7	1.9	2.6	3.2
vline	1.7	1.8	2.4	1.8	2.2	3.0	1.8	2.5	3.4
hline	1.4	1.4	1.3	1.7	1.9	2.0	1.8	2.4	2.7
comb	1.7	1.8	1.9	1.8	2.2	2.6	1.8	2.5	3.0
squares	1.7	2.0	2.2	1.8	2.3	2.7	1.9	2.6	3.2
music	1.5	1.7	1.8	1.7	2.2	2.4	1.9	2.6	3.2
CT	1.6	1.8	2.0	1.8	2.3	2.8	1.9	2.7	3.3

Table 3.2. *Speedups for the test set on the ES40.*

	256×256				512×512				1024×1024			
image	S_2	S_4	S_8	S_{16}	S_2	S_4	S_8	S_{16}	S_2	S_4	S_8	S_{16}
empty	1.9	3.5	5.7	6.4	2.0	3.7	7.1	11.0	2.0	3.9	7.8	14.1
vline	2.0	4.0	6.9	9.6	2.0	3.9	7.4	12.2	2.0	4.0	7.7	15.4
hline	1.8	3.3	4.5	3.6	1.9	3.6	6.1	7.6	2.0	3.8	7.3	11.4
comb	1.9	3.4	5.2	5.4	2.0	3.8	7.0	10.9	2.0	4.0	7.8	13.5
squares	2.0	4.0	6.4	6.6	2.0	3.9	7.4	11.4	2.0	4.0	7.9	14.0
music	2.0	3.8	5.9	6.2	1.9	3.8	7.3	10.5	2.0	3.9	7.9	14.1
CT	1.9	2.8	4.4	4.5	2.0	3.6	6.8	10.7	2.0	4.0	7.9	14.7

Table 3.3. *Speedups for the test set on the Cray J90.*

are performed on the test set for different image sizes. The number S_N is the speedup of the algorithm running on N processors relative to execution on one processor, defined by $S_N = T_1/T_N$, where T_N is the running time on N processors. We clearly see, that the speedup gets better if the computational task size increases. This is to be expected, since the ratio between computation and communication gets in favour of the computational side. This effect is especially severe on the ES40, since its processors are much faster than those of the Cray, while the memory speed (and thus communication speed) is about the same.

On both machines we see that the image *vline* performs best. Again, this is to be expected, since there is no communication needed at all. The image *hline* on the other hand performs worst, since here the amount of communication is maximal among the images considered. Even for this case, however, the speedups are satisfactory. The image *empty* gains most from the optimization mentioned above. For this image, the lists *edlis1* initially contain each only a single pair.

For the more realistic images *square*, *music* and *CT*, we see very nice results. This is of course the main goal of the algorithm. For large enough images, up to about 8 processors we see an almost linear speedup. If we add more processors, we see a slight drop in the efficiency as a result of relative increase of communication with respect to the computational task. However, an efficiency of generally more than 75% is very satisfactory.

We also applied the 3-D version of the algorithm to a CT data set with sizes $93 \times 256 \times 256$. In Fig. 3.1(c), we see slice 50 of this set. The amount of grey-values was reduced from 256 to

ES40		CRAY J90									
N	S_N	N	S_N	N	S_N	N	S_N	N	S_N	N	S_N
2	**1.8**	2	**2.0**	5	**4.6**	8	**7.2**	11	**9.2**	14	**10.9**
3	**2.4**	3	**2.9**	6	**5.6**	9	**7.9**	12	**9.9**	15	**11.5**
4	**3.1**	4	**3.8**	7	**6.4**	10	**8.6**	13	**10.5**	16	**12.7**

Table 3.4. *Speedups for the 3-D CT data set.*

32 grey-values. In Table 3.4, we present the results for both architectures. The left-hand frame contains the results on the ES40, with $T_1 = 3.1$ seconds. The right-hand frame contains the results on the Cray J90, with $T_1 = 149$ seconds. The results show the same tendencies as the two-dimensional results.

3.8 Conclusion

The computation of the connected components of an image (2-D or 3-D) can effectively be distributed over a number of processors. The amount of communication needed can only be determined at runtime, but is for most natural images quite modest. We used a variation of Tarjan's connected components algorithm. The communication is based on message passing, but implemented in shared variables by means of POSIX thread primitives. The experiments show a speedup that is often almost linear in the number of processors.

This chapter has been first published as: A. Meijster and M. H. F. Wilkinson, A comparison of algorithms for connected set openings and closings, *IEEE Trans. Patt. Anal. Mach. Intell.*, **24** (4), pp. 484-494, 2002.

Chapter 4

A Comparison of Algorithms for Connected Set Openings and Closings

Abstract

The implementation of morphological connected set operators for image filtering and pattern recognition is discussed. Two earlier algorithms based on priority queues and hierarchical queues respectively are compared to a more recent union-find approach. Unlike the earlier algorithms which process regional extrema in the image sequentially, the union-find method allows simultaneous processing of extrema. In the context of area openings, closings and pattern spectra, the union-find algorithm outperforms the previous methods on almost all natural and synthetic images tested. Finally, extensions to pattern spectra and the more general class of attribute operators are presented for all three algorithms, and memory usages are compared.

4.1 Introduction

In mathematical morphology, connected set operators [39, 84] form a versatile class of image operators with a number of desirable properties, most importantly preservation of shape. The earliest members of this class were openings and closings by reconstruction, for which efficient algorithms have been developed [100]. In the binary case, an opening by reconstruction first performs an erosion with some structuring element, and then reconstructs all connected foreground components which were not completely removed by the erosion. Therefore, openings by reconstruction (and the corresponding closings) can either remove image details completely, or leave them intact, but never alter their shape.

An important development was the introduction of area openings and closings; a phrase coined by Vincent [99], though they were introduced by Cheng and Venetsanopoulos a year earlier [21] as NOP, and NCP operators. Vincent's algorithm was much more efficient in the grey scale case than the earlier method of Cheng and Venetsanopoulos. In the binary case, area openings remove all connected *foreground* components with an area smaller than some threshold λ. Binary area closings fill all *background* components with an area smaller than λ. Figure 4.1 shows a comparison of applying binary openings by a 7×7 pixel square structuring element, opening by reconstruction by the same structuring element, and and an area opening with $\lambda = 49$

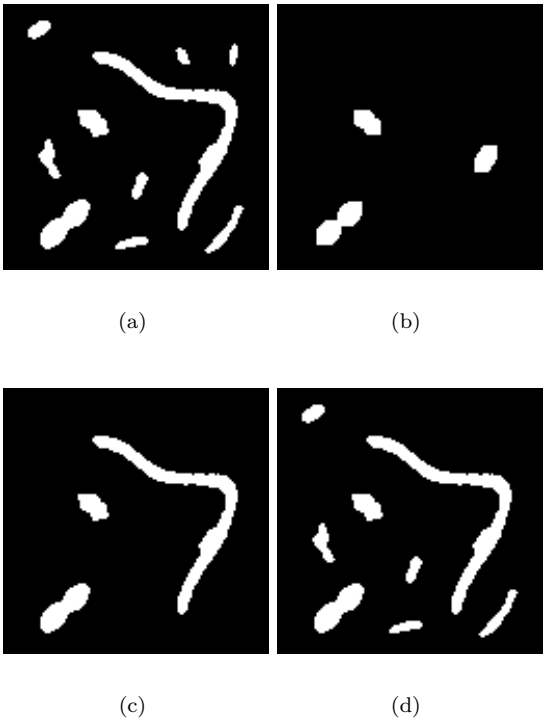


Figure 4.1. *Area versus structural openings: (a) a binary image of 128 × 128 pixels; (b) structural opening with 7 × 7 structuring element; (c)opening by reconstruction with same structuring element; (d) area opening with* $\lambda = 49$*. Neither openings by reconstruction nor area opening change the shapes of connected foreground components, but the former uses width, and the latter area as selection criterion*

of a binary image of bacteria, showing the differences in the action of these filters. An area opening tends to retain long thin objects more than an opening by reconstruction.

Area openings and closings were later extended to the wider class of attribute openings, closings, thickenings and thinnings [19,83]. Breen and Jones [19] extended Vincent's priority-queue algorithm for area opening to attribute openings and thinnings, whereas Salembier et al. [83] developed a new algorithm based on hierarchical queues. The latter also introduced a versatile data structure, dubbed a Max-tree (and its dual the Min-tree), which reduces image filtering to removing nodes from a tree. Attribute openings allow the use of size criteria other than width (used by openings by reconstruction) and area (used by area openings). In the binary case, attribute openings can be made by computing some increasing attribute (such as moment of inertia, diagonal of the smallest enclosing rectangle, etc.) of each connected foreground component, and remove the components for which the attribute is smaller than the threshold. An attribute A is increasing if and only if for all sets C and D with $C \subseteq D$, $A(C) \leq A(D)$.

Attribute thinnings work on a similar principle, but can use shape rather than size criteria. In this case non-increasing attributes, such as the ratio of the square of the perimeter to the area, can be used. This type of filter allows extraction of all image details of a given shape, regardless

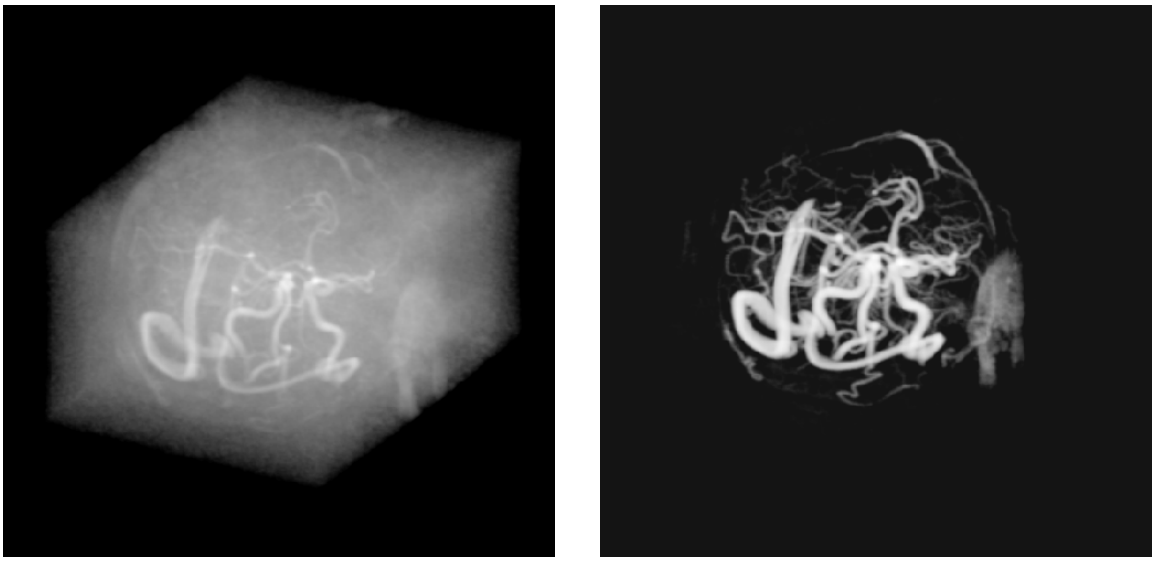


Figure 4.2. *An application of connected set filters (grey-scale attribute thinning) to filament extraction: (left) volume rendering of magnetic resonance angiography* 256^3 *volume data set; (right) result using an attribute thinning as shape filter. The attribute used was* $I/V^{5/3}$*, with* I *the moment of inertia, and* V *the volume of a peak component; the attribute threshold was 2.0.*

of their sizes [92]. A 3-D application of this is the extraction of filamentous details (vessels) from angiograms [104]. An example is shown in Figure 4.2. Other applications of connected set morphology include filtering [21, 99] and segmentation [24].

Finally, connected set filters are important for multi-scale morphology, and in particular non-linear scale spaces. These cannot be constructed from openings or closings by structuring elements in more than one dimension, due to problems with causality [42]. Therefore, Bangham et al. [8, 9] extended their 1-D non-linear sieves [7, 10], to an arbitrary number of dimensions using area openings and closings, and their higher-dimensional counterparts such as volume openings and closings. They show that because connected set filters never introduce new edges, they do not violate causality in any number of dimensions.

The aim of this chapter is to review algorithms for connected set openings and closings, and in particular attribute openings and closings. The two best known algorithms for these image filters are the pixel priority-queue method introduced by Vincent for area operators [99] and extended by Breen and Jones to attribute operators [19], and the Max-tree method of Salembier and co-workers [83] which uses hierarchical queues. These two methods will be compared to an algorithm developed by the authors, which is based on the classical union-find algorithm presented by Tarjan [88]. We will focus on the area opening at first since this is one of the simplest cases. Besides, it can be extended to the more general case of attribute operators [103] in much the same way as Breen and Jones extended Vincent's algorithm [19]. To demonstrate the versatility of the union-find method, we will show how the same principles can be applied to computation of area pattern spectra. The key difference between the algorithm based on union-find and the earlier methods of Vincent [99] and Salembier et al. [83] is that we process multiple maxima simultaneously, rather than sequentially. We show how this may lead to a

parallel implementation of the algorithm. Both earlier algorithms are described briefly in this chapter, followed by a more detailed description of the union-find approach. Their respective performances and memory usages are compared on a range of images.

4.2 Theory

4.2.1 Connected Set Operators

The distinguishing characteristic of connected set operators is that they operate on the flat zones of images, rather than on individual pixels [84]. Let $\{\alpha_i\}$ be the partition of the domain M of image I formed by its flat zones α_i, and $\{\beta_j\}$ the partition of M formed by the flat zones β_j of image $\gamma(I)$, with γ some image operator. A partition of M is any set of disjoint sets $\{S_i\}$ (i.e. $S_i \cap S_j = \emptyset$ for any $i \neq j$), which together form the entire set M, i.e. $\cup S_i = M$.

The operator γ is said to be a connected set operator if and only if the partition $\{\alpha_i\}$ is finer than $\{\beta_j\}$, i.e. for all sets α_i there exists a β_j such that $\alpha_i \subseteq \beta_j$. Thus, the only operations connected filters can perform is merging flat zones, and assigning new grey levels to them. Because flat zones of an image are disjoint sets, and set-union of these sets is a key action taken by connected set operators, we propose to implement connected filters using Tarjan's union-find algorithm [88]. This is an efficient algorithm for maintaining *disjoint sets* under the set-union operation, and has been adapted to connected component labeling and flat zone labeling [29,32,40]. Other applications of union-find in image processing include the computation of the watershed transform [53].

In the next sections we will focus on how anti-extensive connected set filters, in particular attribute openings, are implemented in three previously published algorithms. For the sake of simplicity we will concentrate on area openings. These serve as a simple case to compare the efficiencies of the three approaches.

4.2.2 Area Openings and Closings

The theory of area operators is given only briefly here. For a more thorough discussion the reader is referred to [99]. We will first discuss binary area openings and closings, and then the extension to the grey scale case. Binary area openings are based on binary connected openings. Let the set $X \subseteq \mathbf{M}$ denote a binary image with domain $\mathbf{M}$. The binary connected opening $\Gamma_x(X)$ of X at point $x \in \mathbf{M}$ yields the connected component of X containing x if $x \in X$, and $\emptyset$ otherwise. Thus Γ_x extracts the connected component to which x belongs, discarding all others. Let $A(X)$ be the area of a set $X \subseteq M$. In the case the case of digital images this is usually the number of pixels in X. The binary area opening can now be defined as:

Definition 1 *Let $X \subseteq \mathbf{M}$, $\lambda \geq 0$. The binary area opening of X with parameter λ is given by*

$$\Gamma^A_\lambda(X) = \{x \in X | A(\Gamma_x(X)) \geq \lambda\} \tag{4.1}$$

The binary area closing can be defined by duality:

$$\Phi_\lambda^A(X) = [\Gamma_\lambda^A(X^c)]^c \tag{4.2}$$

The definition of an area opening of a grey scale image f is usually derived from binary images $T_h(f)$ obtained by thresholding f at h. These are defined as:

$$T_h(f) = \{x \in \mathbf{M} | f(x) \geq h\} \tag{4.3}$$

Definition 2 *The area opening for a mapping $f : \mathbf{M} \to \mathbb{R}$ is given by:*

$$(\gamma_\lambda^A(f))(x) = \sup\{h | x \in \Gamma_\lambda^A(T_h(f))\}. \tag{4.4}$$

The grey scale area closing φ_r^A is defined by using a duality relationship similar to (4.2)*:*

$$\varphi_\lambda^A(f) = -\gamma_\lambda^A(-f). \tag{4.5}$$

Thus, the area opening of an image assigns each point the highest threshold at which it still belongs to a connected foreground component of area λ or larger. The area closing assigns each point the lowest threshold at which it belongs to a connected background component of area λ or larger.

4.3 Computing area openings and closings

Before describing the individual algorithms, we first define a *flat zone* L_h at level h of a grey scale image f as a connected component of the set of pixels $\{p \in \mathbf{M} | f(p) = h\}$. A *regional maximum* M_h at level h is a flat zone no members of which have neighbors larger than h. A *peak component* P_h at level h is a connected component of the thresholded image $T_h(f)$. At each level h there may be several such components, which will be indexed as L_h^i, M_h^j and P_h^k, respectively, with i, j, and k from some index set. Any regional maximum M_h^j is also a peak component, but the reverse is not true.

4.3.1 The pixel-queue algorithm

The pixel-queue algorithms for area and attribute operators are given in some detail elsewhere [19,98,99], so we will describe them only briefly here. The versions described here only perform area openings, but more general attribute openings require little modification [19].

Briefly, the image is first scanned using a pixel queue to create a list of all regional maxima M_h^k. After this, all M_h^k are processed sequentially. This is done by growing a peak component $P_{h_1}^j, h_1 \leq h$ around a seed pixel within the maximum M_h^k using a priority queue. As each pixel is added to the growing region, its neighbors which do not (yet) belong to the region are put in the priority queue, from which they are retrieved in reverse grey level order (highest first). The process of adding pixels continues until the area is larger than λ or when the next pixel taken

```
/* List F contains the local maximum components */
while (F not empty) do
  {
     extract C from F;
     area =  A(C);
     curlevel = grey level of component;
     while (area < lambda)
       { n = neighbor of C with I[n]
             is maximum of all neighbors;
         if (I[n] > curlevel)
           break;
         else { add n to C;
                curlevel = I[n];
              }
       }
     for all p in C do
       {  I[p] = curlevel;
          L[p] = PROCESSED;
       }
  }
```

Figure 4.3. *The core of Vincent's area opening algorithm. The parameter* `lambda` *in the code equals the area threshold* λ*. For closing, the local minima components are scanned, the minimum neighbor of C must be sought, and the test before the break must be* $I[n] < curlevel$.

from the priority queue has a grey level h'' larger than the current level h'. In the first case the peak component is large enough, and should be set to the current grey level h'. In the latter case the region grown so far is not a peak component $P^j_{h'}$ *at level* h'. Note that in this case the component may be flooded again later on. In either case the grey levels of all pixels $p \in P^j_{h'}$ are set to h', and M^k_h is removed from the list. The algorithm terminates when all maxima have been processed.

The part of the algorithm after the regional maxima have been found is shown in pseudo code in Figure 4.3. Note that in the code fragments of this chapter the grey level of a pixel (node) `n` is denoted by `I[n]`.

4.3.2 The Max-tree approach

Max-trees were introduced by Salembier et al. [83] as a versatile data structure for anti-extensive connected set operators. A Max-tree is a rooted tree, in which each of the nodes C^k_h corresponds to a peak component P^k_h. However, C^k_h contains only those pixels in P^k_h which have grey level h. In other words, it is the union of all $L^j_h \subseteq P^k_h$. Each node except for the root points towards its

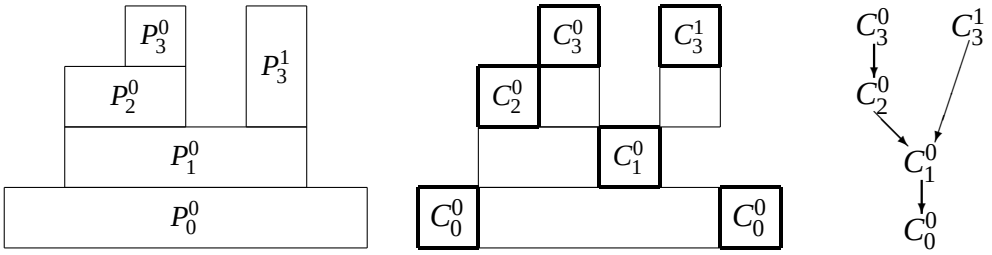


Figure 4.4. *The Max-tree structure: the peak components P_h^k of a 1-D signal level image X (left); the corresponding pixels in each Max-tree node C_h^k (middle); and the Max-tree itself (right).*

parent $C_{h'}^j$ with $h' < h$. An example of a 1-D signal, its peak components and its Max-tree nodes is shown in Figure 4.4. It can be seen that multiple flat zones may be contained in a single node, as in the case of C_0^0 in this example. Furthermore, in a given branch, not all grey levels have to be occupied, as in the case of the branch ending at C_3^1.

In the case of the area opening, the area of the peak component P_h^k is stored in the Max-tree structure in node C_h^k. Once a Max-tree of an image has been computed, computing an area opening reduces to removing all nodes which have an area smaller than λ from the tree. Because the area must increase when descending any branch towards the root, the opening reduces to pruning branches at the appropriate points, and merging the pixels in the removed nodes with their smallest surviving ancestor.

Salembier et al. [83] use an array `STATUS` of the same size as the image to determine to which node a pixel belongs. A pixel `p` with grey level h belongs to node C_h^k if `STATUS[p]`$=$ k. Initially, all elements in `STATUS` are set to `NOTPROCESSED`(< 0). Besides this array, the algorithm uses a hierarchical queue of G levels, with G the number of grey levels. One array of G integers called `NodesAtLevel` is used to store the number of Max-tree nodes detected so far at each grey level. A further array of G booleans `NodePresent` stores at which levels below the current grey level nodes have been detected in the path from current node to root. Finally, the Max-tree itself is stored in an array of N nodes (with N the number of pixels in the image).

In our implementation we first compute a histogram of the image, in order to compute the space needed in each of the levels of the hierarchical queue, and maximum possible number of nodes in each of the levels of the Max-tree. All elements of the `NodesAtLevel` array are set to zero, those of the `NodePresent` array to false. Then, the Max-tree is built by first selecting a pixel with the smallest grey level `hmin` which must belong to the root of the tree, and inserting it at level `hmin` in the hierarchical queue, and `NodePresent[hmin]` is set to true. Using this pixel as a seed a recursive flood filling process is started. At each step a pixel `p` at the highest occupied level `h` in the hierarchical queue (starting at `hmin` obviously) is retrieved. Then `STATUS[p]` is set to `NodesAtLevel[h]`, which labels it as a member of C_h^k, and the area in the appropriate Max-tree node is incremented. Then, all neighbours of `p` for which `STATUS[n]` is `NOTPROCESSED` are put in the queue at level `I[n]`, `STATUS[n]` is set

to INTHEQUEUE(< 0), and NodePresent[I[n]] is set to true. If the neighbour's grey level I[n] is larger than the current level h the flooding proceeds at level I[n]. If at a given level h, no more pixels can be retrieved from the queue at level h, NodePresent[h] is set to false, and the parent node is sought by finding the element in NodePresent with the highest index m which is true. Once m has been determined, the parent node $C^j_{h'}$ is found by letting $h' =$m, and $j =$NodesAtLevel[m]. The current nodes area is added to its parent, and flooding proceeds at level m. Once all pixels have been processed, the Max-tree is complete.

The filtering is carried out by visiting all nodes of the tree once from low grey level to high. We can do this because we store the nodes sorted per grey level in an array, and we know how many nodes exist at each level. For each node we check whether its area is smaller than λ. If so, its output grey level is set to that of its parent (which has already been assigned the correct grey level). The output image O is made by visiting all pixels p in the image, determining its node from I[p] and STATUS[p], and assigning the output grey level of that node to O[p].

4.3.3 The Union-Find Method

A key property of the algorithms described above is that they process the image one peak component at a time. Furthermore, both use a flooding method. The approach described in this section can process multiple peak components simultaneously. Pixels are processed in grey level order. During this process, peak components are created and merged as needed, while keeping track of their areas. Once a peak component has an area of at least λ, it ceases to grow.

Tarjan's union-find algorithm for keeping track of disjoint sets [88] is used to implement merging in an efficient way. For each set an arbitrary member is chosen as representative for that set. The algorithm uses rooted trees to represent sets, in which the root is chosen as the representative. Each non-root node in a tree points to its parent, while the root points to itself. Two objects x and y are members of the same set if and only if x and y are nodes of the same tree, which is equivalent to saying that they share the same root of the tree they are stored in. There are four basic operations.

- MakeSet(x): Create a new singleton set {x}. This operation assumes that x is not a member of any other set.
- FindRoot(x): Return the root of the tree containing x.
- Union(x,y): Form the union of the two sets that contain x and y.
- Criterion(x,y): a symmetric criterion which determines whether x and y belong to the same set.

For flat zone labeling the algorithm becomes:

```
for all pixels p do
  { MakeSet(p);
    for all neighbors n<p do
```

```
    if ( I[n]==I[p] )
      Union( n, p );
}
```

Note that in this context the condition `n<p` means that `n` is a pixel which has been processed before `p`. In this case `Criterion(n,p)` is true if the image value `I[n]` equals `I[p]`. `Union` uses `FindRoot` internally to determine the root nodes of the trees containing `n` and `p`. After this scan, a second "resolving" scan assigns each root pixel a unique label, and to each non-root pixel the label of its root.

Before going into the details of the area opening algorithm itself, we will discuss the general framework for storing the disjoint sets, and the auxiliary functions needed for attribute openings and closings.

The disjoint sets we have to find are all flat zones $L_h^i \subset \Gamma_\lambda^A(X_h(f))$, which are not altered by the area opening γ_λ^A, and, for all other L_h^i, the smallest peak component $P_{h'}^j \supset L_h^i$ which has area λ or more. To store the trees representing these sets for the entire image, we use an integer array `parent` of the same size as the image (i.e., N), in which `parent[p]` is the parent of pixel `p`. Pixels are stored as `width*y+x`,with `x` and `y` the pixel's x and y coordinates, and `width` the image width. If a pixel is a root of a tree, i.e. it has no parent, we flag this by setting `parent[p] < 0`, rather than letting it point to itself. We could use an auxiliary array `area`, in which `area[p]` (for root nodes) stores the area of each set. However, for a set of area A with root node `p`, we can also set `parent[p]` to $-A$, which saves memory space. We define an *active root* as a root of a peak component with area smaller than λ.

The code for the `MakeSet`, `FindRoot`, `Criterion`, and `Union` routines is shown in Figure 4.5. In this case, the `Criterion` and `Union` routines are asymmetrical. This is done to ensure that if the set we are dealing with is a peak component P_h at level h, the root element `r` has a grey level `I[r]`$= h$. Therefore, we process the pixels in decreasing grey level order, and always make the last pixel processed the root of the new tree. We do this by sorting the pixels using counting-sort, and storing the coordinates in an array `SortPixels` of length N. Pixels of the same grey level are processed in scan line order. Scanning of peak components from high to low grey levels is guaranteed, without finding regional maxima explicitly.

As each pixel `p` is processed, the `MakeSet` routine labels `p` as a singleton set, setting `parent[p]` to -1. The `Union` procedure is now called for each neighbor `n` which has already been processed. We briefly describe this procedure here. Since `p` is *always* a root, `FindRoot` is only called to find the root pixel `r` of `n`. If `r` equals `p` nothing needs to be done, because `n` and `p` are already in the same set. Otherwise, `Criterion` is called with `r` and `p` as parameters. If the grey level `I[r]` of `r` is equal to that of `p` *or* if `r` is an active root, `Criterion` returns "true" and the two trees are merged. Merging is done by adding the area of `r` to that of `p`, and making `p` the parent of `r`. If `Criterion` returns "false", a neighbor has a root grey level higher than `I[p]` *and* has a sufficiently large area, and therefore `p` $\in L_h^i \subset \Gamma_\lambda^A(X_h(f))$. In this case, `p` is made inactive by setting `parent[p]` to $-\lambda$.

Note that the `FindRoot` routine, apart from finding the root of a tree, performs path compression. This is a technique that was used by Tarjan to reduce the average cost of `FindRoot`.

```
void MakeSet ( int x )
{ parent[x] = -1;
}

int FindRoot ( int x )
{ if ( parent[x] >=0 )
    { parent[x] = FindRoot( parent[x] );
      return parent[x];
    }
  else return x;
}

boolean Criterion ( int x, int y )
{ return ( (I[x] == I[y]) ||
           ( -parent[x] < lambda ) ) ;
}

void Union ( int n, int p )
{ int r=FindRoot(n);
  if ( r != p )
    { if ( Criterion(r, p) )
        { parent[p] = parent[p] + parent[r];
          parent[r] = p;
        }
      else
        parent[p] = -lambda;
    }
}
```

Figure 4.5. *Implementation of the basic operations for area openings and closings. Note that the areas of components are stored as negative numbers in the corresponding roots. The variable* `lambda` *is equal to the parameter* λ*. The parameters for* `Criterion` *must be root nodes.*

It does this by setting the parent pointer directly to the root for all pixels visited along the path to the root.

At the end of this part of the algorithm, we have found two kinds of disjoint sets: (i) those with constant grey level, which are flat zones $L_h^i \subset \Gamma_\lambda^A(X_h(f))$, and (ii) those with varying grey level, which are peak components P_h^j, with h the maximum grey value for which the area criterion is satisfied. Because the root r of these peak components is always the last pixel processed, its grey level in the input image satisfies $f(r) = h$. Therefore, if we set the grey level of each pixel in the output image to that of its root in the input image, all $L_h^i \subset \Gamma_\lambda^A(X_h(f))$ remain unchanged,

```
/* array S contains sorted pixel list */
for (p=0; p<Length(S); p++)
  {
    pix = S[p];
    MakeSet(pix);
    for all neighbors nb of pix do
      if ((I[pix] < I[nb]) ||
          ((I[pix] == I[nb]) && (nb<pix)))
        Union(nb,pix);
  }
/* Resolving phase in reverse sort order */
for (p=Length(S)-1; p>=0; p--)
  {
    pix = S[p];
    if (parent[pix] >= 0)
        parent[pix] = parent[parent[pix]];
    else
        parent[pix] = I[pix];
  }
```

Figure 4.6. *Code showing how to perform an area opening using the operations of Fig. 4.5.*

whereas all P_h^j are filled uniformly with a grey level of h. This can be done in linear time. The most memory efficient approach is to store the output image in the `parent` array, which is possible because we can visit the pixels in reverse processing order. For each pixel `p` we inspect its parent. If `parent[p]` is negative, `p` is a root, and `I[p]` is the correct grey level for the component. If `parent[p]` is non-negative, then it is pointing to a pixel in `parent` which has already been resolved, and already has been assigned the correct root grey level. In figure 4.6 the above described algorithm is listed.

For area closings we must change each test for grey level in the main loop of the routine to a test for *smaller than*. The `FindRoot` and `Criterion` procedures, and the resolving stage need not be changed.

4.4 Performance testing and results

The three algorithms were implemented in C. For efficiency reasons tail recursion in `FindRoot` was removed, and all procedures in Figure 4.5 were inlined. The code is available on request. The algorithms were tested on a range of synthetic images with different sizes, and numbers of extrema. Furthermore, a number of natural images at various sizes was used. Mean CPU times over multiple runs on HPPA RISC-processor based machines were determined for each image and for different values of λ.

Three-dimensional versions of the algorithms for volume openings were also implemented and tested on a number of angiograms.

4.4.1 Synthetic images

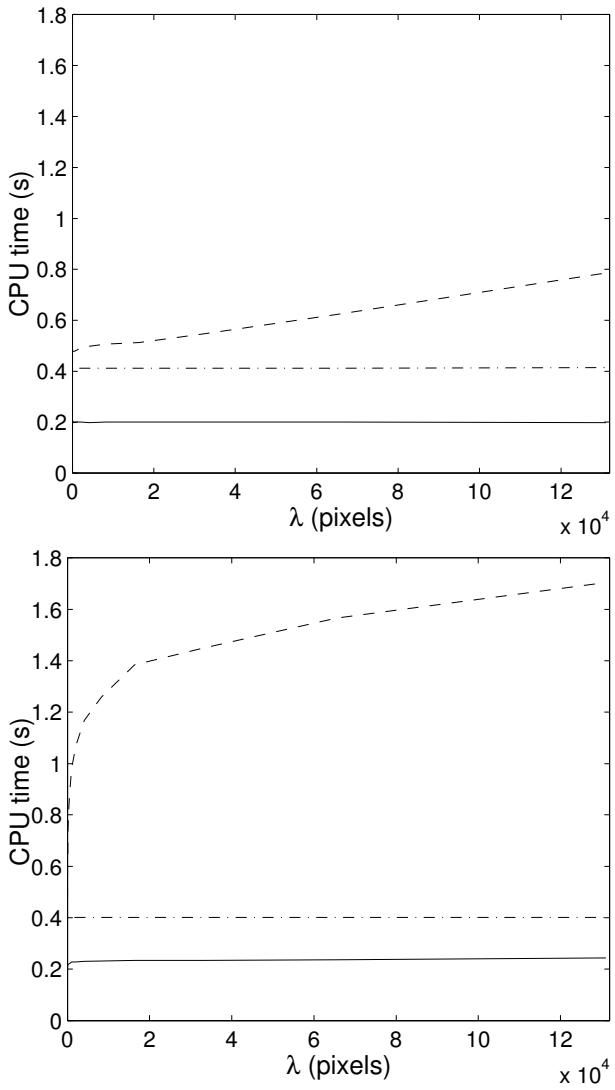


Figure 4.7. *Timing results for distance map images. CPU timing for the priority-queue algorithm (dashed) and the union-find method (solid), and the Max-tree algorithm as a function of area (λ) of the closing for* 512×512 *distance map images for different numbers* N_{min} *of local minima: (top)* $N_{min} = 2$*; (bottom)* $N_{min} = 2000$*. Note the strong dependence of the timings of the priority-queue algorithm on both image content (*N_{min}*) and* λ*. The timings for the union-find and Max-tree methods are practically independent of* λ*, and only slightly dependent on* N_{min}*.*

To test the sensitivity of the algorithm to the image complexity in a controlled manner, syn-

thetic images were used. To generate these, N_{min} randomly placed dots on a black background were generated. After this Euclidean distance maps were generated from these random dot images. These distance map images allow controlled testing of the performance of each algorithm as a function of the number of extrema (minima) of roughly circular shape. Because the distance map images contain controllable numbers of minima, we decided to perform the timings on closings rather than openings. However, we could in principle have inverted these images and done the timings on openings, obtaining the same results.

The CPU times for area closings with increasing λ for 512×512 distance map images with different numbers of local minima M are shown in Figure 4.7. Quite clearly, the CPU times for the priority-queue algorithm depend strongly on both λ and N_{min}, unlike either the union-find method or the Max-tree method. In all cases, the union-find method outperformed the priority-queue algorithm, by a factor ranging from 2.4 for small λ, to about 7 for $\lambda = 131072$ and $N_{min} = 2000$. The λ dependency of the priority queue algorithm is clearly nonlinear for smaller λ. By contrast, the λ dependence of the modified Tarjan method is very weak, rising by at most 17% over λ ranging from 2 to 131072. The Max-tree algorithm has no significant λ-dependence. However, it is slower than the union-find approach by a factor of 2.1 at small λ and 1.65 at $\lambda = 131072$ for $N_{min} = 2000$. For the union-find method timings range from 0.200 s to 0.244 s from $N_{min} = 2$ to $N_{min} = 2000$, or an 22 % increase for $\lambda = 131072$. The Max-tree method shows no significant change over the same range. For $\lambda = 131072$, Vincent's algorithm shows an increase in CPU time from 0.786 s to 1.704 s, or a 117 % increase, over the same range of N_{min}.

To measure dependence of CPU times on the image size N, distance map images of different sizes with the same density α of minima ($\alpha = N_{min}/N = constant$) were used. The results are shown in Figure 4.8. All three methods appear to be linear in N for all practical purposes. On certain machines deviations were observed for very small images, probably due to caching.

4.4.2 Natural images and volume openings

A selection of 40 natural images with different image characteristics was used to assess the performance under more realistic conditions. The images include images of faces, a house, landscapes, airplanes, an aerial photograph of a city, various astronomical images, and microscopic images of diatoms, bacteria and skin tissue. Some of these images are shown in Figure 4.9.

The results are similar to those for synthetic images. CPU times for the union-find and Max-tree methods depend mainly on image size, and only slightly on image content or λ. By contrast, CPU times for the priority-queue algorithm depend strongly on both image content and λ. The image of Jupiter's red spot in Figure 4.9(b) shows a very strong λ-dependence at all scales. This may be caused by the many nested structures in this image.

In certain images, such as the image of bacteria in Figure 4.9(c), the dependence of CPU time on λ is only slight. Indeed, based on this image alone there would be little reason to prefer any algorithm. However, even here, the union-find method is faster than the others for all $\lambda > 8$. In this case alone, it is the Max-tree method which is slowest, except for $\lambda > 16384$.

The volume openings yielded slightly different results. On a $128 \times 128 \times 62$ volume, the union-find method (CPU time 1.29 s) is 3.5 times faster than the Max-tree (4.51 s), and 3.3 times faster than the priority-queue method (4.26 s), both for $\lambda = 256$. At $\lambda = 131072$, the union-

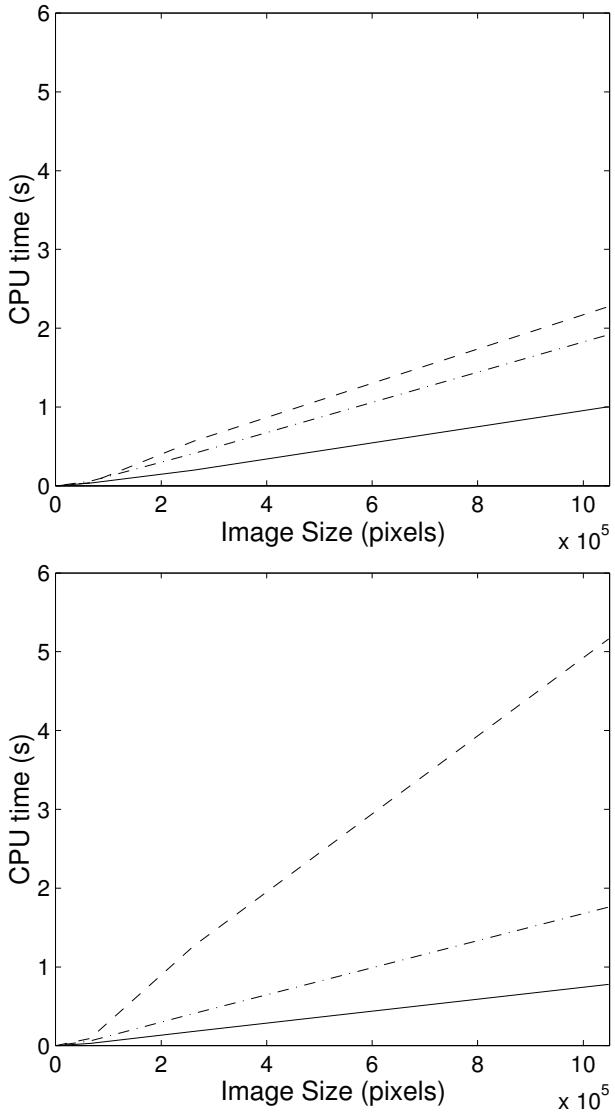
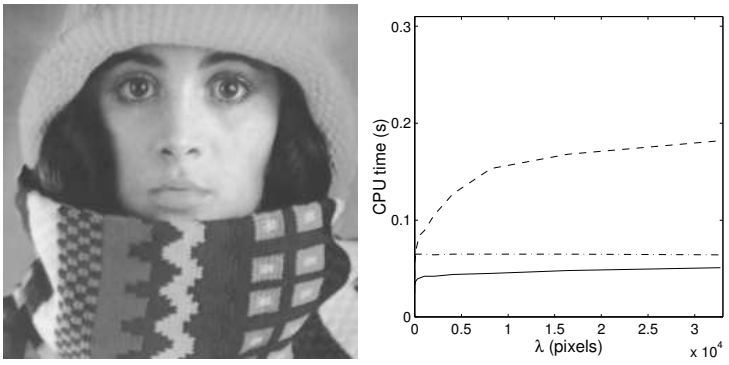
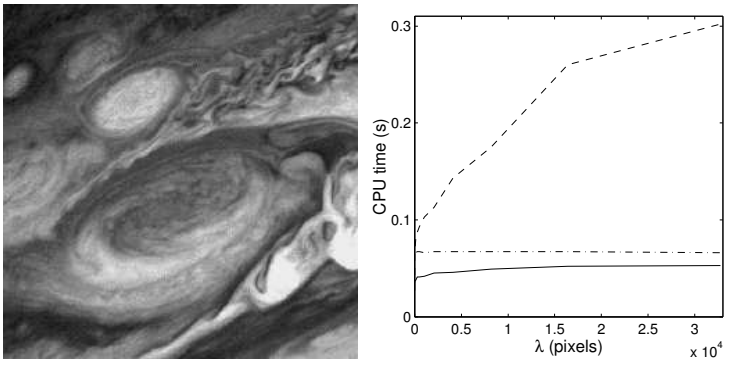


Figure 4.8. *Image size dependence of computing times for distance map images for the pixel-queue (dashed), Max-tree (dash-dot), and union-find (solid) algorithm for* $\lambda = 256$ *and different values of the density* α *of local minima: (top)* $\alpha = 1/1024$*, (bottom)* $\alpha = 100/1024$*.*

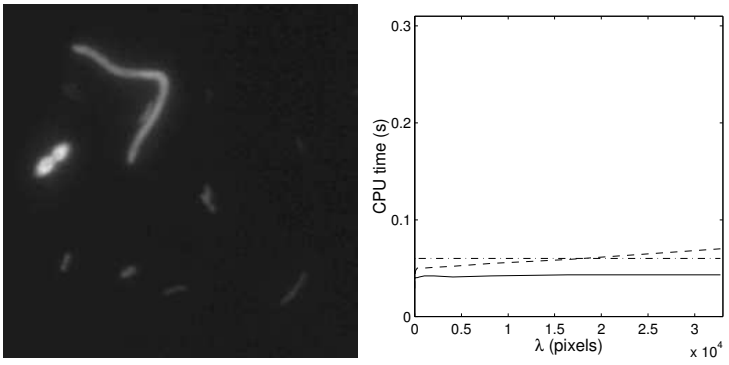
find method takes 1.34 s, Max-tree algorithm 4.53 s, and the priority-queue method falls far behind with 72 s. We performed the same tests on the $256 \times 256 \times 256$ volume of Figure 4.2 on a Pentium II at 400 MHz, with 640 MB RAM. At $\lambda = 256$ and $\lambda = 131072$ respectively, the union-find method needs 6.35 s and 6.68 s, the Max-tree algorithm needs 20.37 s and 20.47 s, and the priority-queue method needs 19.01 s and 108 s. Apparently, the advantage of the union-find algorithm is greater in these 3-D data sets. This is probably due to its smaller memory requirements, especially given the limited bandwidth of the memory bus of personal computers, as is discussed in the next section.



(a)



(b)



(c)

Figure 4.9. *Some natural images used for performance testing algorithms for area closings and the corresponding results for the Max-tree algorithm (dash-dot), priority-queue (dashed) and the union-find approach (solid): (a) face, (b) red spot, and (c) bacteria.*

4.5 Computational Complexity and Memory Use

In the case of the union-find approach, complexity analysis is straightforward if we assume that the union-find part is the most costly. This is the case when we can use a linear time sorting algorithm as counting-sort which only works well for 8-16 bit integers. Even though $O(N)$ sorting algorithms exist for floating point numbers, or larger integer ranges (e.g. radix-sort), it may be quicker to use one of many $O(N \log N)$ methods such as heap-sort. For an image of N pixels, using C-connectivity, we perform N `MakeSets` and at most $NC/2$ `FindRoots`. This is because for all pixels we do a `FindRoot` on those of the C neighbors which have been processed before the current pixel. Disregarding image edge effects, one half of the neighbors of each pixel will have been processed beforehand on average. For these numbers of `FindRoots` and `MakeSets`, Tarjan [88] derives a worst case complexity of $\Theta(NC \log_{1+C/2} N)$, or $\Theta(N \log N)$ for fixed connectivity, when path compression is used, as is the case in our algorithm. In our measurements, CPU-time was a linear function of image size N, though an $N \log N$ relationship cannot be ruled out from the data.

Explicitly including the connectivity in the computational complexity is particularly necessary when contemplating extension to more than two dimensions. Extension to three dimensional images is straightforward, since all that has to be changed is the way pixel coordinates are stored in integers, and 6, 18 or 26 connectivity must be used. This does increase the computing time somewhat, but not prohibitively.

The computational complexity of the priority-queue algorithm is $O(N\lambda \log \lambda)$, which becomes $O(N^2 \log N)$ if $\lambda = N$. The latter is the case when computing pattern spectra (see section 4.7). An example of this $O(N^2 \log N)$ behaviour is seen when the number M of regional maxima in an image is maximal. In the case of 4-connectivity we have $M = N/2$ points in a checkerboard pattern. Suppose all these points have the maximum grey level, that the non-regional maximum pixels are assigned strictly descending grey levels in scan-line order, and that $\lambda = N$. As each of the $N/2$ maxima is processed, *all* the previously processed pixels are flooded, before the algorithm encounters the next regional maximum, and stops the flooding process. Therefore, on average, a region of $N/2$ pixels is flooded for each of the $N/2$ maxima. Each visit to a pixel bears a cost of $O(\log N)$ due to the use of a priority queue. Therefore CPU times are expected to increase proportional to $N^2 \log N$ in this case. This has been verified experimentally [103].

The computational cost of the Max-tree algorithm is dominated by the flood filling process which is inherently linear in both the number of pixels and in the connectivity. Pruning the Max-tree requires visiting all nodes in the tree at most twice, so it is also $O(N)$. Computing the output image is also linear in the number of pixels.

One somewhat peculiar feature in the computational complexity of the Max-tree algorithm is its dependency on the number of unoccupied grey levels between parent and child nodes. As stated, the flood filling procedure uses an array of booleans to store which levels below the current are occupied. Insertion into this array is done in constant time, but retrieval of the highest grey level below the current takes a time proportional to the number of unoccupied levels between Max-tree nodes in the same branch. Thus, a contrast stretch followed by a Max-tree filter is slower than the reverse order. We tested this in a particular bad case: a checkerboard pattern consisting of alternating pixels either of grey levels 0 and 1 or of 0 and 255. In this case, $N/2$

maxima must be processed. For each maximum there are four insertions into the array, followed by one retrieval. In the first case the retrieval needed to inspect only one element, in the second 255 elements. For a 256×256 image, the first case took 80 ms, the second 222 ms. The other two algorithms show no differences in CPU times, as expected. It is possible to replace the boolean array by other data structures in which retrieval is cheaper, such as a heap ($\log n$), but only at the expense of higher insertion cost. Since the number of insertions is at least C times greater than the number of retrievals, the original choice appears to be the best.

Apart from computational complexity, memory use must be taken into account, not just to estimate whether enough resources are available for a given image size, but also because memory bandwidth may actually limit the processing speed if large (volume) data sets are used. Apart from the original image, the priority-queue method requires a queue for detection of the regional maxima, which in our implementation requires N integers (with N the image size), a priority queue of either 2λ or N integers, depending on which value is the smaller, a label image of N integers, a fill list of λ (worst case N) integers to store the pixels belonging to the current maximum, and optionally an output image (the standard implementation overwrites the original image, which may not be desirable). The union-find method uses only the parent array (which becomes the output image) of N integers and the sorted pixel array (also N integers), apart from the input image. It therefore yields a memory saving of at least $2N$ integers with respect to the priority-queue method, and possibly an image of N pixels, as well, if a separate output image is used in the priority-queue approach. The Max-tree approach is the most costly, memory-wise. The hierarchical queue and status arrays each require N integers. The Max-tree itself may have N nodes (worst case), each of which contains its area, pointer to its parent, and the output grey level, or $2N$ integers and N pixels. An output image may also be required, if the input image may not be overwritten. The savings obtained by using the union-find approach compared to the Max-tree method is $2N$ integers and N pixels (or $2N$ pixels if a separate output image is used).

These differences in memory use are particularly important for 3-D analysis. In the case of the $128 \times 128 \times 62$ volume, the total memory use of the Max-tree method was 57.5 MB, priority-queue methods needed 45 MB, whereas the union-find algorithm required only 25 MB.

4.6 Extension to Attribute Openings

All three algorithms can be extended to compute other attribute openings and closings. The priority-queue approach has been extended by Breen and Jones [19], and a union-find algorithm based on the area opening version described here has also been published [103]. The original Max-tree algorithm can be used for not just attribute openings and closings, but also for thinnings and thickenings [83].

Unlike the area of a set, many attributes cannot be computed "on the fly". Therefore, the versions for attribute openings of all three algorithms rely on auxiliary data sets per peak component which has been processed partially. The auxiliary data are chosen in such a way that they can be updated pixel by pixel easily, and that the desired attribute can be computed efficiently. In the case of the union-find and Max-tree approaches, easy merging of auxiliary data sets of different connected sets of pixels must also be possible. For example, consider the opening using

the moment of inertia $I(C)$ of connected set C, which is defined in 2-D as

$$I(C) = \sum_{(x,y)\in C} (x-\bar{x})^2 + (y-\bar{y})^2 \tag{4.6}$$

with x and y the pixel coordinates, and $\bar{x}$ and $\bar{y}$ their mean values. In this case it can be shown that the area $A(C)$, $\sum x$, $\sum y$, $\sum x^2$, and $\sum y^2$ as auxiliary data are sufficient. As pixels are added to the set, the sums and area are updated in constant time. Merging two data sets reduces to adding the area and sums.

A number of adaptations to data structures and algorithms are needed in both the priority-queue and union-find approaches. First of all, in both cases the attributes are only computed when a peak component has completely been processed, i.e., when a grey level boundary is crossed. In the pixel-queue approach, this occurs when a pixel with lower grey level is retrieved from the priority queue. The attribute is computed, compared to λ and if it is smaller the flooding proceeds, otherwise the component is filled with the current grey level. Because the priority queue algorithm processes one peak component at a time, only one auxiliary data set is needed. Memory-wise this is the optimal solution.

Breen and Jones [19] use a single "driver" routine with function pointers as arguments to be able to compute any attribute opening, without code duplication. Their method requires functions to create, destroy, add a pixel to an auxiliary data set, and computing the attribute value. The union-find version uses a similar approach, but a merge function is also needed.

In the union-find case we need some more adaptations. First of all, it is no longer possible to store the attribute value as a negative number in the root node parent array itself. Instead, negative values (`ACTIVE` and `INACTIVE`) are used to flag active and inactive root nodes of sets. A second array `auxdata` is used to store pointers to auxiliary data sets. Only active root nodes have valid auxiliary data sets assigned to them. Because multiple peak components are processed simultaneously, $N/2$ auxiliary data sets are needed in the worst case. Computing attributes is done when the next pixel in the sorted pixel array has a lower grey level than the current. In that case all attributes of all active peak components at the current grey level are computed and compared to λ. All peak components which have attribute larger than lambda are set to inactive. For details see [103].

In the Max-tree case similar adaptations are needed with respect to the area opening algorithm presented here. In this case, though only a single peak component is flooded at a time, it may have collected data at multiple grey levels. In the worst case, G sets (with G the number of grey levels) are needed. Though still worse than the priority-queue method, this is considerably less than the worst case in the union-find approach. In this scheme, a Max-tree node stores a pointer to its parent, the attribute value, and grey level in the output image as before. It does not need a pointer to its auxiliary data, because we can store these in an array of length G, and use the current grey level of the node to reference the correct data set.

We have performed timings of the three algorithms for the moment-of-inertia opening, and the relative results are very similar to those obtained with the area opening. On 256×256 natural images, the union-find method took some 155 ms, compared to 33 ms for the area opening. The Max-tree approach took some 200 ms, and the priority-queue approach 540 ms at $\lambda = 32768$.

4.7 Extension to Pattern Spectra

All three algorithms can readily be adapted to computing area opening and closing pattern spectra efficiently. For the priority-queue method this has been done by Breen and Jones [19], and a similar extension has been done for the union-find case [56].

Let f and g be grey scale images. A size distribution or granulometry is a set of operators $\{\alpha_r\}$ with r from some totally ordered set Λ (usually $\Lambda \subset \mathbb{R}$ or $\mathbb{Z}$), with the following three properties [50, 101]:

$$\alpha_r(f) \leq f \tag{4.7}$$

$$f \leq g \Rightarrow \alpha_r(f) \leq \alpha_r(g) \tag{4.8}$$

$$\alpha_r(\alpha_s(f)) = \alpha_{\mathbf{max}\,(r,s)}(f), \tag{4.9}$$

for all $r, s \in \Lambda$. Since (4.9) implies idempotence, it can be seen that size distributions are openings. The pattern spectrum S_f^α obtained by applying a size distribution α_r to grey scale image f can be defined as the integral of the grey level of $\alpha_r(f)$ over the image domain. In the discrete case we have:

$$S_f^\alpha(r) = \sum_{x \in \mathbf{M}} (\alpha_r(f))(x). \tag{4.10}$$

A trivial algorithm for computation of area pattern spectra consists of performing area openings at $N_s \leq N$ scales, ranging from $\lambda = 1$ to $\lambda = N$, and computing the sum of grey levels of the resulting image at each scale, and storing the results in an array of N_s bins. This method yields a worst-case computational complexity of $O(N_s N^2 \log N)$ for the priority-queue algorithm and $O(N_s N \log N)$ for the union-find approach. In both cases it is possible to reduce the computational cost by a factor of N_s, by effectively performing an area opening with $\lambda = N$, which will merge all flat-zones of the image. During the merging process, the spectrum is updated as each peak component is merged with flat zones at a lower grey level (for details see [19, 56]). If a peak component of area A is merged with a set at a lower grey level, a function `Bin` is used to map the area to the range of bin numbers in the spectrum $(0, 1, \ldots, N_s)$. These adaptations can be seen in Figure 4.10 for the union-find case. It has been demonstrated that it is indeed possible to compute an area pattern spectrum in the time needed for a single area opening. A pattern spectrum version of the Max-tree algorithm is also trivial to implement. In this case the tree-traversal used for computing the grey levels of the output image is used to update the pattern spectrum as above.

The three algorithms have been implemented and yield CPU times indistinguishable from performing a single area opening with the same method with $\lambda = N$ in all cases tested (data not shown).

4.8 Conclusions

The Max-tree and union-find algorithms for computation of area openings and closings have clear advantages over the priority-queue algorithm in terms of computing time, especially when using

```
void Union ( int n, int p ) { int r=FindRoot(n);
  if ( r != p ) {
    if ( I[p]!=I[r] ) {
      spec[Bin(-parent[r])] -=
        (I[r]-I[p]) * parent[r];
    }
    parent[p] = parent[p] + parent[r];
    parent[r] = p;
  }
}
```

```
  /* array S contains sorted pixel list */
  for (i=0;i<Ns;i++)
    spec[i]=0;
  greysum=0;
  for (p=0; p<Length(S); p++) {
    pix = S[p];
    greysum += I[p];
    MakeSet(pix);
    for all neighbors nb of pix do
      if ((I[pix] < I[nb]) ||
          ((I[pix] == I[nb]) && (nb<pix)))
        Union(nb,pix);
  }
  for (i=0; i<Ns; i++) {
    spec[i] = greysum - spec[i];
    greysum = spec[i];
  }
```

Figure 4.10. *Pseudo-code showing how to compute an area opening pattern spectrum: (top) modified* `Union` *procedure, in which the function* `Bin` *maps the range of areas to the range of bins; (bottom) code of the area pattern spectrum algorithm itself. The* `FindRoot` *procedure is left unchanged (see Fig. 4.5).*

large values of λ. Furthermore, the strong dependence of computing time on image content seen in the priority-queue algorithm is lacking in both. Though theoretically less computationally efficient, the union-find approach outperforms the Max-tree method, especially in large (3-D) data sets. This may be caused by the much larger memory requirements of the Max-tree approach, and its less regular access of memory due to the flooding process.

In the case of the area opening, the union-find is also the most efficient in terms of memory usage, especially if the original image may not be overwritten. In that case it uses between 2.3 and 3 times less memory than the Max-tree approach, and about 40-50% less than the priority-

queue method (depending slightly on the size of a pixel). However, in the more general case of attribute operators, the union-find approach requires more space for auxiliary data sets in the worst case. If auxiliary data sets required to compute the attributes are large, the priority-queue method in particular may be more efficient in terms of memory usage.

The union-find algorithm is suitable for parallel implementation on shared-memory systems, since it processes extrema concurrently, while the other algorithms process extrema sequentially. Besides, the algorithm does not use a queue based flooding process, which is particularly difficult to parallelize. It is not hard to augment the union-find algorithm with locking-primitives to avoid concurrent accesses to shared memory locations, as in the case of connected component labeling [40].

Unlike the current implementations of the priority-queue and union-find algorithms, the Max-tree method can be used for attribute thinnings as well. The separation of Max-tree construction, Max-tree filtering and image restitution is elegant and very versatile. However, there is no particular reason to use a flooding approach to construct the Max-tree. Indeed, it is slightly impractical, because merging two sets of pixels belonging to different nodes requires reflooding one of the sets and setting all grey levels and STATUS values correctly. Therefore, in chapter 5 we present a union-find approach to building and filtering Max-trees, in which merging nodes can be done in constant time. This algorithm will also be designed for parallel execution on shared-memory systems.

Finally, the disjoint set approach discussed here could be generalized to other, and possibly all connected filters. The reason for this is that all connected filters operate on the flat zones of an image, merging them, or changing their intensities, but never splitting them. Tarjan's union-find method provides an efficient means to perform such operations.

This chapter is a major extension of: A. Meijster, M. A. Westenberg and M. H. F. Wilkinson, Interactive shape preserving filtering and visualization of volumetric data. In: *Proc. Fourth IASTED Conf. Comp. Signal Image Proc.* (SIP2002), pp. 640-643, Kauai, Hawaii, USA, August 12-14, 2002.

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

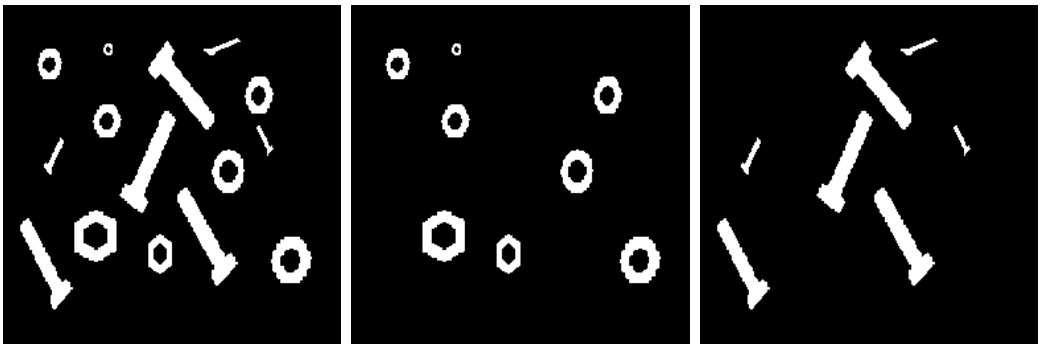


Figure 5.1. *Decomposition of a binary image of nuts and bolts of different sizes into two different shape classes: (left) original image; (middle) filter with criterion "number of holes* > 0*"; (right) difference between (left) and (middle).*

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 λ, 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

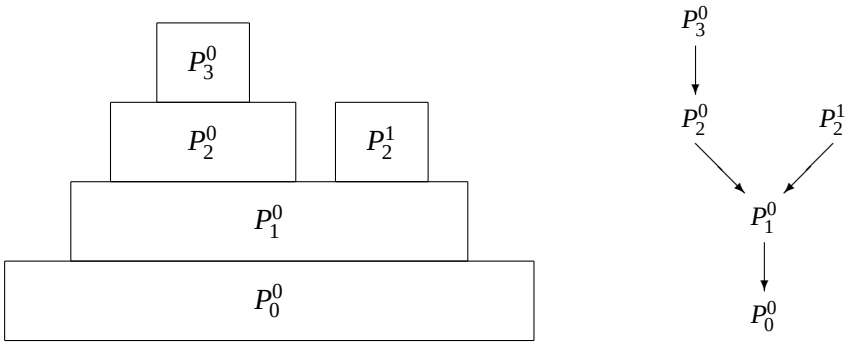


Figure 5.2. *Peak components of a discrete signal (left) and the corresponding Max-tree (right).*

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 $f : V \to \mathbb{R}$ 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.

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

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_h^k to denote the kth 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

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 $T(P_h^k)$ of a node P_h^k, from a set (usually $\mathbb{R}$ or $\mathbb{Z}$) with an order $\leq$. Given a threshold value λ 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_h^k is removed if $T(P_h^k) < \lambda$ or if one of its ancestors is removed.

Max A node P_h^k is removed if $T(P_h^k) < \lambda$ 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_h^k is removed if $T(P_h^k) < \lambda$; 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_h^k 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.

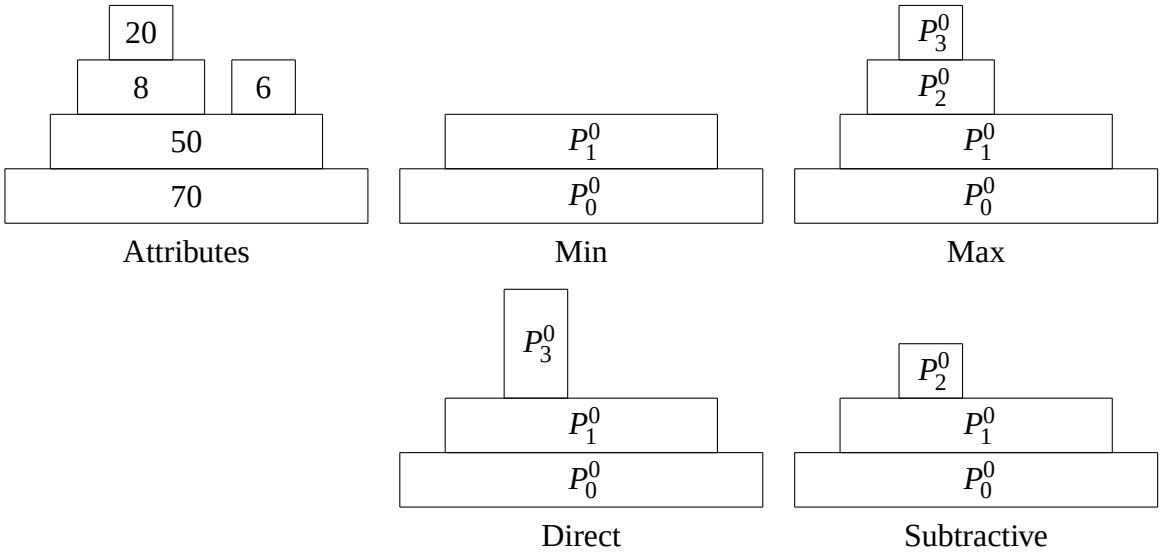


Figure 5.3. *Result after filtering the signal in Fig. 5.2 with four different decision rules, using* $\lambda = 10$ *as the attribute threshold.*

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 `par` which can be regarded as a (modifiable) function from nodes to nodes. We call a tree which is encoded in this `par` array, a `par`-tree.

We define $\texttt{par}^n[x]$ by repeated application of `par`, i.e. $\texttt{par}^0[x] = x$ and $\texttt{par}^{n+1}[x] = \texttt{par}^n[\texttt{par}[x]]$. A node $x \in V$ is called a *root* of a `par`-tree if $\texttt{par}[x] = x$. For each non-root, we can find its ancestors by repeated application of function `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 `par` are monotonically descending. In other words, we construct `par` such that for all $x \in V$, we have $f(\texttt{par}[x]) \leq f(x)$.

During the construction of the `par`-tree, we make sure that we do not introduce any cycles in the `par`-tree. For any vertex x, we define the "oldest ancestor" up to h by means of the recursive function *anc* given by

$$anc(x,h) \quad = \quad \textbf{if } \texttt{par}[x] = x \vee f(\texttt{par}[x]) < h \textbf{ then } x \textbf{ else } anc(\texttt{par}[x], h)\ .$$

Since `par` is acyclic, this recursion always terminates.

In Fig. 5.4, an example is shown of a small artificial image and a possible `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

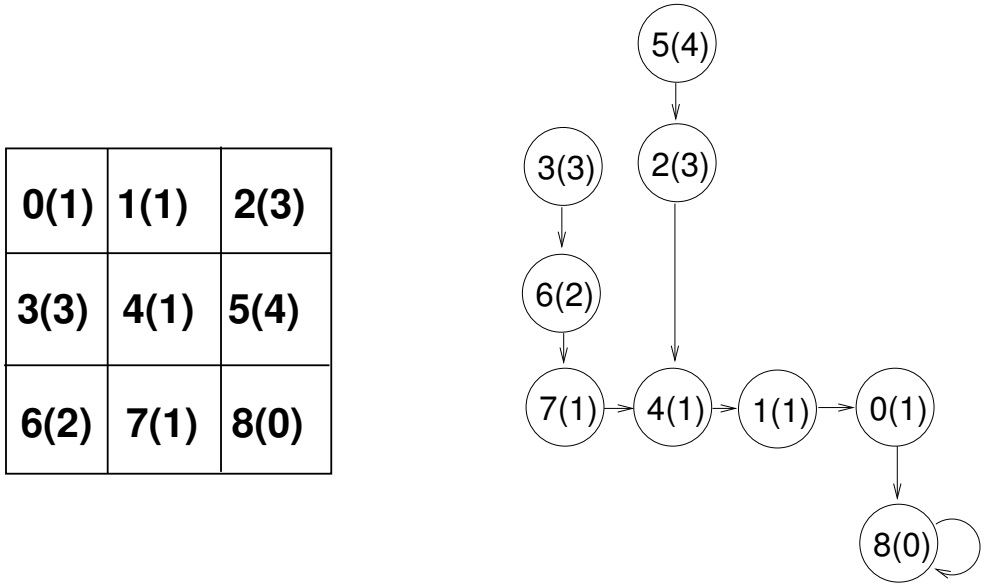


Figure 5.4. *An image (left) and a possible* `par`*-tree structure (right). Nodes are numbered* $p(q)$*, where* p *is the node number, and* $q = f(p)$.

maximum, and therefore becomes a leaf node of the `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, $\texttt{par}[5]$ is set to 2. Furthermore, since pixel w is reachable from 5, via a descending path, we decide to set $\texttt{par}[2]$ 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 `par`-path from 4 to 8. Pixel 8 has the lowest function value of all pixels, and therefore becomes the root of the entire `par`-tree. Verifying the remaining parent-pointers is left as an exercise for the reader. Note, that the property $f(\texttt{par}[x]) \leq f(x)$ allows more than one possible tree: for example we could set $\texttt{par}[6] = 4$ instead of $\texttt{par}[6] = 7$.

As we can see in Fig. 5.4, following a `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 `par`-tree. We define a vertex x to be a *level root* if x is the root of the tree (i.e. $\texttt{par}[x] = x$) or $f(\texttt{par}[x]) < f(x)$. It is easy to see that $h \leq f(x)$ implies that $x' = anc(x,h)$ satisfies $f(\texttt{par}[x']) < h \leq f(x')$ 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 `par`-tree. We start with a forest of singleton-trees, by setting $\texttt{par}[x] = x$, 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 `par`-structure gets updated by merging the trees to which p and q belong. This process stops when

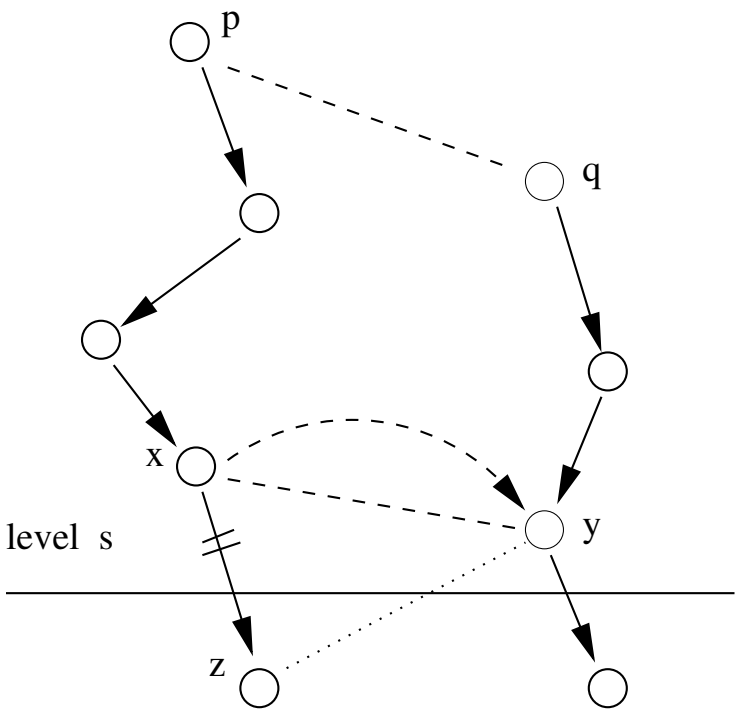


Figure 5.5. *Pair replacement.*

$E = \emptyset$.

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 $f(\texttt{par}[r]) < s \leq f(r) \leq f(q)$. 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 `par`-path from p to r. Using this argument, we replace the pair (p,q) by the pair (x,y), where $x = anc(p,s)$ and $y = anc(q,s)$ (see Fig. 5.5).

We now would like to set $\texttt{par}[x]$ to y. However, if we do this directly, we disconnect the ancestors of x from x and its descendant. Therefore, we store $\texttt{par}[x]$ 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 $\texttt{par}[x] = x$. 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 := anc(x,s) ; y := anc(y,s) ;
    if f(x) < f(y) then swap(x,y) end; (* s = f(y) ≤ f(x) *)
    while x ≠ y ∧ par[x] ≠ x do
        z := par[x] ; par[x] := y ;
        x := y ; y := anc(z, f(z)) ;
    end ;
```

```
    if x ≠ y then par[x] := y ;
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

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

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

```
procedure levroot(x) returns V =
    if par[x] = x ∨ f(x) ≠ f(par[x]) then return x end ;
    par[x] := levroot(par[x]) ;
    return par[x] ;
end .
```

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

```
procedure anc(x,s) returns V =
    while par[x] ≠ x ∧ s ≤ f(par[x]) do x := levroot(par[x]) end ;
    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 `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 `par`-path starting at p.

5.4 Concurrent Merging of the trees

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

We use *nthreads* processes that are numbered consecutively from 1 to *nthreads*. Each processor has a private integer variable *self* ($1 \leq self \leq nthreads$), 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 $lwb(p) = (p-1) * N/nthreads$, and $upb(p) = p * N/nthreads$. 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.

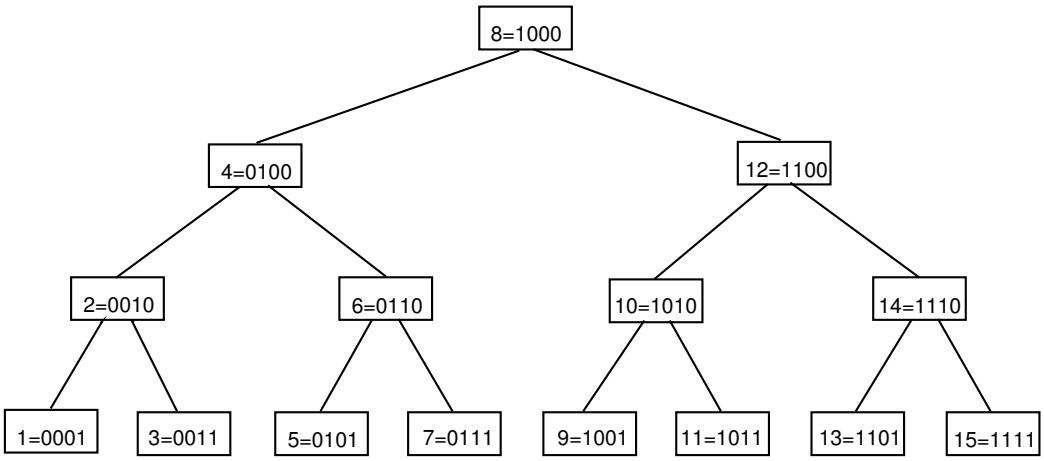


Figure 5.6. *Binary tree used for merging domains.*

We decide that the process with number p ($p < nthreads$) 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 $nthreads = 15$. 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 *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 $P(sem[self])$. 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 $self + 1 \leq nthreads$ suffices to determine whether a right child exists. When process p gets woken up by its child(ren), it calls the routine $glue(self)$, which is a routine that calls $merge(x,y)$ for all $x \in V_{self}$ and $y \in V_{self+1}$. 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 *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:

```
procedure globalmerge (self) =
    if nthreads = 1 then return ;
    if self mod 2 = 0 then (* internal nodes *)
        V(sem[self − 1]); P(sem[self]);
        if self+1 ≤ nthreads then
            P(sem[self]); glue(self);
        end ;
        if self ≠ root then
            V(sem[ancestor(self)]);
            P(sem[self]); (* 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[self]); glue(self);
        end ;
        V(sem[ancestor(self)]);
        P(sem[self]); (* 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

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

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

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

where $\bigoplus$ denotes summing of attributes. Obviously, in this simple case $accat[v]$ 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 $accat[v] = 0$, for all $v \in V$. Accumulating attributes simply boils down to traversing `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 $accat[v]$. If v is not the root of the Max-tree, we traverse down the `par`-path of v until we reached the root. During this traversing we add the unit-attribute to $accat[p]$ for all p visited along the `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 $levroot(v)$ 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 `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 *filt* in which we store the filtered data set. The only thing a process p has to do, is to follow `par`-pointers for each point v of its private domain $[lwb(p), upb(p))$, until it reaches an ancestor w of v which satisfies the filter criterion. When w is found, $filt[v]$ is set to $f(w)$.

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

```
procedure directfilter (self f, lambda) =
    for v := lwb(self f) to upb(self) do
        if filt[v] = −1 then
```

```
            w := v;
            while par[w] ≠ w ∧ filt[w] = −1 ∧
                (f(w) = f(par[w]) ∨ accat[w] < lambda) do
                w := par[w];
            end;
            if filt[w] ≠ −1 then
                val := filt[w]; (* criterion satisfied at level filt[w] *)
            else if accat[w] ≥ 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 ≠ w do
                if u ∈ [lwb(self), upb(self)] then
                    filt[u] := val;
                end;
                u := par[u];
            end;
            if w ∈ [lwb(self), upb(self)] then
                filt[w] := val;
            end;
        end;
    end;
end.
```

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 λ without recomputing a new Max-tree. This allow 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

Table 5.1. *Timings and speedups for the build and filtering stages.*

threads	build	speedup	filter	speedup	threads	build	speedup	filter	speedup
1	19.9	1.0	3.0	1.0	10	2.4	8.3	0.60	5.0
2	10.3	1.9	1.7	1.8	12	2.1	9.5	0.53	5.7
4	5.3	3.8	1.0	3.0	14	1.8	11.1	0.47	6.4
6	3.5	5.7	0.75	4.0	16	1.7	11.7	0.40	7.5
8	3.0	6.6	0.68	4.4	32	1.6	12.4	0.35	8.6

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 λ 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.

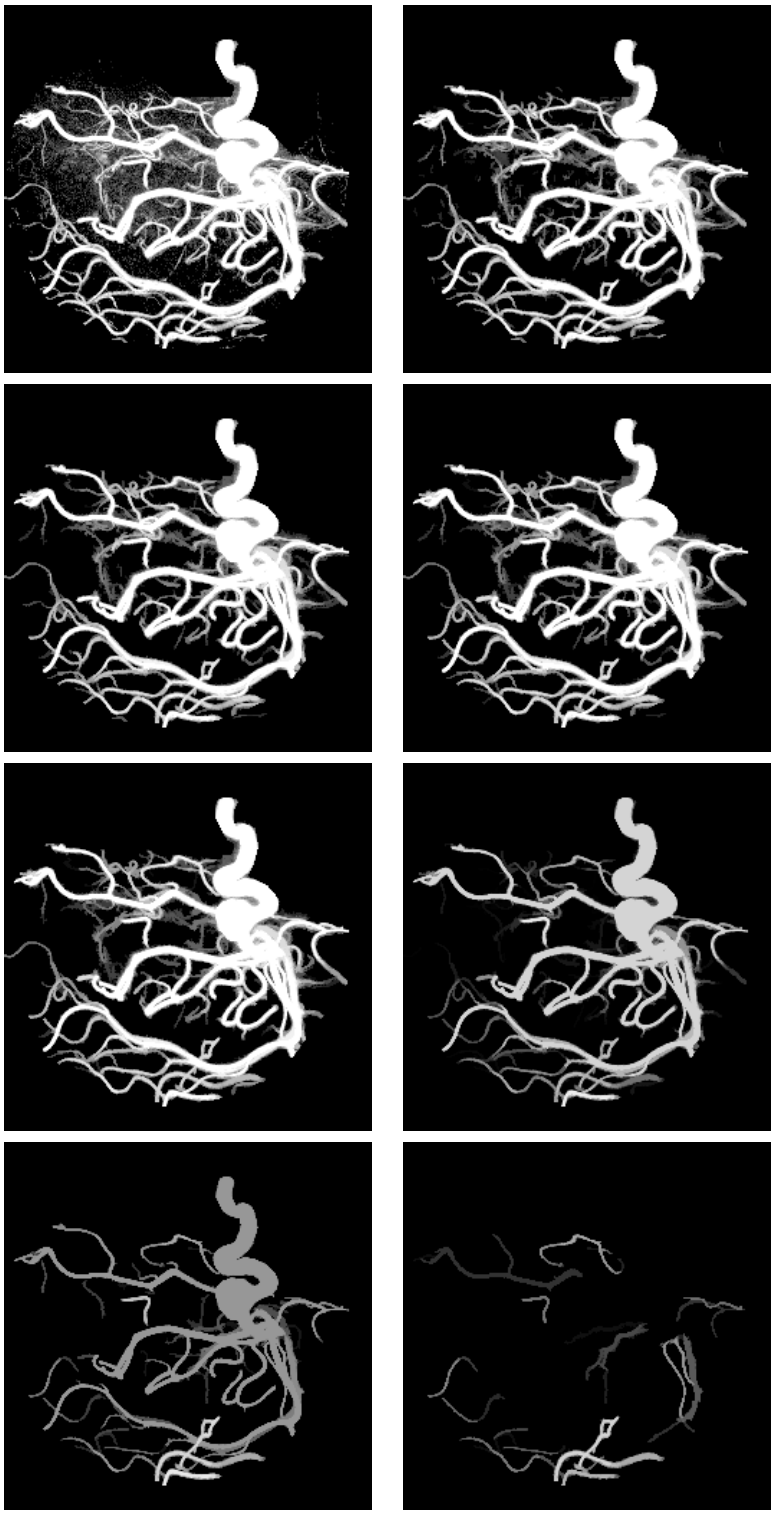


Figure 5.7. *Magnetic resonance angiography volume data set (size 256^3) filtered interactively with an attribute thinning as shape filter. The attribute used was $I/V^{5/3}$, with I the moment of inertia, and V the volume of a peak component; the top left-hand image is the original, in the others the 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.*

This chapter is a slightly modified version of: J. B. T. M. Roerdink and A. Meijster, The watershed transform: definitions, algorithms, and parallellization strategies, *Fundamenta Informaticae*, **41**, pp. 187-228, 2000.

Chapter 6

The Watershed Transform: Definitions, Algorithms and Parallelization Strategies

Abstract

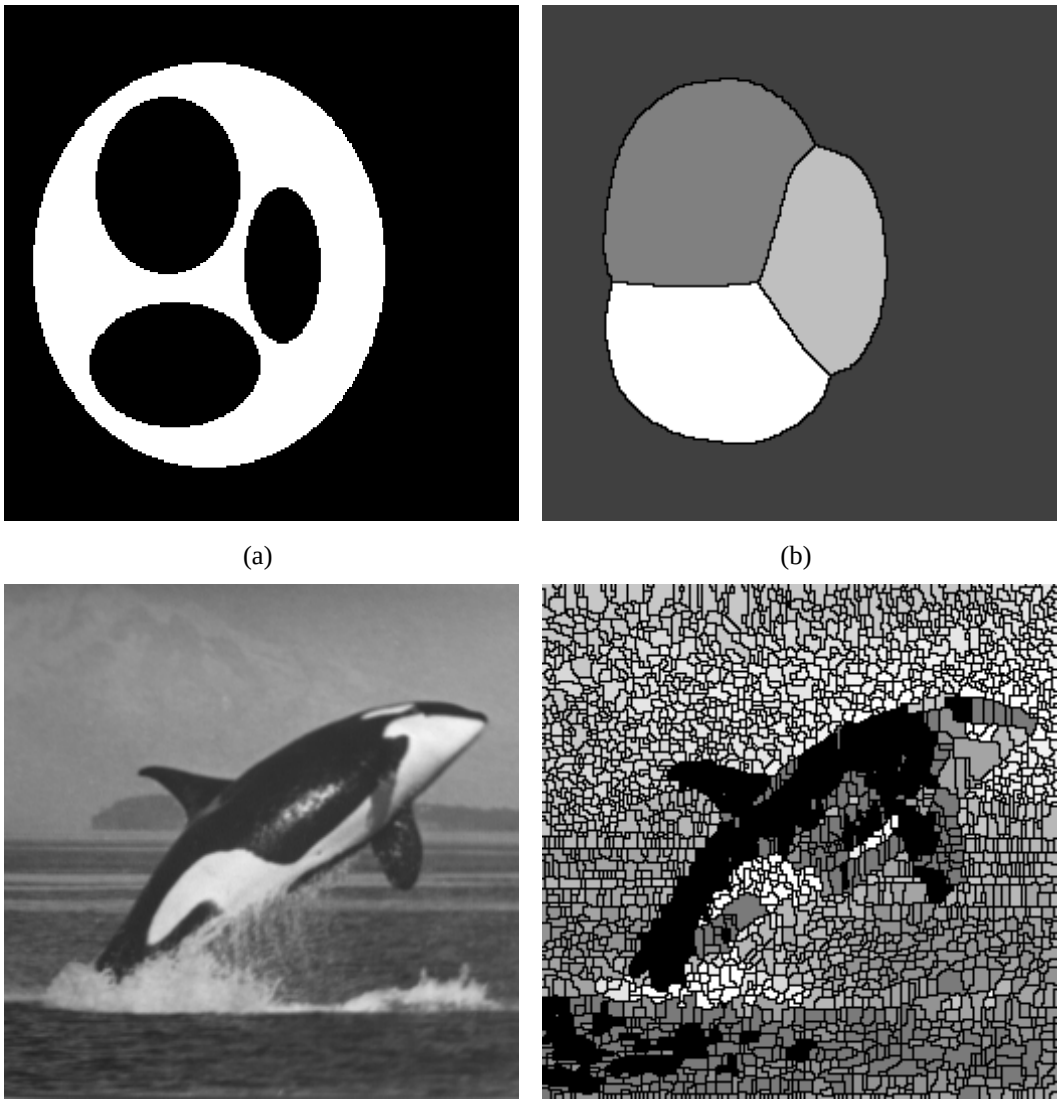
The watershed transform is the method of choice for image segmentation in the field of mathematical morphology. We present a critical review of several definitions of the watershed transform and the associated sequential algorithms, and discuss various issues which often cause confusion in the literature. The need to distinguish between definition, algorithm specification and algorithm implementation is pointed out. Various examples are given which illustrate differences between watershed transforms based on different definitions and/or implementations. The second part of the paper surveys approaches for parallel implementation of sequential watershed algorithms.

6.1 Introduction

In grey scale mathematical morphology the watershed transform, originally proposed by Digabel and Lantuéjoul [26, 46] and later improved by Beucher and Lantuéjoul [13], is the method of choice for image segmentation [14,86,102]. Generally spoken, *image segmentation* is the process of isolating objects in the image from the background, i.e., partitioning the image into disjoint regions, such that each region is homogeneous with respect to some property, such as grey value or texture [37].

The watershed transform can be classified as a region-based segmentation approach. The intuitive idea underlying this method comes from geography: it is that of a landscape or topographic relief which is flooded by water, watersheds being the divide lines of the domains of attraction of rain falling over the region [86]. An alternative approach is to imagine the landscape being immersed in a lake, with holes pierced in local minima. Basins (also called 'catchment basins') will fill up with water starting at these local minima, and, at points where water coming from different basins would meet, dams are built. When the water level has reached the highest peak in the landscape, the process is stopped. As a result, the landscape is partitioned into regions or basins separated by dams, called *watershed lines* or simply *watersheds*.

When simulating this process for image segmentation, two approaches may be used: either one first finds basins, then watersheds by taking a set complement; or one computes a complete



(a) (b)

(c) (d)

Figure 6.1. *Examples of watershed segmentation by immersion (see Definition 4). (a): synthetic image; (b): watershed transform of (a); (c): natural image; (d): watershed transform of (c). Different basins are indicated by distinct grey values.*

partition of the image into basins, and subsequently finds the watersheds by boundary detection. To be more explicit, we will use the expression 'watershed transform' to denote a labelling of the image, such that all points of a given catchment basin have the same unique label, and a special label, distinct from all the labels of the catchment basins, is assigned to all points of the watersheds. An example of a simple image with its watershed transform is given in Fig. 6.1(a-b). We note in passing that in practice one often does not apply the watershed transform to the original image, but to its (morphological) gradient [61]. This produces watersheds at the points

of grey value discontinuity, as is commonly desired in image segmentation.

One of the difficulties with this intuitive concept is that it leaves room for various formalizations. Different watershed definitions for continuous functions have been given, which will be briefly reviewed in Section 6.3.1. However, our main interest here is in *digital* images, for which there is even more freedom to define watersheds, since in the discrete case there is no unique definition of the path a drop of water would follow. Many sequential algorithms have been developed to compute watershed transforms, see e.g. [61,97] for a survey. They can be divided into two classes, one based on the specification of a recursive algorithm by Vincent & Soille [102], and another based on distance functions by Meyer [60]. In the context of parallel implementations there exists a notable tendency for introducing other definitions of the watershed transform, enabling easier parallelization. Examples are presented in Section 6.5.

The impression which the current literature on watershed algorithms makes upon the uninitiated reader can only be one of great confusion. Often it is uncertain exactly which definition for the watershed transform is used. Sometimes the definition takes the form of the specification of an algorithm. A careful distinction between algorithm specification and implementation is in many cases lacking. Without such a separation, correctness assessment of proposed algorithms is impossible. Even when a specification is given, the implementation often does not adhere to it. Ad hoc modifications are made to eliminate 'undesirable' consequences of a watershed definition, but such changes tend to create new problems by solving an old one. Or 'optimizations' are introduced, for greater speed or memory reduction, which in the process change the outcome of the algorithm as well, although this may often go undetected in the case of natural images. These questions are not purely academic, since the algorithm is widely used in e.g. medical image processing where undocumented side effects should be avoided.

The purpose of this paper is twofold. In the first part we present a critical review of several definitions of the watershed transform and the associated sequential algorithms, emphasizing the distinction between definition, algorithm specification and algorithm implementation. The second part of the paper surveys the main current approaches towards parallel implementation of watershed algorithms. An essential difficulty lies in the fact that the watershed transform is not a local concept. The decision whether a pixel belongs to a basin cannot be based on purely local considerations. Another problem with some algorithms is that the result depends on the order in which pixels are treated during execution. In the sequential case, this can be resolved by fixing the scanning order (e.g. raster scan), so that a deterministic result is obtained. In a parallel implementation this is no longer true since the outcome depends on the relative time instants at which different processors treat the pixels, and this is unpredictable in the case of asynchronous processors. The emphasis in the second part is on methodology and trends in current research. We point out the difficulties in the design of parallel watershed algorithms. Efficiency results are quoted to some extent in order to give the reader an idea of what is currently achievable. However, an in-depth comparison of the large body of results which have been obtained for different watershed algorithms on different architectures with different programming methodologies is beyond the scope of this paper.

There are a number of issues concerning the watershed transform which are not discussed explicitly. We mention a few of them. First, there is the question of *accuracy* of watershed lines. Usually, one has in mind here that the result should be a close approximation of the con-

tinuous case. That is, the digital distances playing a role in the watershed calculation should approximate the Euclidean distance. *Chamfer distances* are an efficient way to achieve accurate watershed lines [60]. Second, the watershed method in its original form produces a severe *oversegmentation* of the image, i.e., many small basins are produced due to many local minima in the input image, see Fig. 6.1(c-d). Several approaches exist to remedy this, such as markers or hierarchical watersheds [12, 61]; also parallellization of marker-based watershed algorithms has been studied [62, 66]. Third, we do not consider dedicated hardware architectures for fast computation of watershed transforms and related operations, see e.g. [44, 73]. Such architectures tend to solve a very restricted class of image processing tasks, whereas our interest here is in medium level image processing on general purpose (parallel) architectures.

The organization of this paper is as follows. In Section 6.2 some preliminaries are given. Section 6.3 presents definitions of the watershed transform, both for the continuous and the discrete case. Sequential watershed algorithms are reviewed in Section 6.4. Section 6.5 contains a survey of parallelization strategies for the watershed transform. Conclusions are drawn in Section 6.6.

6.2 Preliminaries

This section contains some background material on graphs (see e.g. [23]) and digital images.

6.2.1 Graphs

A graph $G = (V, E)$ consists of a set V of *vertices* (or *nodes*) and a set $E \subseteq V \times V$ of pairs of vertices. In a (un)directed graph the set E consists of (un)ordered pairs (v, w). Instead of 'directed graph' we will also write *digraph*. An unordered pair (v, w) is called an *edge*, an ordered pair (v, w) an *arc*. If $e = (v, w)$ is an edge (arc), e is said to be *incident* with its vertices v and w; conversely, v and w are called incident with e. We also call v and w *neighbours*. The set of vertices which are neighbours of v is denoted by $N_G(v)$. A *path* π of length ℓ in a graph $G = (V, E)$ from vertex p to vertex q is a sequence of vertices $(p_0, p_1, \ldots, p_{\ell-1}, p_\ell)$ such that $p_0 = p$, $p_\ell = q$ and $(p_i, p_{i+1}) \in E$, $\forall i \in [0, \ell)$. The length of a path π is denoted by $\mathsf{length}(\pi)$. A path is called *simple* if all its vertices are distinct. If there exists a path from a vertex p to a vertex q, then we say that q is *reachable* from p, denoted as $p \rightsquigarrow q$.

An undirected graph is *connected* if every vertex is reachable from every other vertex. A graph $G' = (V', E')$ is called a *subgraph* of $G = (V, E)$ if $V' \subseteq V$, $E' \subseteq E$. A *connected component* of a graph is a maximal connected subgraph of G. The connected components partition the vertices of G.

In a digraph, a path $(p_0, p_1, \ldots, p_{\ell-1}, p_\ell)$ forms a *cycle* if $p_0 = p_\ell$ and the path contains at least one edge. If all vertices of the cycle are distinct, we speak of a *simple cycle*. A *self-loop* is a cycle of length 1. In an undirected graph, a path $(p_0, p_1, \ldots, p_{\ell-1}, p_\ell)$ forms a cycle if $p_0 = p_\ell$ and $p_1, \ldots, p_\ell$ are distinct. A graph with no cycles is *acyclic*. A *forest* is an undirected acyclic graph, a *tree* is a connected undirected acyclic graph. A *directed acyclic graph* is abbreviated as DAG.

A *weighted graph* is a triple $G = (V, E, w)$ where $w : E \to \mathbb{R}$ is a weight function defined on the edges. A *valued graph* is a triple $G = (V, E, f)$ where $f : V \to \mathbb{R}$ is a weight function defined on the vertices. A *level component at level h* of a valued graph is a connected component of the set of nodes v with the same value $f(v) = h$. The *boundary* of a level component P at level h consists of all $p \in P$ which have neighbours with value different from h; the *lower boundary* of P is the set of all $p \in P$ which have neighbours with value smaller than h; the *interior* of P consists of all points of P which are not on the boundary. A *descending path* is a path along which the value does not increase. By $\Pi_f^{\downarrow}(p)$ we denote the set of all descending paths starting in a node p and ending in some node q with $f(q) < f(p)$. A *regional minimum* (*minimum*, for short) at level h is a level component P of which no points have neighbours with value lower than h, i.e. $\Pi_f^{\downarrow}(p) = \emptyset$ for all $p \in P$. A valued graph is called *lower complete* when each node which is not in a minimum has a neighbouring node of lower value.

6.2.2 Digital grids

A *digital grid* is a special kind of graph. Usually one works with the square grid $D \subseteq \mathbb{Z}^2$, where the vertices are called *pixels*. When D is finite, the *size* of D is the number of points in D. The set of pixels D can be endowed with a graph structure $G = (V, E)$ by taking for V the domain D, and for E a certain subset of $D \times D$ defining the connectivity. Usual choices are *4-connectivity*, i.e., each point has edges to its horizontal and vertical neighbours, or *8-connectivity* where a point is connected to its horizontal, vertical and diagonal neighbours. Connected components of a set of pixels are defined by applying the definition for graphs.

Distances between neighbouring nodes in a digital grid are introduced by associating a positive weight $d(p,q)$ to each edge (p,q). In this way a weighted graph is obtained. The distance $d(p,q)$ between non-neighbouring pixels p and q is defined as the minimum path length among all paths from p to q (this depends on the graph structure of the grid, i.e., the connectivity).

6.2.3 Digital images

A *digital grey scale image* is a triple $G = (D, E, f)$, where (D, E) is a graph (usually a digital grid) and $f : D \longrightarrow \mathbb{N}$ is a function assigning an integer value to each $p \in D$. A *binary image* f takes only two values, say 1 ('foreground') and 0 ('background'). For $p \in D$, $f(p)$ is called the *grey value* or *altitude* (considering f as a topographic relief). For the range of a grey scale image one often takes the set of integers from 0 to 255, but we do not make this assumption in this paper. A *plateau* or *flat zone* of grey value h is a level component of the image, considered as a valued graph, i.e., a connected component of pixels of constant grey value h. The *threshold set* of f at level h is

$$T_h = \{p \in D \mid f(p) \leq h\}. \tag{6.1}$$

6.2.4 Geodesic distance

Let $A \subseteq \mathcal{E}$, with $\mathcal{E} = \mathbb{R}^d$ or $\mathcal{E} = \mathbb{Z}^d$, and a, b two points in A. The *geodesic distance* $d_A(a,b)$ *between a and b within A* is the minimum path length among all paths within A from a to b (in the continuous case, read 'infimum' instead of 'minimum'). If B is a subset of A, define $d_A(a,B) = \text{MIN}_{b\in B}(d_A(a,b))$. Let $B \subseteq A$ be partitioned in k connected components $B_i, i = 1,\ldots,k$. The *geodesic influence zone* of the set B_i within A is defined as

$$iz_A(B_i,B) = \{p \in A \mid d_A(p,B_i) < d_A(p,B\backslash B_i)\}$$

The set $IZ_A(B)$ is the union of the geodesic influence zones of the connected components of B, i.e.,

$$IZ_A(B) = \bigcup_{i=1}^{k} iz_A(B_i,B)$$

The complement of the set $IZ_A(B)$ within A is called the SKIZ (*skeleton by influence zones*):

$$\text{SKIZ}_A(B) = A\backslash IZ_A(B)$$

So the SKIZ consists of all points which are equidistant (in the sense of the geodesic distance) to at least two nearest connected components (for digital grids, there may be no such points). For a binary image f with domain A, the SKIZ can be defined by identifying B with the set of foreground pixels.

6.3 Definitions of the watershed transform

In this section we introduce definitions of the watershed transform, which may be viewed as a generalization of the skeleton by influence zones (SKIZ) to grey value images. We start with the continuous case, followed by two definitions for the digital case, the algorithmic definition by Vincent & Soille [102], and the definition by topographical distance by Meyer [60]. A discussion of algorithms is postponed until Section 6.4.

6.3.1 Watershed definition: continuous case

A watershed definition for the continuous case can be based on distance functions. Depending on the distance function used one may arrive at different definitions. We restrict ourselves here to the one given in [60, 72], but other choices have been proposed as well [76].

Assume that the image f is an element of the space $\mathcal{C}(D)$ of real twice continuously differentiable functions on a connected domain D with only isolated critical points (the class of Morse functions on D forms an example [35, 71]). Then the *topographical distance* between points p and q in D is defined by

$$T_f(p,q) = \inf_{\gamma} \int_{\gamma} \|\nabla f(\gamma(s))\| \, \mathrm{d}s,$$

where the infimum is over all paths (smooth curves) γ inside D with $\gamma(0) = p$, $\gamma(1) = q$. The topographical distance between a point $p \in D$ and a set $A \subseteq D$ is defined as $T_f(p,A) = \inf_{a \in A} T_f(p,a)$. The path with shortest T_f-distance between p and q is a path of steepest slope. This motivates the following rigorous definition of the watershed transform.

Definition 3 (Watershed transform) *Let $f \in \mathcal{C}(D)$ have minima $\{m_k\}_{k \in I}$, for some index set I. The* catchment basin *$CB(m_i)$ of a minimum m_i is defined as the set of points $x \in D$ which are topographically closer to m_i than to any other regional minimum m_j:*

$$CB(m_i) = \{x \in D \mid \forall j \in I \backslash \{i\} : f(m_i) + T_f(x, m_i) < f(m_j) + T_f(x, m_j)\}$$

The watershed *of f is the set of points which do not belong to any catchment basin:*

$$Wshed(f) = D \cap \left(\bigcup_{i \in I} CB(m_i) \right)^c . \tag{6.2}$$

Let W be some label, $\mathsf{W} \notin I$. The watershed transform *of f is a mapping $\lambda : D \to I \cup \{\mathsf{W}\}$, such that $\lambda(p) = i$ if $p \in CB(m_i)$, and $\lambda(p) = \mathsf{W}$ if $p \in Wshed(f)$.*

So the watershed transform of f assigns labels to the points of D, such that (i) different catchment basins are uniquely labelled, and (ii) a special label W is assigned to all points of the watershed of f.

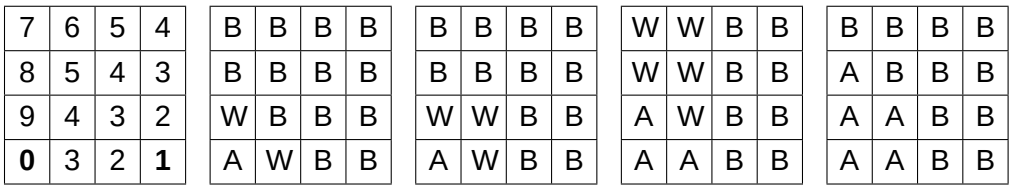
6.3.2 Watershed definitions: discrete case

A problem which arises for digital images is the occurrence of plateaus, i.e., regions of constant grey value, which may extend over large image areas. Such plateaus form a difficulty when trying to extend the continuous watershed definition based on topographical distances to discrete images. This nonlocal effect is also a major obstacle for parallel implementation of watershed algorithms, see Section 6.5.

The next algorithmic definition automatically takes care of plateaus, because it computes a watershed transform level by level, where each level constitutes a binary image for which a SKIZ is computed.

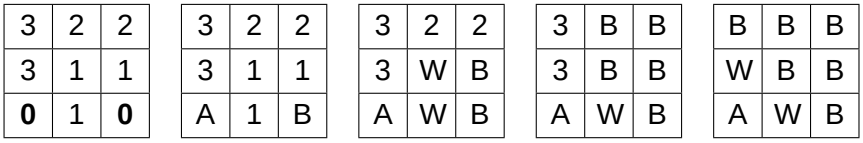
6.3.2.1 Algorithmic definition by immersion

An algorithmic definition of the watershed transform by simulated immersion was given by Vincent and Soille [97, 102] (see also [86, Ch. XI, H.5] for the binary case). Let $f : D \to \mathbb{N}$ be a digital grey value image, with h_{min} and h_{max} the minimum and maximum value of f. Define a recursion with the grey level h increasing from h_{min} to h_{max}, in which the basins associated with the minima of f are successively expanded. Let X_h denote the union of the set of basins computed at level h. A connected component of the threshold set T_{h+1} at level $h+1$ (cf. (6.1)) can be either a new minimum, or an extension of a basin in X_h: in the latter case one computes the geodesic influence zone of X_h within T_{h+1} (cf. Section 6.2.4), resulting in an update X_{h+1}. Let MIN_h denote the union of all regional minima at altitude h.



(a) original (b) 4-conn. (c) 8-conn. (d) 4-conn. (e) 8-conn.

Figure 6.7. *Watershed transform on the square grid, for different connectivity. (a): original image (minima indicated in bold); (b-c): results according to immersion (Definition 4); (d)-(e): results according to topographical distance (Definition 3, with T_f as defined in (6.6)).*



(a) (b) $h=0$ (c) $h=1$ (d) $h=2$ (e) $h=3$

Figure 6.13. *Watershed transform by immersion on the 4-connected grid, showing relabelling of 'watershed' pixels. (a): Original image; (b-e): labelling steps based on (6.3).*

Definition 4 (Watershed by immersion) *Define the following recursion:*

$$\begin{cases} X_{h_{min}} &= \{p \in D \mid f(p) = h_{min}\} = T_{h_{min}} \\ X_{h+1} &= \text{MIN}_{h+1} \cup IZ_{T_{h+1}}(X_h), \qquad h \in [h_{min}, h_{max}) \end{cases} \tag{6.3}$$

The watershed $Wshed(f)$ *of* f *is the complement of* $X_{h_{max}}$ *in* D*:*

$$Wshed(f) = D \setminus X_{h_{max}}$$

For an example of the watershed transform according to the above recurrence, see Fig. 6.7(a-c), in which A and B are labels of basins, and W is used to denote watershed pixels (in this and other figures to follow, minima pixels in the input image are indicated in bold). Note the dependence on the connectivity.

According to the recursion (6.3), it is the case that at level $h+1$ all non-basin pixels (i.e. all pixels in T_{h+1} except those in X_h) are potential candidates to get assigned to a catchment basin in step $h+1$. Therefore, the definition allows that pixels with grey value $h' \leq h$ which are not yet part of a basin after processing level h, are merged with some basin at the *higher* level $h+1$. Pixels which in a given iteration are equidistant to at least two nearest basins may be provisionally labelled as 'watershed pixels' by assigning them the label W (we will refer to such pixels as W-pixels). However, in the next iteration this label may change again. A definitive labelling as watershed pixel can only happen after all levels have been processed. An example [79] is given in Fig. 6.13, for a 3×3 discrete image on the square grid with 4-connectivity. There are two local minima (the zeroes), so there will be two basins whose pixels are labelled A, B. The labelling

according to (6.3) is shown in Fig. 6.13(b)-(e). This shows the phenomenon of relabelling of W-pixels: the pixel in the second row, second column, is first labelled W, then B.

The algorithm presented by Vincent & Soille in [102] as an implementation of (6.3) in fact does not adhere to this definition, see Section 6.4.1 below.

6.3.2.2 Watershed definition by topographical distance

We follow here the presentation in [60]. Let f be a digital grey value image. Initially, we assume that f is *lower complete*, that is, each pixel which is not in a minimum has a neighbour of lower grey value [61]. This assumption will be relaxed later.

The *lower slope* $LS(p)$ of f at a pixel p, is defined as the maximal slope linking p to any of its neighbours of lower altitude. Formally,

$$LS(p) = \underset{q \in N_G(p) \cup \{p\}}{\text{MAX}} \left(\frac{f(p) - f(q)}{d(p,q)} \right), \tag{6.4}$$

where $N_G(p)$ is the set of neighbours of pixel p on the grid $G = (V,E)$, and $d(p,q)$ is the distance associated to edge (p,q) (for $q = p$ the expression following the MAX-operator in (6.4) is defined to be zero). Note that always $LS(p) \geq 0$, and that $LS(p) = 0$ if and only if f has a local minimum at p. The *cost* for walking from pixel p to a neighbouring pixel q is defined as

$$cost(p,q) = \begin{cases} LS(p) \cdot d(p,q) & \text{if } f(p) > f(q) \\ LS(q) \cdot d(p,q) & \text{if } f(p) < f(q) \\ \frac{1}{2}(LS(p) + LS(q)) \cdot d(p,q) & \text{if } f(p) = f(q) \end{cases} \tag{6.5}$$

Definition 5 *The set of lower neighbours q of p for which the slope $(f(p) - f(q))/d(p,q)$ is* maximal, *i.e. equals the value $LS(p)$, is denoted by $\Gamma(p)$. The set of pixels q for which $p \in \Gamma(q)$ is denoted by $\Gamma^{-1}(p)$.*

The *topographical distance along a path* $\pi = (p_0, \ldots, p_\ell)$ between $p_0 = p$ and $p_\ell = q$ is defined as

$$T_f^\pi(p,q) = \sum_{i=0}^{\ell-1} cost(p_i, p_{i+1}).$$

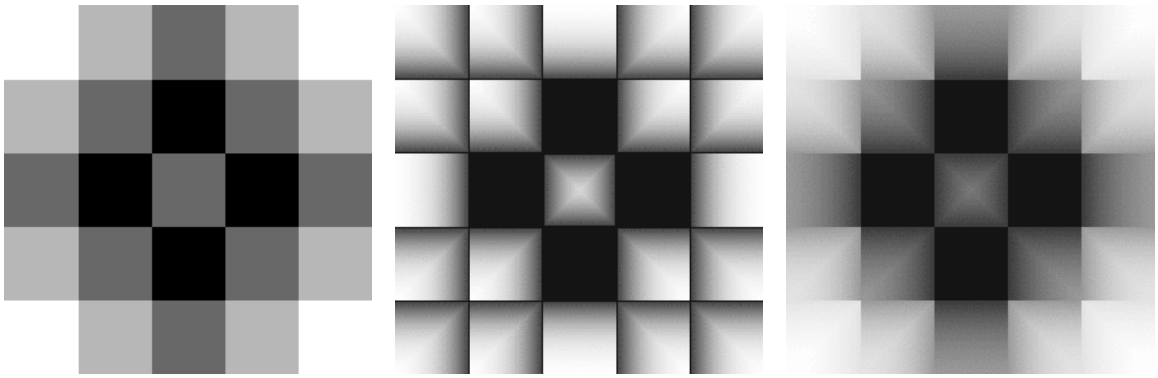
The *topographical distance* between p and q is the minimum of the topographical distances along all paths between p and q:

$$T_f(p,q) = \underset{\pi \in [p \rightsquigarrow q]}{\text{MIN}} T_f^\pi(p,q), \tag{6.6}$$

where the set of all paths from p to q is denoted by $[p \rightsquigarrow q]$. The topographical distance between a point $p \in D$ and a set $A \subseteq D$ is defined as $T_f(p,A) = \text{MIN}_{a \in A} T_f(p,a)$.

We call $(p_0, p_1, \ldots, p_n)$ a *path of steepest descent* from $p_0 = p$ to $p_n = q$ if $p_{i+1} \in \Gamma(p_i)$ for each $i = 0, \ldots, n-1$. A pixel q is said to belong to the *downstream* of p if there exists a path of steepest descent from p to q. A pixel q is said to belong to the *upstream* of p if p belongs to the downstream of q.

The topographical distance has the following property, on which the watershed definition crucially depends.



(a) (b) (c)

Figure 6.18. *Image (a), lower distance image (b) and lower complete image (c).*

Plateau problem

Problems arise when we try to extend the above approach to images which are not lower complete. In such images non-minima plateaus with nonempty interior occur. When we directly apply the above definitions, the topographical distance between interior pixels of a plateau turns out to be identically zero. If we want the watershed to divide such a plateau, additional means to distinguish plateau points are required. Therefore an additional ordering relation between such pixels is introduced. The usual solution is to compute geodesic distances to the lower boundary of the plateau. This can be formalized by first transforming the image to a lower complete image, to which the definitions above then can be applied.

Recall that $\Pi_f^{\downarrow}(p)$ is the set of all descending paths starting in a pixel p and ending in some pixel q with $f(q) < f(p)$, and $\mathsf{length}(\pi)$ is the length of a path π.

Definition 8 (Lower completion) *Let f be a digital grey value image with domain D. Define the function $d : D \to \mathbb{N}$ by*

$$d(p) = \begin{cases} 0 & \text{if } \Pi_f^{\downarrow}(p) = \emptyset \\ \mathrm{MIN}_{\pi \in \Pi_f^{\downarrow}(p)} \mathsf{length}(\pi) & \text{otherwise} \end{cases}$$

Let $L_c = \max_{p \in D} d(p)$. Then the lower completion *f_{LC} of f is defined by*

$$f_{LC}(p) = \begin{cases} L_c \cdot f(p) & \text{if } d(p) = 0 \\ L_c \cdot f(p) + d(p) - 1 & \text{otherwise} \end{cases}$$

The process of lower completion transforms the image f into a lower complete image f_{LC}. An example is given in Fig. 6.18. The function d has the value zero for minima pixels, and for all other pixels p, $d(p)$ equals the length of the shortest path from p to the set of pixels with grey value lower than that of p. We will refer to $d(p)$ as the *lower distance* of p. If f is already

lower complete, then $f_{LC} = f$. An algorithm for lower completion is given in the next section (Algorithm 6.4.5).

By lower completion, we can define an order relation $\sqsubset$ between pixels:

$$x \sqsubset y \iff f_{LC}(x) < f_{LC}(y). \tag{6.7}$$

After lower completion, the function T_{f^*} with $f^* = f_{LC}$ is a proper distance function on $D' \times D'$, where D' equals the domain D from which the minima are excluded.

The particular form of the lower slope and cost function was devised to ensure that steepest descent paths would realize the smallest topographical distance. The mapping $\Gamma(p)$ can be used to define a directed graph by *arrowing* [14, 60] as follows.

Definition 9 *Let $G = (V, E, f)$ be a digital grey value image. The* lower complete graph $G' = (V, E')$ *is the subgraph of G which is defined as follows. For points p having a lower neighbour,*

$$(p, p') \in E' \iff p' \in \Gamma(p) \tag{6.8}$$

On the interior of plateaus, an arc is created from p to p' if the geodesic distance to the lower boundary of the plateau is greater for p than for p', i.e. if $p' \sqsubset p$.

The lower complete graph is acyclic (a DAG).

Definition 10 (Watershed transform by topographical distance)

Let f be a grey value image, with $f^* = f_{LC}$ the lower completion of f. Let $(m_i)_{i \in I}$ be the collection of minima of f. The basin $CB(m_i)$ *of f corresponding to a minimum m_i is defined as the basin of the lower completion of f:*

$$CB(m_i) = \{p \in D \mid \forall j \in I \backslash \{i\} : f^*(m_i) + T_{f^*}(p, m_i) < f^*(m_j) + T_{f^*}(p, m_j)\}, \tag{6.9}$$

and the watershed of f is defined as in (6.2).

So, basically we define the watershed transform by topographical distance of an arbitrary digital grey value image as the watershed transform of its lower completion.

In practice, algorithms to compute the watershed transform for images with plateaus often do not explicitly carry out the lower completion step, but assign plateau pixels to basins in another way. This is the case for algorithms based on so-called ordered queues. As a cautionary note we would like to point out that such algorithmic solutions lead to results which may differ to varying degree from the result of Definition 10, depending on the precise implementation. This will be discussed in more detail in Section 6.4.2.

Lowering the minima values. Meyer states in [60] that the watershed lines will not change if one replaces the values of all minima of f by the value of the deepest one. This statement is correct for Definition 4 of the watershed transform based on immersion, as is easy to verify. But for the definition based on topographical distance this property does in fact not hold, as already observed in [96]. An example illustrating this is given in Fig. 6.23, where there are three minima, two with value 0 and one with value 4. Replacing the value 4 by 0 does change the result. Even more, the effect of lowering the value of this single minimum pixel propagates in a global way through the entire image (the image can be enlarged arbitrarily with the effect propagating accordingly).

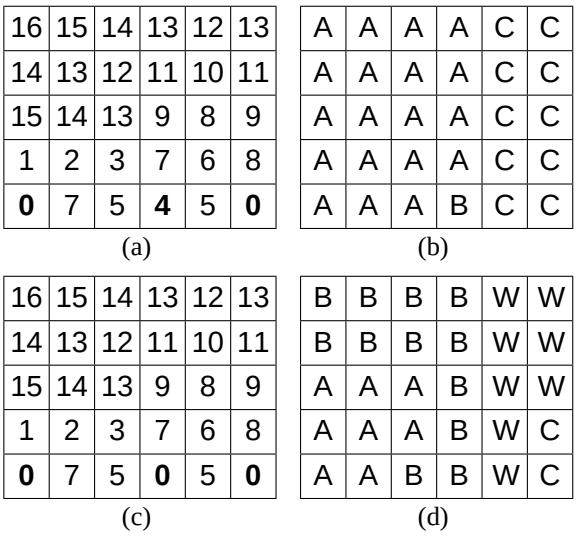


Figure 6.23. *Watershed transform according to topographical distance on the square grid with* 4*-connectivity, showing effect of lowering minima. (a): original image; (b): watershed labelling of (a); (c): image (a) with all minima set to zero. (d): watershed labelling of (c).*

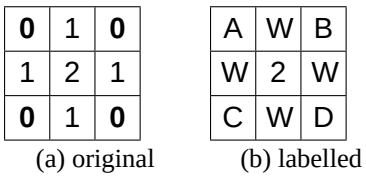


Figure 6.26. *Watershed according to topographical distance (4-connectivity). (a): original image; (b): Output after labelling pixels with grey values* 0 *and* 1.

'Isolated' regions. When computing the watershed transform, regions in the image may arise which are completely surrounded by watershed pixels. An example is given in Fig. 6.26. The center pixel with value 2 has four watershed neighbours, therefore is watershed pixel. In some implementations of watershed transforms by topographical distance, such regions may in fact become temporarily or permanently 'isolated', see [30, 96]. This is a defect of the particular implementation, since, according to Corollary 7, watershed pixels *should* be propagated. Such 'problems' are often solved by ad hoc modifications of the implementation, which still do not correctly implement the definition.

6.3.2.3 Watersheds based on a local condition

Several watershed algorithms exist which do not construct watershed pixels, but instead assign to each pixel the label of some minimum, so that the set of basins tessellates the image plane. Various motivations for such an approach can be given. First of all, 'watershed lines' may in fact comprise large areas (thick watersheds), see Fig. 6.17, although the use of a higher connectivity

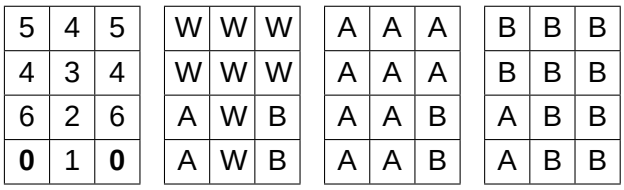


Figure 6.31. *Watershed transform on the 4-connected square grid. (a): original image; (b): result according to topographical distance (Definition 10); (c-d): two watershed labellings consistent with the local condition (Definition 11).*

alleviates the problem. Next, some implementations of the watershed transform by topographical distance have problems with isolated regions caused by watershed pixels, see above. Another reason is efficiency, since a correct determination of watershed pixels generally requires more computation time and memory.

An explicit definition of a watershed transform based on topographical distance which does not construct watershed lines was given by Bieniek et al. [15, 16], by introducing a *local condition*.

Definition 11 *For any image without plateaus, a function L assigning a label to each pixel is called a* watershed segmentation *if:*

1. $L(m_i) \neq L(m_j)\ \forall i \neq j$, *with* $\{m_k\}_{k\in I}$ *the set of minima of* f*;*
2. *for each pixel* p *with* $\Gamma(p) \neq \emptyset$, $\exists p' \in \Gamma(p)$ *with* $L(p) = L(p')$.

Here the condition $\Gamma(p) \neq \emptyset$ means that p has at least one lower neighbour (cf. Definition 5). The new element is that for a given input image, many labellings exist which qualify as a watershed segmentation. Pixels which would have been labelled as watershed points according to Definition 10, are now merged by random choice with a basin belonging to some minimum m_k. For an example, see Fig. 6.31.

The meaning of 'locality' in this definition is that one may subdivide an image in blocks, do a labelling of basins in each block independently, and make the results globally consistent in a final merging step. Such increased locality is very advantageous for parallel implementation of the watershed transform, which is exactly the context in which this local condition was proposed. Note however that locality should not be misinterpreted as saying that the watershed transform now has become a purely local operation: in the merging step, basins in local blocks have to be made consistent, and the resulting global basins can again extend over large regions of the image (for a fuller discussion, see Section 6.5.2.3).

For an input image which would contain watershed pixels according to Definition 10, the output of a watershed algorithm based on Definition 11 is no longer deterministic, but will depend on the order in which pixels are treated during execution of the algorithm. Whereas in the sequential case a deterministic result can be obtained by fixing the scanning order (e.g. raster

scan), this is no longer true for parallel implementation, since in that case the outcome depends on the relative time instants at which different processors treat the pixels, and this is unpredictable in the case of asynchronous processors. Therefore, in principle considerable differences among watershed labellings computed in different runs of the same algorithm may occur, although the effect may be small for natural images.

6.4 Sequential watershed algorithms

Generally spoken, existing watershed algorithms either simulate the flooding process, or directly detect the watershed points. In some implementations, one computes basins which touch, i.e., no watershed pixels are generated at all.

6.4.1 Watershed algorithms by immersion

6.4.1.1 Vincent-Soille algorithm

An implementation of the watershed transform of Definition 4 was presented by Vincent & Soille [102]. Since we want to discuss this implementation in some detail, we reproduce their algorithm here in pseudocode, see Algorithm 6.4.1. In this algorithm there are two steps: (i) sorting the pixels w.r.t. increasing grey value, for direct access to pixels at a certain grey level; (ii) a flooding step, proceeding level by level and starting from the minima. The implementation uses a FIFO queue of pixels, that is, a first-in-first-out data structure on which the following operations can be performed: *fifo_add*$(p, queue)$ adds pixel p at the end of the queue, *fifo_remove*$(queue)$ returns and removes the first element of the queue, *fifo_init*$(queue)$ initializes an empty queue, and *fifo_empty*$(queue)$ is a test which returns true if the queue is empty and false otherwise.

The algorithm assigns a distinct label $lab[\,]$ to each minimum and its associated basin by iteratively flooding the graph using a breadth-first algorithm [23], as follows. In the flooding step, all nodes with grey level h are first given the label MASK. Then those nodes which have labelled neighbours from the previous iteration are inserted in the queue, and from these pixels geodesic influence zones are propagated inside the set of masked pixels. If a pixel is adjacent to two or more different basins, it is marked as a watershed node by the label WSHED. If the pixel can only be reached from nodes which have the same label, the node is merged with the corresponding basin. Pixels which at the end still have the value MASK belong to a set of new minima at level h, whose connected components get a new label. As shown in [102], the time complexity of Algorithm 6.4.1 is linear in the number of pixels of the input image.

Algorithm 6.4.1 *Vincent-Soille watershed algorithm* [102].

```
1: procedure Watershed-by-Immersion
2: INPUT: digital grey scale image G = (D, E, im).
3: OUTPUT: labelled watershed image lab on D.
4: #define INIT − 1          (∗initial value of lab image∗)
5: #define MASK − 2          (∗initial value at each level∗)
```

```
 6: #define WSHED 0                          (∗label of the watershed pixels∗)
 7: #define FICTITIOUS (−1,−1)               (∗fictitious pixel ∉ D∗)
 8: curlab ← 0                               (∗curlab is the current label∗)
 9: fifo_init(queue)
10: for all p ∈ D do
11:    lab[p] ← INIT ; dist[p] ← 0           (∗dist is a work image of distances∗)
12: end for
13: SORT pixels in increasing order of grey values (minimum h_min, maximum h_max)
14:
15: (∗Start Flooding∗)
16: for h = h_min to h_max do                (∗Geodesic SKIZ of level h−1 inside level h∗)
17:    for all p ∈ D with im[p] = h do       (∗mask all pixels at level h∗)
18:                                          (∗these are directly accessible because of the sorting step∗)
19:       lab[p] ← MASK
20:       if p has a neighbour q with (lab[q] > 0 or lab[q] = WSHED) then
21:          (∗Initialize queue with neighbours at level h of current basins or watersheds∗)
22:          dist[p] ← 1 ; fifo_add(p, queue)
23:       end if
24:    end for
25:    curdist ← 1 ; fifo_add(FICTITIOUS, queue)
26:    loop                                  (∗extend basins∗)
27:       p ← fifo_remove(queue)
28:       if p = FICTITIOUS then
29:          if fifo_empty(queue) then
30:             BREAK
31:          else
32:             fifo_add(FICTITIOUS, queue) ; curdist ← curdist + 1 ;
33:             p ← fifo_remove(queue)
34:          end if
35:       end if
36:       for all q ∈ N_G(p) do              (∗labelling p by inspecting neighbours∗)
37:          if dist[q] < curdist and (lab[q] > 0 or lab[q] = WSHED) then
38:             (∗q belongs to an existing basin or to watersheds∗)
39:             if lab[q] > 0 then
40:                if lab[p] = MASK or lab[p] = WSHED then
41:                   lab[p] ← lab[q]
42:                else if lab[p] ≠ lab[q] then
43:                   lab[p] ← WSHED
44:                end if
45:             else if lab[p] = MASK then
46:                lab[p] ← WSHED
47:             end if
48:          else if lab[q] = MASK and dist[q] = 0 then   (∗q is plateau pixel∗)
49:             dist[q] ← curdist + 1 ; fifo_add(q, queue)
50:          end if
```

```
51:       end for
52:     end loop
53:     (∗detect and process new minima at level h∗)
54:     for all p ∈ D with im[p] = h do
55:       dist[p] ← 0            (∗reset distance to zero∗)
56:       if lab[p] = MASK then          (∗ p is inside a new minimum∗)
57:         curlab ← curlab + 1 ;        (∗create new label∗)
58:         fifo_add(p, queue) ; lab[p] ← curlab
59:         while not fifo_empty(queue) do
60:           q ← fifo_remove(queue)
61:           for all r ∈ N_G(q) do          (∗inspect neighbours of q∗)
62:             if lab[r] = MASK then
63:               fifo_add(r, queue) ; lab[r] ← curlab
64:             end if
65:           end for
66:         end while
67:       end if
68:     end for
69:   end for
70:   (∗End Flooding∗)
```

The Vincent-Soille algorithm in fact does not implement the recursion (6.3), for the following reasons (the line numbers mentioned refer to the pseudocode of Algorithm 6.4.1).

1. At level h only pixels with grey value h are masked for flooding (line 17), instead of all non-basin pixels of level $\leq h$, as the definition would require (see the discussion in Section 6.3.2.1).

2. Not only labels of catchment basins are propagated, but also labels of WSHED-pixels (line 20). The need for this is a consequence of the previous point. Since the algorithm tries to classify pixels as WSHED-pixels at the *current* grey level, watershed labels have to be propagated, because it may be the case that pixels with grey value h only have WSHED-pixels in their neighbourhood.

3. A pixel which is adjacent to two different basins, and therefore initially gets labelled WSHED, is allowed to be overwritten at the current grey level by the label of another neighbouring pixel, if that pixel is part of a basin (lines 40-41). The motivation given in [102] is that otherwise 'deviated' watershed lines may result. This statement is probably based on an intuitive expectation for the case of functions in continuous space. From our point of view, an assessment of the correctness of the implementation should be based solely on agreement with the definition.

It is not very difficult to modify Algorithm 6.4.1 in order to implement the recursion (6.3) exactly. In line 17 all pixels with $im[p] \leq h$ have to be masked, the queue has to be initialized with basin pixels only (drop the disjunct $lab[q] =$ WSHED in line 20), the resetting of distances (line

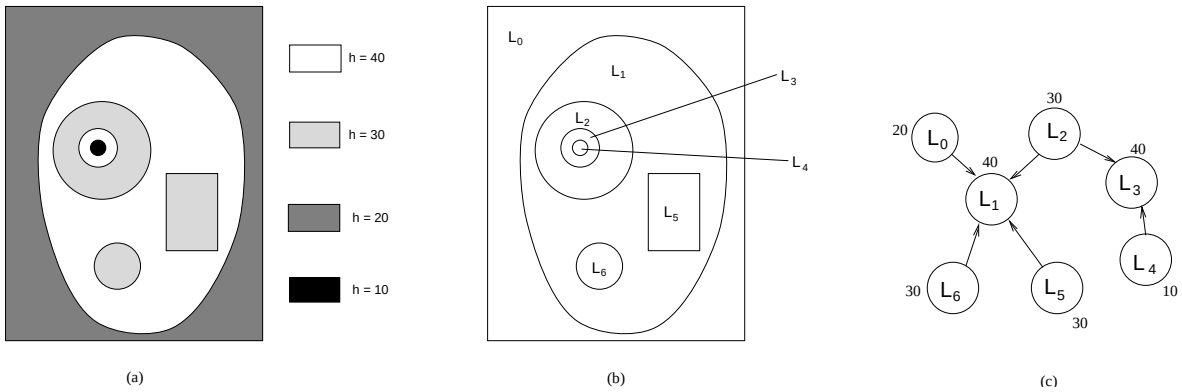


Figure 6.32. *(a) input image. (b) labelled level components. (c) components graph, with grey values of the nodes indicated.*

55) has to be done in line 19, and the propagation rules in lines 36-51 have to be slightly changed. Note, however, that the theoretical time complexity would change from linear to quadratic in the number of pixels of the input image, due to repeated processing of 'watershed' pixels, although in practice the number of such pixels may actually be rather small.

Remark.

In [79] we tried to formalize what the Vincent-Soille algorithm computes by defining a modified recursion as follows:

$$\begin{cases} X_{h_{min}} &= \{p \in D \mid f(p) = h_{min}\} \\ X_{h+1} &= X_h \cup \text{MIN}_{h+1} \cup (IZ_{T_{h+1}}(X_h) \backslash T_h), \qquad h \in [h_{min}, h_{max}) \end{cases} \tag{6.10}$$

The '$\backslash T_h$' term in (6.10) was introduced to ensure that at level $h+1$ only pixels with grey value $h+1$ are added to existing basins. In the example of Fig. 6.13, the pixel in the second row, second column remains labelled as WSHED according to (6.10). However, also this modified recursion does not always correctly represent the implementation of Algorithm 6.4.1: it is possible that a catchment basin becomes disconnected by the '$\backslash T_h$' term. In fact, we have been unable to find a recursion which formalizes what actually is computed by Algorithm 6.4.1. □

6.4.1.2 Components graph algorithm

A straightforward parallel implementation of the Vincent-Soille algorithm is difficult for images in which plateaus occur. Therefore, an alternative approach was proposed in [51], in which the image is first transformed to a directed valued graph with distinct neighbour values, called the *components graph of* f. On this graph the watershed transform can be computed by a simplified version of the Vincent-Soille algorithm, where FIFO queues are no longer necessary, since there are no plateaus in the graph. The steps are as follows.

1. Consider the input image as a valued graph (V, E, f), where $f(p)$ denotes the grey value of pixel p, $p \in V$. Transform this to the components graph (V^*, E^*, f^*) defined as follows.

Algorithm 6.4.2 *Watershed transform w.r.t. topographical distance based on image integration via the Dijkstra-Moore shortest paths algorithm.*

```
1: procedure ShortestPathWatershed;
2: INPUT: lower complete digital grey scale image G = (V,E,im) with cost function cost.
3: OUTPUT: labelled image lab on V.
4: #define WSHED 0                    (* label of the watershed pixels *)
5: (* Uses distance image dist. On output, dist[v] = im[v], for all v ∈ V. *)
6:
7: for all v ∈ V do        (* Initialize *)
8:    lab[v] ← 0 ; dist[v] ← ∞
9: end for
10: for all local minima m_i do
11:    for all v ∈ m_i do
12:       lab[v] ← i ; dist[v] ← im[v]        (* initialize distance with values of minima *)
13:    end for
14: end for
15: while V ≠ ∅ do
16:    u ← GetMinDist(V)        (* find u ∈ V with smallest distance value dist[u] *)
17:    V ← V \ {u}
18:    for all v ∈ V with (u,v) ∈ E do
19:       if dist[u] + cost[u,v] < dist[v] then
20:          dist[v] ← dist[u] + cost(u,v)
21:          lab[v] ← lab[u]
22:       else if lab[v] ≠ WSHED and dist[u] + cost[u,v] = dist[v] and lab[v] ≠ lab[u] then
23:          lab[v] = WSHED
24:       end if
25:    end for
26: end while
```

All pixels of a level component C_h at level h are represented by a single node $v \in V^*$: $v = \{p \in V | p \in C_h\}$, with $f^*(v) = h$. A pair (v,w) of level components is an element of E^* if and only if $\exists (p \in v, q \in w : (p,q) \in E \wedge f(p) < f(q))$, cf. Fig. 6.32.

2. Compute the watershed transform of the directed graph.

3. Transform the labelled graph back to an image. Pixels corresponding to a watershed node are coloured white, the other pixels black. This yields a binary image with plateaus representing watersheds of the original image. Thin watersheds can be obtained by computing a skeleton of this image, for which different skeleton algorithms can be used.

6.4.2 Watershed algorithms by topographical distance

Several shortest paths algorithms for the watershed transform with respect to topographical distance can be found in the literature [14, 60, 61].

Algorithm 6.4.3 *Watershed transform w.r.t. topographical distance by hill climbing.*

```
 1: procedure Hill Climbing
 2: INPUT: lower complete digital grey scale image (V,E,im).
 3: OUTPUT: labelled image lab on V.
 4: #define WSHED 0                    (*label of the watershed pixels*)
 5:
 6: LabelInit      (*initialize image lab with distinct labels for minima*)
 7:                 (*and special label MASK for all other pixels*)
 8: S ← {p ∈ V | ∃q ∈ N_G(p) : im[p] ≠ im[q]}      (*interior pixels of minima excluded*)
 9: while not empty(S) do
10:    select point p ∈ S with minimal grey value
11:    remove p from S
12:    for all q ∈ Γ^{-1}(p) ∩ S do      (*label steepest upper neighbours of p*)
13:       if lab[q] = MASK then
14:          lab[q] ← lab[p]
15:       else if lab[q] ≠ WSHED and lab[q] ≠ lab[p] then
16:          lab[q] = WSHED
17:       end if
18:    end for
19: end while
```

Ordered algorithms. The nodes for which the shortest topographical distance is known are ordered w.r.t. their distance. These methods are based upon the shortest paths algorithm associated with the names of Dijkstra [27] and Moore [69].
a. *integration*: this algorithm is based on integration of the lower slope of the image, by propagating distances starting from the regional minima. The distances are related to the lower slope of the image through the cost function (6.5). On output, the distance value of a pixel p equals $f(p)$, where f is the input image. The pseudocode is given in Algorithm 6.4.2, which is described in more detail in Section 6.4.2.2.
b. *hill climbing*: The geodesics between points of a basin and the corresponding minimum are paths of steepest descent. This relation may be inverted as follows. Label all minima with distinct labels. Starting from the boundary pixels of the minima, label all pixels q in the set $\Gamma^{-1}(p)$ of all steepest upper neighbours of the current pixel p by the label of p, unless q is already labelled and the label differs from that of p, in which case q is classified as a watershed pixel. The pseudocode is given in Algorithm 6.4.3, see Section 6.4.2.3 for details.

Unordered algorithms. The shortest path algorithm of Berge [11] assumes no order on the treatment of pixels, so that classical raster scanning modes can be used. This algorithm can be adapted for flooding from the minima and solving the eikonal equation [95]. The implementation is based on an iterative algorithm [60] which integrates the lower slope of the input image, see Algorithm 6.4.4. In [60] a variant is mentioned based on propagation of labelled pixels to steepest upper neighbours, as in hill climbing.

In [60] several slightly different versions of the above algorithms are presented which do

Algorithm 6.4.4 *Watershed transform w.r.t. topographical distance by sequential scanning based on image integration.*

```
 1: procedure Sequential scanning
 2: INPUT: lower complete image im on a digital grid G = (D,E) with cost function cost.
 3: OUTPUT: labelled image lab on D.
 4: #define WSHED 0                    (* label of the watershed pixels *)
 5: (* Uses distance image dist. On output, dist[v] = im[v], for all v ∈ D. *)
 6:
 7: for all v ∈ D do        (* Initialize *)
 8:    lab[v] ← 0 ; dist[v] ← ∞
 9: end for
10: for all local minima m_i do
11:    for all v ∈ m_i do
12:       lab[v] ← i ; dist[v] ← im[v]        (* initialize distance with values of minima *)
13:    end for
14: end for
15: stable ← true        (* stable is a boolean variable *)
16: repeat
17:    for all pixels u in forward raster scan order do
18:       Propagate (u)
19:    end for
20:    for all pixels u in backward raster scan order do
21:       Propagate (u)
22:    end for
23: until stable
24:
25: procedure Propagate (u)
26: for all v ∈ N_G(u) in the future (w.r.t. scan order) of u do
27:    if dist[u] + cost[u,v] < dist[v] then
28:       dist[v] ← dist[u] + cost(u,v)
29:       lab[v] ← lab[u]
30:       stable ← false
31:    else if lab[v] ≠ WSHED and dist[u] + cost[u,v] = dist[v] and lab[v] ≠ lab[u] then
32:       lab[v] = WSHED
33:       stable ← false
34:    end if
35: end for
```

not produce watershed labels (lines 21-22 in Algorithm 6.4.2, lines 14-15 in Algorithm 6.4.3 and lines 30-32 in Algorithm 6.4.4 are omitted), and therefore are not exact implementations of Definition 10. All pixels are merged with some basin, so that, dependent on the order in which pixels are treated, different results may be produced. Unfortunately, a discussion of this point is missing in [60]. In fact, those algorithms are in agreement with the local definition of the watershed transform, as discussed in Section 6.3.2.3.

Algorithm 6.4.5 *Algorithm for lower completion using a* FIFO *queue.*

```
 1: procedure LowerCompletion
 2: INPUT: digital grey scale image G = (D,E,im).
 3: OUTPUT: lower complete image G' = (D,E,lc).
 4:
 5: fifo_init(queue)
 6: for all p ∈ D do        (*Initialize queue with pixels that have a lower neighbour*)
 7:    lc[p] ← 0
 8:    if p has a lower neighbour then
 9:       fifo_add(p,queue)
10:       lc[p] ← −1
11:    end if
12: end for
13: dist ← 1        (*dist is an integer variable*)
14: fifo_add(FICTITIOUS,queue)        (*insert fictitious pixel*)
15: while not fifo_empty(queue) do
16:    p ← fifo_remove(queue)
17:    if p = FICTITIOUS then
18:       if not fifo_empty(queue) then
19:          fifo_add(FICTITIOUS,queue)
20:          dist ← dist + 1
21:       end if
22:    else
23:       lc[p] ← dist
24:       for all q ∈ N_G(p) with (im[q] = im[p] and lc[q] = 0) do
25:          fifo_add(q,queue)
26:          lc[q] ← −1        (*to prevent from queuing twice*)
27:       end for
28:    end if
29: end while
30:
31: for all p ∈ D do        (*Put the lower complete values in the output image*)
32:    if lc[p] ≠ 0 then
33:       lc[p] = dist · im[p] + lc[p] − 1
34:    else
35:       lc[p] = dist · im[p]
36:    end if
37: end for
```

To solve the plateau problem, the image may first be made lower complete. This can be done by a linear-time breadth-first algorithm using a FIFO queue [23] to propagate distances, cf. Algorithm 6.4.5. In the case of the ordered algorithms, an alternative to lower completion as preprocessing is to use ordered queues. This will be discussed in more detail below. But first we consider the initial step which is necessary in these algorithms, i.e., detection of the minima.

6.4.2.1 Minima detection

Usually a flooding algorithm based on FIFO queues is used for minima detection [52, 64, 67]. However, the UNION-FIND algorithm for implementing disjoint sets [90], see also [23, 89], can be used for computing connected components, and therefore for minima detection, as well. In practice the UNION-FIND algorithm outperforms the flooding algorithm.

Algorithm 6.4.6 *Computing level components by breadth-first search using a* FIFO *queue.*

```
 1: procedure LevelComponents
 2: INPUT: digital grey scale image G = (V, E, im).
 3: OUTPUT: image lab on V, with labelled level components.
 4: #define INIT − 1                    (∗initial value of lab image∗)
 5:
 6: for all p ∈ D do
 7:    lab[p] ← INIT
 8: end for
 9: curlab ← 1        (∗curlab is the current label∗)
10: fifo_init(queue)
11:
12: for all p ∈ V with lab[p] = INIT do
13:    lab[p] ← curlab
14:    fifo_add(p, queue)
15:    while not fifo_empty(queue) do
16:       s ← fifo_remove(queue)
17:       for all q ∈ N_G(s) with im[s] = im[q] do
18:          if lab[q] = INIT then
19:             lab[q] ← curlab
20:             fifo_add(q, queue)
21:          end if
22:       end for
23:    end while
24:    curlab ← curlab + 1
25: end for
```

FIFO algorithm. Standard 'flooding' (breadth-first) implementations use a FIFO queue to find the level components, i.e. the connected components of pixels of constant grey value, cf. Algorithm 6.4.6. For each component a pixel is stored in an empty FIFO queue, followed by a flooding process which runs until the queue is empty. The flooding process consists of removing a pixel from the queue, and inserting into the queue its neighbours with the same grey value that have not been labelled yet. The time complexity is linear in the number of edges of the graph. In practice, the image is a graph with a fixed connectivity k, so that the complexity of the algorithm is linear in the number of pixels of the image. We cannot construct an algorithm with a better time complexity. However, the minimally required size of the queue is not known in advance,

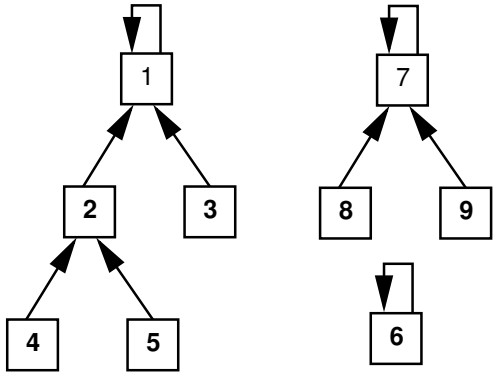


Figure 6.33. *Disjoint set forest of sets of integers* $\{1,2,3,4,5\}$, $\{6\}$, $\{7,8,9\}$.

and memory is addressed in a very unstructured manner, causing performance degradation on virtual memory and especially on parallel computers, since it requires a lot of synchronization.

UNION-FIND algorithm. In the UNION-FIND algorithm, disjoint sets are stored in trees, forming a *disjoint-set forest*, in which each node p is pointing to its parent $parent[p]$; Fig. 6.33 gives an example where sets of integers are stored. A node p of a tree is called the *root* of the tree if $parent[p] = p$. For each tree, the root is chosen as the *representative* of the set stored in the tree.

If two sets are merged (united), it is sufficient to change the root of one of the trees such that it points to the root of the other tree. To prevent the height of the tree from increasing too drastically, resulting in longer search times to find representatives, *path compression* is applied. This means that not only the root, but all nodes on the path from an arbitrary node p to the root, are set to point directly to the root. By this technique the length of paths to roots rarely exceeds 3 in practical cases.

In [89] Tarjan uses a second technique, called *union by rank*, to prevent the height of the trees from growing too drastically as well, keeping the resulting tree reasonably balanced when merging two trees. In [89] it is shown that the time complexity of the algorithm using both techniques, for an input of size N, is $O(N\alpha(N,N))$, where $\alpha(N,N)$ is the inverse of the Ackermann function, whose value is smaller than 5 if N is of the order 10^{80}. So, in practice, this algorithm can be regarded to run in linear time with respect to its input. When using the algorithm for computing connected components in images, it turns out that only the path compression technique really pays off, and therefore the ranking technique is omitted.

Using the disjoint-set technique, the labelling of connected components can easily be performed in a *scan-line* fashion, cf. Algorithm 6.4.7. In this case, the nodes of the trees are pixels. Let $\prec$ denote the lexicographical order between pixels. E.g., in a 2-D image with pixels $p = (i, j), q = (k, l)$, $p \prec q$ denotes that $(i < k) \vee ((i = k) \wedge (j < l))$; also, $p \preceq q \equiv (p \prec q \vee p = q)$. In the scan-line algorithm, pixels are visited in lexicographical order. Let p_0 denote the first pixel, and $curpix$ the current pixel, during scanning. Then the following order on the array $parent$ is maintained: $\forall(p : p_0 \preceq p \preceq curpix : p_0 \preceq parent[p] \preceq p)$. Since this order prevents cycles, we

can iteratively evaluate *parent* to find the root of the tree containing p, denoted by FindRoot(p).

Let p be the current pixel. If p has no neighbours q (with $q \prec p$) with the same image value, a new set is created by setting $parent[p]$ to p. If there exist neighbouring pixels q (with $q \prec p$) that have the same grey value as p, the representatives of these neighbours are computed and the (lexicographically) smallest of them is chosen as the representative of the union of the sets containing these neighbours. Then the paths of these neighbours are compressed using PathCompress( ), and p is merged with this set. In a second pass through the input image, the output image lab is created. All root pixels get a distinct label; for any other pixel p its path is compressed, making explicit use of the order imposed on *parent* (see line 29 in Algorithm 6.4.7), and p gets the label of its representative.

This algorithm can be used for the computation of connected components (see chapter 3) in images of any dimension, size and connectivity, in contrast to the algorithm of Rosenfeld-Pfaltz [82], which works only for 2-dimensional images using 4-connectivity. The same restriction holds for the UNION-FIND algorithm in [32] which performs in exact linear time by post-processing each scan line. Also an *in-situ* variation of the algorithm is possible in which the array *parent* has been removed. In this case the image lab plays the role of output image and *parent* array at the same time.

We now resume our discussion of watershed algorithms based on topographical distance.

6.4.2.2 Image integration by the Dijkstra-Moore shortest paths algorithm.

Given a directed weighted graph $G = (V, E, w)$, with $w : E \to \mathbb{N}$ a nonnegative weight function on the arcs, the Dijkstra-Moore algorithm computes the length of the shortest path from a source node s to every other node v [23, 27]. This algorithm can be simply adapted for computing the watershed transform. First, an edge (p, p') in the image is considered as a pair of arcs (p, p') and (p', p) with the same weight. Next, a label image lab and a distance image $dist$ are introduced, just as in the case of Algorithm 6.4.4, where $lab[v]$ is the index of the minimum nearest to v, and $dist[v]$ is the distance to this minimum [52]. From each minimum a wavefront is started, labelled by the index of the minimum it started in, and the distance is initialized with the value of the minimum, cf. (6.9). If wavefront i reaches a node v after it has propagated over a distance ℓ, and ℓ is less than $dist[v]$, the value ℓ is placed in $dist[v]$, while $lab[v]$ is set to i. If a node v is reached by another wavefront that has propagated over the same distance but originated from a different minimum (if it already carries the label WSHED this is also the case), $lab[v]$ is set to the artificial value WSHED, designating that v is a watershed pixel. For the pseudo-code, see Algorithm 6.4.2.

If the input image has non-minima plateaus, it may be first lower completed. An alternative is to keep track of distances to the lower border of plateaus during execution of the algorithm. This can be achieved by the use of ordered queues.

Implementation by ordered queues. The function *GetMinDist* in Algorithm 6.4.2 can be implemented such that it has a time complexity which is linear in the number of pixels of the image. This can be realized with a data structure called 'hierarchical' or 'ordered' queue (OQ), which is a priority queue of N FIFO queues, one queue for each of the N grey values in the image, such that the lower grey values have higher priority [14, 59]. The OQ processes lower grey levels

Algorithm 6.4.7 *Scan-line algorithm for labelling level components based on disjoint sets.*

```
 1: procedure UNION-FIND-ComponentLabelling
 2: INPUT: grey scale image im on digital grid G = (D,E).
 3: OUTPUT: image lab on D, with labelled level components.
 4: (*Uses array parent of pointers.*)
 5:
 6: (*First pass*)
 7: for all p ∈ D in lexicographical order do
 8:    r ← p
 9:    for all q ∈ N_G(p) with q ≺ p do
10:       if im[q] = im[p] then
11:          r ← r MIN FindRoot(q)        (*min denotes minimum w.r.t. lexicographical order*)
12:       end if
13:    end for
14:    parent[p] ← r
15:    for all q ∈ N_G(p) with q ≺ p do        (*compress paths*)
16:       if im[q] = im[p] then
17:          PathCompress(q,r)
18:       end if
19:    end for
20: end for
21:
22: (*Second pass*)
23: curlab ← 1        (*curlab is the current label*)
24: for all p ∈ D in lexicographical order do
25:    if parent[p] = p then        (*p is a root pixel*)
26:       lab[p] = curlab
27:       curlab = curlab + 1
28:    else
29:       parent[p] = parent[parent[p]]        (*Resolve unresolved equivalences*)
30:       lab[p] = lab[parent[p]]
31:    end if
32: end for
33:
34: function FindRoot(p : pixel)
35: while parent[p] ≠ p do
36:    r ← parent[p] ; p ← r
37: end while
38:  return r
39:
40: procedure PathCompress(p : pixel, r : pixel)
41: while parent[p] ≠ r do
42:    h ← parent[p] ; parent[p] ← r ; p ← h
43: end while
```

before higher ones, and is initialized with the labelled border pixels of minima. Pixels with grey value h are inserted in the FIFO queue with priority level h of the OQ. Pixels are removed from the OQ by priority, and propagate their labels to (i) non-labelled neighbouring pixels, which are inserted in the OQ, or to (ii) neighbouring labelled pixels still in the OQ, which change to watershed pixels if the propagated label differs from the current label. By using the priority order of grey values, pixels always propagate labels to steepest upper neighbours, except on plateaus, where synchronous breadth-first propagation of labels coming from different pixels of the lower border takes place. Thus an OQ automatically implements a hierarchical order relation between pixels, so that preprocessing to make the input image lower complete can be avoided.

It should be noted however, that the OQ does not always give exactly the same result as when the input image is first lower completed. For example, when the image has a plateau whose pixels, after lower completion, are assigned to different basins without any pixel being labelled as watershed pixel (no pixel is equidistant to two or more minima), the OQ algorithm may nevertheless introduce a watershed line at points where wavefronts coming from different parts of the boundary meet. The exact location of this watershed line is dependent on the processing order, and is biased towards that part of the lower boundary from which the propagation proceeded last.

Remark. Algorithm 6.4.2 requires updating of the set V: distances and labels are only propagated to pixels which are still in V. In the ordered queue implementation, V is the set of pixels which have not yet entered the OQ, or are still in it. In the case of a lower complete image, one may instead propagate from a pixel u to *all* neighbours v of u: since the cost function is positive (except on minima plateaus), the computed distance to an already processed pixel $v \notin V$ will always increase, so the algorithm will not change anything for such a pixel v. This entails redundant computation, but has the advantage that no memory is needed to encode the set V. However, when the OQ implementation is used for an image which is not lower complete, and the set V is not properly encoded, a broadening of the watershed line may occur on the interior of plateaus, where the cost function is identically zero. □

6.4.2.3 Hill climbing

Compared to image integration, hill climbing is much simpler since no distances have to be computed: labels are simply propagated to all steepest upper neighbours, see Algorithm 6.4.3. For a lower complete image, determination of the upstream set $\Gamma^{-1}(p)$ of a pixel p only requires local computation. Again, if an image contains non-minima plateaus, it may first be lower completed. Alternatively, just as above, ordered queues can be used.

If the version of the algorithm is used which does not compute watershed pixels, and the distance values on the edges of the underlying grid are equal to 1 ($d(p,q) = 1$ in Eq. (6.4)), such as is the case for the 4-connected and 8-connected neighbourhoods mostly used in practice, one may simply replace the upstream set $\Gamma^{-1}(p)$ by all unlabelled neighbours q of p. Because the algorithm processes pixels with lowest grey value first, an unlabelled neighbour of a pixel p is necessarily in the upstream of p, and a labelled pixel never has to be inspected again, since no watershed labels are assigned. This implies that the initial computation of lower distances and cost function can be avoided, leading to a time and memory efficient implementation. But, of course, the result is not exact and dependent on the processing order.

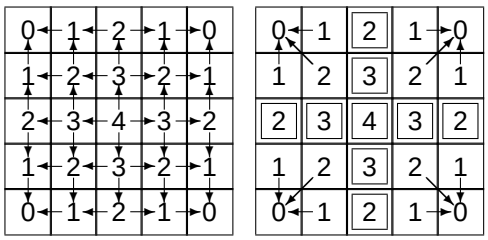


Figure 6.34. *Left: image and its corresponding DAG; right: graph after resolving (watershed pixels are surrounded by a box).*

6.4.2.4 Watershed transform by UNION-FIND algorithm

The UNION-FIND algorithm described in Section 6.4.2.1 can be modified to compute the watershed transform itself [53], by the following steps.

1. First, plateaus have to be removed from the image f by computing the lower completion f_{LC} of f, see Algorithm 6.4.5. The last loop in the algorithm can be slightly adapted to label the minima pixels of f (i.e., pixels p with $lc[p] = 0$) as well.

2. From the lower complete image, the lower complete graph $G' = (V, E')$ is constructed (see Definition 9), which is a directed acyclic graph (DAG). See Fig. 6.34 for an example. The DAG is stored in an array *sln*, where $sln[p,i]$ is a pointer to the i^{th} *steepest lower neighbour* of pixel p (the number of steepest lower neighbours is at most the connectivity). For each minimum m, one pixel $r \in m$ is chosen as the representative of this minimum, and a pointer is created from r to itself. The array *sln* plays the role of *parent* in the level components algorithm, but note that a node can now have more than one 'parent' (steepest lower neighbour). Therefore the graph G is not a disjoint set forest, as in the case of connected components. The DAG can be constructed in a single pass scan-line algorithm, in which for each pixel only its neighbours are referenced.

3. The last step is to apply the UNION-FIND algorithm to the DAG. The first pass is similar to that of Algorithm 6.4.7. The resolving step has to be modified so that watershed pixels can be detected, which are points having paths in the DAG to distinct roots. For the pseudo-code of the resolving algorithm, which closely resembles Tarjan's *FindRoot* operation [89], see Algorithm 6.4.8.

Results. We applied the algorithm (both stages) to a number of test images (see Fig. 6.35). The results shown in Table 6.1 are for hundred runs, and are performed on square images of sizes 256×256, 512×512, and 1024×1024 respectively. The computer used is a 300MHz Pentium PC, with 64 Mb RAM memory, and 512 Kb cache memory.

The image 'blobs' is a binary image with very large minima plateaus (more than half of the pixels are in a minimum), resulting in short root-paths in the DAG. This explains the significant

Algorithm 6.4.8 *Watershed transform w.r.t. topographical distance based on disjoint sets.*

```
 1: procedure UNION-FIND-Watershed
 2: INPUT: lower complete graph G' = (V, E').
 3: OUTPUT: labelled image lab on V.
 4: #define WSHED 0            (* label of the watershed pixels *)
 5: #define W (-1,-1)          (* fictitious coordinates of the watershed pixels *)
 6:
 7: LabelInit                  (* initialize image lab with distinct labels for minima *)
 8:
 9: for all p ∈ V do           (* give p the label of its representative *)
10:    rep ← Resolve(p)
11:    if rep ≠ W then
12:       lab[p] ← lab[rep]
13:    else
14:       lab[p] ← WSHED
15:    end if
16: end for
17:
18: function Resolve(p : pixel)
19: (* Recursive function for resolving the downstream paths of the lower complete graph. *)
20: (* Returns representative element of pixel p, or W if p is a watershed pixel *)
21: i ← 1 ; rep ← (0,0)        (* some value such that rep ≠ W *)
22: while (i ≤ CON) and (rep ≠ W) do        (* CON indicates the connectivity *)
23:    if (sln[p,i] ≠ p) and (sln[p,i] ≠ W) then
24:       sln[p,i] ← Resolve(sln[p,i])
25:    end if
26:    if i = 1 then
27:       rep ← sln[p,1]
28:    else if sln[p,i] ≠ rep then
29:       rep ← W
30:       for j ← 1  →  CON do
31:          sln[p,j] ← W
32:       end for
33:    end if
34:    i ← i + 1
35: end while
36: return rep
```

shorter running time for this image compared to the other ones. By doubling the image dimensions, the number of pixels increases by a factor of 4. Since all phases of the algorithm are performed in (nearly) linear time with respect to the image size, we expect to find this reflected in the timings. On average we find that the running time increases by a factor of 4.2, which is quite close to linear behavior. The images 'blobs', 'chess', and 'waves' are artificially generated images, with relatively few minima, while the other images are camera-made, containing a lot

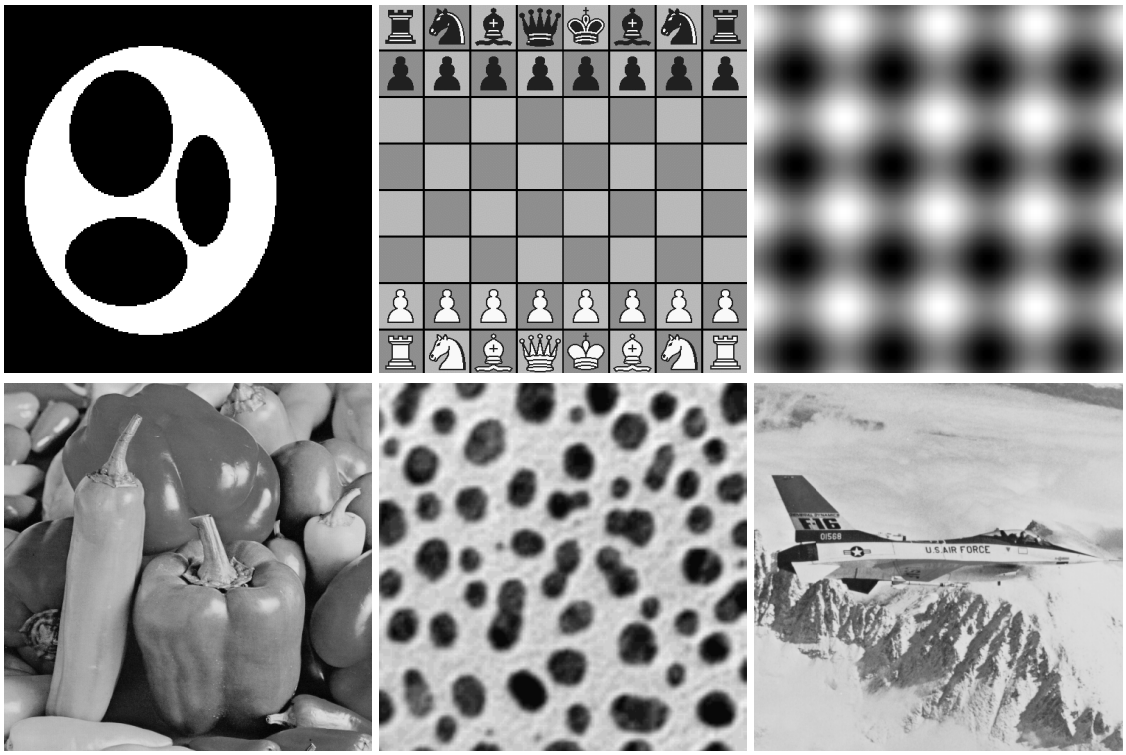


Figure 6.35. *Test images (from top left to bottom right): (a) blobs (b) chess (c) waves (d) peppers (e) particles (f) aircraft.*

Table 6.1. *Timing results for watershed by* UNION-FIND

image	minima	256	512	1024
blobs	4	17.2	74.8	313
chess	67	43.2	178	716
waves	20	36.3	165	720
peppers	44426	37.0	170	712
particles	359	41.6	182	756
aircraft	19053	38.7	174	724

of noise and minima. We see that the number of minima has little effect on the total running time. The timings we find are comparative with the ones found in the literature for other algorithms ([15, 102]). The main interest of our algorithm is the second stage, since it can be parallelized on shared memory computers with very little synchronization overhead, while most other algorithms are difficult to parallelize as a result of global dependencies.

This technique computes the exact watershed transform by topographical distance [60]. A similar approach was developed by Bieniek et al. [16], based on earlier work [15] on parallel implementation of the watershed transform. However, these authors use the local condition in which no watershed pixels are computed (see Sections 6.3.2.3, 6.5.2.3); when several steepest

5	4	3	2	3	4	5
4	3	2	1	2	3	4
3	2	1	**0**	1	2	3
2	1	**0**	1	**0**	1	2
3	2	1	**0**	1	2	3
4	3	2	1	2	3	4
5	4	3	2	3	4	5

(a)

W	W	W	B	W	W	W
W	W	W	B	W	W	W
W	W	W	B	W	W	W
A	A	A	W	C	C	C
W	W	W	D	W	W	W
W	W	W	D	W	W	W
W	W	W	D	W	W	W

(b)

W	B	B	B	B	B	W
A	W	B	B	B	W	C
A	A	W	B	W	C	C
A	A	A	W	C	C	C
A	A	W	D	W	C	C
A	W	D	D	D	W	C
W	D	D	D	D	D	W

(c)

Figure 6.17. *Watershed transform on the square grid with* 4*-connectivity, showing thick watersheds. (a): original image; (b): result according to topographical distance (Definition 3, with T_f as defined in (6.6)); (c): result according to immersion (Definition 4).*

Proposition 6 *Let $f(p) > f(q)$. A path π from p to q is of steepest descent if and only if $T_f^{\pi}(p,q) = f(p) - f(q)$. If a path π from p to q is not of steepest descent, $T_f^{\pi}(p,q) > f(p) - f(q)$.*

This proposition implies that paths of steepest descent are the geodesics (shortest paths) of the topographical distance function. With the introduction of the topographical distance for digital images, the definition of catchment basins and watersheds is the same as for the continuous case, cf. Definition 3.

It is a consequence of Proposition 6 that $CB(m_i)$ is the set of points in the upstream of a single minimum m_i. The watershed consists of the points p which are in the upstream of at least two minima, i.e., there are at least two paths of steepest descent starting from p which lead to different minima. Also, the following corollary is obvious.

Corollary 7 *Any pixel in the upstream of a watershed pixel is itself a watershed pixel.*

An example of the watershed transform according to topographical distance is given in Fig. 6.7(d-e). Note that the result differs from that obtained by immersion according to Definition 4. A consequence of Definition 3 in the digital case is the occurrence of *thick watersheds*, meaning that the watershed pixels do not form one-pixel thick lines but extended areas. An example for the case of 4-connectivity is given in Fig. 6.17. The result according to simulated immersion is given for comparison; although thick watersheds also occur for this watershed definition, they tend to be less pronounced.

Remark. A distance transform [81] on a digital grid (with unit distance values on the edges) of a binary image b produces a grey value image f whose cost function equals 1 on every edge outside the minima of f. The watershed of f therefore equals the SKIZ of b [60]. □

Next we consider images which are not lower complete.

lower neighbours exist, one of them is arbitrarily chosen. Therefore, that algorithm, sometimes referred to as *rainfalling*, is a variant of the watershed transform by UNION-FIND, where the graph is not a DAG, but a disjoint-set forest.

6.5 Parallelization

In this section we first make some general remarks about parallel computer systems and parallel programming. Then a review of parallelization strategies for the watershed transform is given, for both distributed and shared memory architectures.

6.5.1 General considerations

6.5.1.1 Parallel computer systems

A standard classification of parallel computer systems into four types is due to Flynn, see [33,78] for details. The two types most often encountered in practice are SIMD (Single Instruction, Multiple Data), and MIMD (Multiple Instruction, Multiple Data). In a SIMD computer all processor elements simultaneously execute the same operation on different data items, whereas in a MIMD machine the processors may execute different operations on their own data. MIMD computers are more flexible, but are in general more difficult to program. Both SIMD and MIMD computers can be either of the shared memory or distributed memory type. In a *shared memory* parallel computer, there are a number of processors and a single (large) memory which is accessible to all processors. In contrast, in a *distributed memory* architecture, each processor has its own local memory and a processor can retrieve data in the memory of another processor by messages over a communication network.

The performance of a parallel computer is very much dependent on the bandwidth of the connection of the processors to the memory, that is, the maximum number of simultaneous load or store operations per time unit. Shared memory systems typically have a bandwidth problem since there is only a single memory, so that conflicts may arise when many processors try to access the same memory locations. On the other hand, distributed memory MIMD machines have the disadvantage that the communication between processors is much slower than for shared memory machines, so that the synchronization overhead is much higher when tasks have to communicate. This mismatch between communication vs. computational speed often makes communication the speed-limiting factor on distributed memory MIMD architectures, while memory congestion is usually the speed-limiting factor on shared memory systems. The maximum amount of work a process can perform before communication with other processors becomes necessary is called the *granularity* or *grain size*. *Load balancing*, i.e. ensuring equal work load of different processors during program execution, is an important requirement of parallel program design. In this context, an important issue is that of *mapping*, i.e. the assignment of tasks to processors. This may be done *statically* at initialization, or *dynamically* during execution of the program.

6.5.1.2 Parallel programming models

Various parallel programming models exist. In *message-passing programming*, tasks are created, which interact by sending and receiving messages. The approach most often used is called SPMD (single program multiple data), meaning that every processor runs the same program, performing operations on its own data space. In the *shared-memory programming* model, tasks share a common address space. Mechanisms such as locks and semaphores [28] may be used to control access to the shared memory. Below we will compare implementations of the watershed transform on distributed memory machines making use of message passing, and on shared memory architectures where synchronization takes place through shared variables.

6.5.1.3 Classification of parallel watershed algorithms

The following classification of current parallel implementations of the watershed transform can be made:

- *domain decomposition*: distribute the image over the processors in a regular way (static mapping) and use a sequential algorithm for the subimage in each subdomain. Insert synchronization and communication points where the result depends on neighbouring subdomains. Merge subresults to obtain the final solution.

- *functional decomposition*: when simulating flooding from local minima, distribute the local minima over the processors. In this case, the efficiency depends crucially on the number of local minima, and the sizes of the corresponding basins. Load imbalance may arise when the sizes of basins differ significantly.

6.5.2 Watershed implementation on distributed memory architectures

In the case of the watershed algorithm, usually domain decomposition is used on distributed memory architectures. Granularity depends on the distribution of data among processors, the number of processors and the image content. When many subimages are used, the grains are small with a relatively large number of pixels on the boundary between subdomains, requiring more communication.

In all algorithms discussed in this subsection, the image is distributed in stripes or blocks $D_i, i = 1, \ldots, N$ over all processors. Each processor has access to an overlap region between domains, determined by the neighbourhood $N_G(p)$ of each boundary pixel. By $D_i^+ = \cup_{p \in D_i} N_G(p)$ is denoted the extension of subdomain D_i, and by $D_i^- = D_i \cap (\cup_{j \neq i} D_j^+)$ the pixels of D_i which have outside neighbours. Pixels in boundary regions are written only by the process to which the subdomain is assigned, but is available for reading by processors of neighbouring subdomains. An approach where a division of the image in rectangular blocks is used naturally leads to an implementation where the processors are connected in a rectangular mesh topology, which for example is easily realizable by a transputer system (each transputer having four communication links).

Speedups are usually measured excluding the time needed for image loading, distribution, retrieval and saving.

6.5.2.1 Hill climbing by ordered queues

Parallellization of the watershed transform by ordered queues is discussed in [62]. The algorithm does not construct watershed lines. The program uses the SPMD approach with synchronization by messages from and to a master process. The image is distributed in blocks. The steps in the watershed computation are:

1. *Minima detection*: plateaus are examined by breadth-first scans in each subimage using a FIFO queue. If a plateau is spread over different subdomains, communication between processors is necessary during which merging of parts in different subdomains takes place. This may require repeated communication until stabilization (i.e. no more changes occur).

2. *Flooding by local OQ's*: each processor performs flooding in its own subdomain based on ordered queues, as in the sequential algorithm. To allow flooding to propagate to neighbouring subdomains, two approaches have been considered. In the first one [67] processors are tightly synchronized at each grey level by analyzing border pixels of subdomains whose steepest lower neighbours (or, when these do not exist, neighbours of the same grey value) are in the extension area of the subdomains. When a processor reaches synchronization level h, labels and values in their extension areas are exchanged with neighbouring processors. Communication and reflooding takes place until the label propagation stabilizes, as detected by the master process. Due to this tight synchronization considerable idle times are introduced, since processors do not execute the same code at approximately the same time. A second approach [62] first performs local flooding at all grey levels in the subdomain, followed by communication and reflooding until the label propagation stabilizes. This reduces the amount of communication necessary for reflooding.

Performance measurements. Speedups for both schemes are reported in [62, Ch. 2]. The tight synchronization scheme was implemented on a Parsytec Supercluster 128, which is a massively parallel reconfigurable network of transputers, under PIPS (Parallel Image Processing System) [74]. (An initial implementation on a loosely coupled cluster of workstations using the PVM (Parallel Virtual Machine) package [77] resulted in marginal speedups with efficiency deteriorating quickly as the number of processes was increased [67]). The second scheme was implemented on a Cray T3D MIMD distributed memory architecture with 256 nodes using MPI (Message Passing Interface) [34]. The experimental results show a moderate increase of speedup with number of processors for some images, the speedup for the second scheme being almost twice as high as that of the first scheme. (Note however, that these schemes were implemented on different architectures.) For natural images, efficiency ranges from $25-50\%$ at 16 processors, to 10% or less at 128 processors. However, both stages of the algorithm (minima detection and flooding) are very data dependent, leading to load imbalance. For artificial images with large or snake-like plateaus spread over different subdomains, speedup may be marginal or even decrease

with number of processors, due to extensive relabelling. Also, better performance is not always obtained for larger images.

6.5.2.2 Hill climbing and rainfalling after lower completion

Hill climbing, with lower completion as preprocessing, was considered by Moga et al. [62, 63], effectively using, but not explicitly introducing, the local condition of Definition 11. In addition, the *rainfalling* algorithm was studied, see Section 6.4.2.4. For both algorithms, the steps are: (i) minima detection, (ii) lower completion, (iii) flooding (by hill climbing and rainfalling, respectively).

Minima detection with lower completion on non-minima plateaus again requires repeated communication until stabilization to achieve global consistency. The flooding step is considered as labelling each vertex in the lower complete graph by the label of the minimum to which it is connected by a path. The procedure of choosing arbitrarily one of the steepest lower neighbours of a given pixel, in case several exist, turns the DAG into a disjoint-set forest. This reduces the amount of non-locality, but introduces scanning order dependence (cf. Section 6.4.2.4).

For the rainfalling algorithm, the forest is labelled inside subdomains as described above, using a FIFO queue to store root pixels of not yet resolved paths. Processors perform communication with neighbours as long as there are unresolved paths in their subdomain. But, since a processor can decide locally when to terminate its calculation, no global reduction operation is necessary; also no relabelling or synchronization between paths are needed. In the case of hill climbing, each processor initializes by a raster scan a FIFO queue with border pixels of minima in its subdomain. A pixel p removed from the queue propagates its label to all pixels q for which there is an arc from q to p in the lower complete graph. Labels are repeatedly exchanged with neighbouring processors through the extension area, initiating new labelling. A processor becomes inactive as soon as all pixels in its subdomain have been labelled. Summarizing, plateaus are treated in breadth-first order, while labelling is along paths generated by depth-first search, c.q. breadth-first search, for rainfalling, and hillclimbing, respectively.

Performance measurements. Implementations were carried out on a Parsytec Supercluster 128 under PIPS [74] and on Cray T3D under MPI. Speedup curves are rather similar for rainfalling and hillclimbing, with rainfalling having shorter running times but lower speedup. Efficiency decreases with increasing number of processors, and is very data dependent in the case of artificial scenes ($E(128) \leq 25\%$ on the Parsytec system, $E(128) \leq 12.5\%$ on the Cray T3D). Compared to the implementation using ordered queues (cf. Section 6.5.2.1) the time spent for flooding has been reduced, but the time of the first stages has increased due to the lower distance computation. Overall execution time has not improved significantly. An advantage may be that ordinary queues are easier to implement correctly than ordered queues.

6.5.2.3 Hill climbing by ordered queues combined with a connected component operator

Parallelization of the hill climbing algorithm combined with a connected component operator has been considered by Bieniek et al. [15] using the local condition of Definition 11, and by Moga et

al. [62, 65]. We first describe the former approach [15].

The main idea is to solve the watershed problem independently on all subdomains without synchronization. Instead temporary labels are assigned to pixels which will be flooded from adjacent subdomains. The boundary connectivity information is stored in a graph or equivalence table. Global labels are computed by a reduction operation using the resolving step as in the UNION-FIND algorithm (cf. Section 6.4.2.1). If N is the number of processors, computation of the global labels then takes $\log_2 N$ steps, independent of the complexity of the data. The latter problem is strongly related to the connected component labelling problem [2, 31, 85].

The algorithm for images without plateaus is as follows:

1. Give all local minima in each domain D_i a globally unique label, using information from D_i^+.

2. Give all pixels p of D_i^- a temporary label (globally unique) if the downstream neighbours of p are in another subdomain. The set of boundary pixels requiring a temporary label is thus
$$D_i^{temp} = \{p \in D_i | \Gamma(p) \cap (D_i^+ \setminus D_i) \neq \emptyset\}. \tag{6.11}$$

3. Produce a watershed segmentation consistent with Definition 11, independently on each subdomain, using the minima and temporary labels as seeds for basins. By using ordered queues, non-minima plateaus which are completely within a subdomain will be flooded in accordance with Definition 11.

4. Merge subdomains pairwise: give all labels of the subdomains globally consistent values by linking basins, which have grown from pixels p with a temporary label, to basins in the downstream of p.

An efficient implementation of step 4 in this algorithm can be based upon the UNION-FIND algorithm, as discussed in Section 6.4.2.1.

As an example, consider Fig. 6.40. Figure 6.40(b) shows a watershed segmentation of the image in Fig. 6.40(a) consistent with Definition 11 (several other labellings are possible). Next consider a subdivision of the image into two strips of three rows each. In the figure, we show the result after step 2 and step 3 of the above algorithm. Clearly, after step 4 (not shown) a correct result (w.r.t. Definition 11) is obtained.

Of course, the real problem is to treat images with plateaus. To do this, the following procedure is proposed in [15]. Define the neighbour set $\Gamma'(p)$ of a plateau pixel as the union of all neighbour sets $\Gamma(p')$ with p' running over those boundary pixels of the plateau which have minimal geodesic distance to p. Extend Definition 11 by replacing $\Gamma(p)$ by $\Gamma'(p)$. Then the claim is that with a similar replacement of $\Gamma(p)$ by $\Gamma'(p)$ in the algorithm above, a watershed segmentation is produced in agreement with the extended definition. It is easy to see, however, that this claim cannot be true. When in (6.11), $\Gamma(p)$ is replaced by $\Gamma'(p)$, the set D_i^{temp} as defined in (6.11) may be empty, because the set $\Gamma'(p)$ may be located in a subdomain which is far away from D_i, and therefore has zero overlap with $D_i^+ \setminus D_i$. Therefore, the watershed segmentation according to the algorithm above produces a result in a subdomain D_i which is completely independent of the

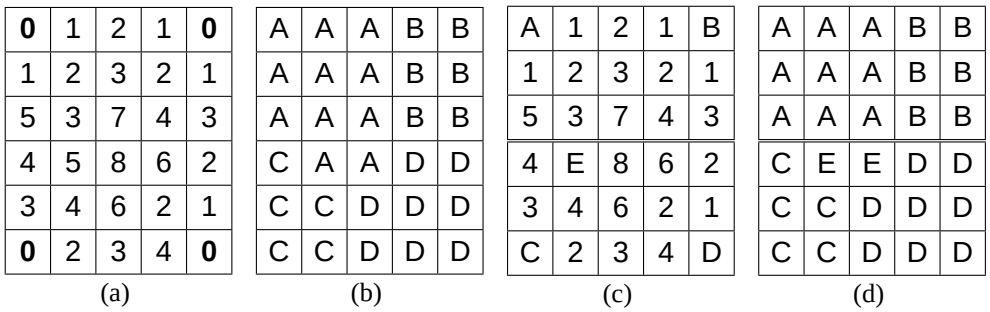


Figure 6.40. *Watershed according to Definition 11. (a): original image; (b): a watershed segmentation of the complete image; (c): result after step 2 of the parallel algorithm with two subdomains. (d): result after step 3 of the parallel algorithm with two subdomains.*

downstream. In fact, what should be done is to assign temporary labels to all pixels p for which $\Gamma'(p)$ extends to any other subdomain, whether it is an adjacent one or not. But in that case, the locality aimed at by introducing Definition 11 is lost. This effectively annihilates the idea of computing the watershed transform independently on each subdomain, followed by a merging step requiring communication of boundary information only. It comes therefore as no surprise that in the implementation in [15], which uses a variant of the sequential watershed transform (8-connected) based on ordered queues, an iterative plateau correction is required after step 3 of the algorithm for labels and distances stretched out over more than one subdomain, needing a global synchronization step. This clearly demonstrates that the locality assumption, which is implicit in the specification of the algorithm with plateaus, does not hold.

A similar approach was used by Moga et al. [62,65]. The difference with the approach in [15] is that for non-minima plateaus which are shared by several processors the globally correct lower distance values are computed *before* flooding, instead of during flooding, so that no relabelling of wrongly labelled higher neighbourhoods of plateaus is necessary.

Performance measurements. Experiments have been reported in [15] on a Parsytec Supercluster 128 under PIPS and in [65] for an implementation under MPI on a Cray T3D, both with similar results. An almost linear speedup is obtained for a number of processors up to 64, after which saturation sets in. Efficiency at 128 processors ranges from 10-50 %. Local minima detection and local flooding takes most of the time, until the number of processors becomes very large: then the plateau correction is the most time limiting factor, even for images with small plateaus. This implementation performs better with respect to scalability and execution time than the implementation of hill climbing or rainfalling discussed in Section 6.5.2.2. Also, the data dependence is less severe, although it is still present due to the plateau correction step.

6.5.2.4 Parallel watershed transform based on sequential scanning

Finally we mention a watershed algorithm based on sequential scannings [62, 68] (see Section 6.4.2). Although the sequential algorithm is very slow, the method has good scaling prop-

erties in a parallel implementation. The implementation consists of repeated raster and anti-raster scans within the subdomains and message passing among processors until stabilization. Although the implementation is carried out on a MIMD architecture (Parsytec cluster), the algorithm is also suitable for SIMD computers since no queues are used.

The parallel implementation has the following steps: (i) detection of minima; (ii) lower completion of the image; (iii) labelling minima; (iv) flooding by image integration. Labels in the boundaries between subimages have to be communicated between processors. Even if the computation has stabilized within a subdomain, new raster scans may be necessary due to changes in the boundary region caused by other processors. A master process detects when global stabilization has been obtained. Since all subdomains are equal in size, and each processor executes simple operations in raster scan mode, chances are higher that communication points are reached at approximately the same time, resulting in a better load balance.

Performance measurements. Implementations were carried out under PIPS on a Parsytec Supercluster 128 [68], and under MPI on a Cray T3D [62, Ch. 3]. An approximately linear speedup, close to the ideal line, is obtained in many cases, but efficiency drops when the number of processors becomes larger. Speedup may even decrease for small image size, because the amount of work per processor becomes too small. Larger image sizes yield higher speedups for the same number of processors. The algorithm is still data dependent: for images with large, snake-like plateaus, speedup may drop with increasing number of processors.

6.5.2.5 Conclusions

Summarizing the performance results for the various parallel implementations discussed in Section 6.5.2.1-Section 6.5.2.4, it was found that slow methods such as sequential scanning have the best scaling properties. Ordered queues are relatively fast, but have the worst scaling, since in a distributed memory system local ordered queues need to be kept in global order by proper synchronization. The improvement of hill climbing and rainfalling over ordered queues is marginal. Better results have been obtained by combining the ordered queue implementation with a connected component operator. The data dependence is less severe, since the use of the connected component operator allows the computation of globally correct labels with fixed time complexity, independent of the image content.

All algorithms have problems, although to varying degree, with images containing large or snake-like plateaus spread over different subdomains: speedup may be marginal or even decrease with increasing number of processors.

6.5.3 Shared memory implementations

The main motivation to use shared memory architectures for the watershed transform derives from the global data dependence caused by extended basins in grey value images. This requires direct access to data which may be far separated in memory. For that reason, a shared memory is an obvious choice, cf. Section 6.5.1.

6.5.3.1 Components graph algorithm

A parallelization of the components graph algorithm (cf. Section 6.4.1.2) is proposed in [51]. All pixels which are in the same level component are clustered in one single node of the components graph, therefore plateaus no longer exist. Since shared memory is used, the parallel programs are very similar to the sequential counterparts. Only concurrent references to the same memory locations have to be protected by synchronization primitives.

The steps are the following.

1. *Level components labelling.* A single processor labels level components of the entire image and distributes the input image and the labelled image over the processors. To each processor a slice of consecutive scan lines of (approximately) equal size is assigned, with one scan line overlap so that it can be decided whether level components are shared with neighbouring processors.

2. *Parallel watershed transform of a graph.* Every processor builds a local components graph for its own image slice. Since some level components are shared between several processors the graphs on the processors are not disjoint. Next every processor performs an adapted version of the flooding algorithm, taking care of shared vertices.

3. *Back transformation*. After flooding each processor transforms its local components graph back to an image slice, as in the sequential case.

6.5.3.2 Topographical distance algorithm by ordered queue

The first step of the algorithm is detecting local minima, see Section 6.4.2.1. Computation of lower slope and cost function are local operations, and therefore easy parallelizable. Actually, minima detection and lower completion can be obtained in one step (see Section 6.3.2.2). Computation of the watershed transform on the graph is based on Algorithm 6.4.2, which does compute watershed pixels. Each processor computes the basins of an (approximately) equal number of minima. However, there is a strong data dependence since the sizes of basins may differ substantially. Therefore dynamic instead of static mapping is needed to obtain good performance.

Results. Timing results for the test images of Fig. 6.35 using static mapping on a Cray-J932 shared memory computer with 16 processors are given in Table 6.2, which is taken from [52]. The column T_1 is the time (in seconds) for the computation of 100 watersheds on a single processor. In the column S_p the speedup is given if we use p processors.

We see that the speedup in the case of the *blobs* image remains the same if we keep adding more processors. The image contains only 4 regional minima, and thus each extra processor will remain idle. The poor speedup in the case of the *chess board* image is caused by the fact that it contains a widespread regional minimum – the boundaries of the squares. This minimum reaches over the entire image, causing a big load imbalance. The *peppers* image and the *aircraft* image contain many regional minima, most of them are noise resulting in many very small tasks

Table 6.2. *Timings and speedups for the 6 test images of Fig. 6.35.*

image	#minima	T_1	S_2	S_4	S_8	S_{16}
blobs	4	88	1.7	2.5	3.0	3.0
chess	67	101	1.6	2.3	3.6	4.0
waves	20	115	1.7	2.4	3.6	8.5
peppers	44426	111	1.7	2.1	3.0	5.0
gold	359	115	1.7	2.5	3.7	10.4
aircraft	19053	114	1.6	2.1	2.9	4.8

causing a lot of overhead. The *waves* image and the *gold* image contain a reasonable number of uniformly distributed regional minima, resulting in a fairly good speedup.

The speedup for computing lower slope and cost function is almost linear in the number N of processors. The same holds for minima detection, although the influence of concurrent references to the same memory locations starts to play a major role if we use many processors, typically 8 or more. If the number of minima is smaller than N, no speed is gained by using more processors. In practice, however, the number of minima is usually much larger than N.

6.5.3.3 Topographical distance algorithm by modified UNION-FIND

This approach to compute the watershed transform is relatively easy to parallelize for shared memory architectures. In a first phase the input image is transformed into a lower complete image using FIFO queues. The very same algorithm can be used on parallel architectures by splitting the domain of the image in (almost) equally sized subdomains. Each processor has its own private FIFO queue, which it initializes with seed points (pixels that are at the lower boundary of a plateau) in its private subdomain. After this initialization, each processor starts propagating distances in a private image *dist*. When all processors have finished this operation, the minimum over all the *dist* images is the desired lower complete image. Computing this minimum can be efficiently performed in parallel using a so-called *reduction operator* on most parallel architectures. After computing the lower complete image, the DAG *sln* can be constructed in a single image pass, in which for each pixel only its neighbours are addressed. There is no dependence between pixels, as far as the computation order is concerned, and thus this operation can trivially be performed in parallel. The resolving phase is also easy to parallelize, by replacing the domain D in Algorithm 6.4.8 by the private domain of a processor. Note that the mapping procedure is hybrid in this case: initially, a processor starts to work on pixels in its private domain, but during the resolving phase it will access pixels in domains of other processors. Work by the present authors on this approach is in progress.

6.6 Summary

We have reviewed various existing definitions of the watershed transform based on immersion or on shortest paths with respect to a topographical distance function. The main sequential algorithms for computing the watershed transform according to both definitions were described. Emphasis was put on the fact that watershed algorithms found in the literature often do not adhere to their definition, or are the implementation of an algorithm without a proper specification.

Strategies for parallel implementation were discussed, distinguishing between distributed memory and shared memory architectures. The watershed algorithm by immersion is hard to parallelize because of its inherently sequential nature. A parallel implementation of this algorithm can be based upon a transformation to a components graph. The distance-based definition allows various parallel implementations. The main ones are based on (ordered) queues, repeated raster scanning, a modified UNION-FIND algorithm, or a combination of these. The main conclusion to be drawn from this review is that, despite all the techniques and architectures used, there is always a stage in the watershed transform which remains a global operation, and therefore in the case of parallel implementation at most modest speedups are to be expected.

Chapter 7

Concluding Remarks

7.1 Summary

This thesis deals with efficient algorithms for morphological image processing. Each chapter deals with some algorithm from the field of mathematical morphology. Reviews of existing algorithms, as well as new algorithms or improvements to existing algorithms are given. In most chapters, the focus is on the design and implementation of parallel algorithms. The interest for parallel algorithms is mainly driven by the growing demand of real-time 2D image processing applications, and algorithms for the interactive processing of 3D volume data sets. Parallel algorithms for the processing of 3D volume data sets are especially interesting in the field of medical imaging, where the processing of large MRI and CT-data sets has become daily practice.

In chapter 2 a general algorithm for computing the *distance transform* of a binary image in linear time is presented. A distance transform computes a grey scale image of which each pixel is assigned its distance to the nearest foreground pixel in the binary image. The presented algorithm is general in the sense that it can be used for different metrics. The metrics discussed in chapter 2 are the Manhattan or City-block metric (L_1-metric), the chessboard metric (L_∞-metric), and the Euclidean metric (L_2-metric).

The algorithm can be summarized as follows. In a first phase each column C_x (defined by points (x,y) with x fixed) is separately scanned. For each point (x,y) on C_x, the distance $G(x,y)$ of (x,y) to the nearest foreground pixel in the same column of the binary image is determined. In a second phase each row R_y (defined by points (x,y) with y fixed) is separately scanned, and for each point (x,y) on R_y the minimum of $(x-x')^2+G(x',y)^2$ for the L_2-norm, $|x-x'|+G(x',y)$ for the L_1-norm, and $|x-x'|$ **max** $G(x',y)$ for the L_∞-norm is determined, where (x',y) ranges over row R_y. Both phases consist of two scans, a forward and a backward scan. The time complexity of the algorithm is linear in the number of pixels. Since the computation per row (column) is independent of the computation of other rows (columns), the algorithm is easily parallelized for concurrent execution on shared memory computers. A barrier is used as a synchronization point between the two phases. The algorithm turns out to be very efficient, and scales almost perfectly on shared memory architectures.

The algorithm is easily extended to d-dimensional ($d > 2$) distance transforms by separating the algorithm into d phases, each solving a one-dimensional problem.

In chapter 3 the design is described of a parallel algorithm for the determination of *connected*

components in binary and grey-scale images. Points of an image are regarded connected if they are neighbors and have the same grey-value. A set of points C is a connected component if for each point in C a connected path of points in C exists to any other point in the set, and no more points can be added to C while maintaining this property.

A concurrent version of Tarjan's Union-Find algorithm for determining equivalence classes in a graph is used. Although we are interested in the application to images, we present the algorithm for undirected graphs. The design goes through four stages. We first give a version of Tarjan's sequential algorithm. Distribution of the vertices of the graph over a number of processes leads to a message passing algorithm. This design is then mapped to a shared memory architecture by means of mutual exclusion and synchronization. Finally, the mutual exclusion and synchronization are implemented by means of POSIX thread primitives. The resulting algorithm is a concurrent one in which the amount of communication is decided at runtime. As a consequence, processes must not stop when they finished some task, since other processes may send a new one. This yields a termination detection problem, which is solved by counting the number of tasks that still need to be done.

We finally describe the application to the determination of connected components in images. Graphs have no dimension, so the algorithm is applicable to images of any size and dimension. Since images are usually more or less constant locally, we sketch an optimization that can significantly reduce the number of communications needed. Experiments show that distribution is quite effective, and we observe a speedup that is often almost linear in the number of processors.

In chapter 4 a review is given of algorithms for *connected set openings and closings*, and in particular *attribute openings and closings*. The two best known algorithms for these image filters are the pixel priority-queue method introduced by Vincent for area operators [99] and extended by Breen and Jones to attribute operators [19], and the Max-tree method of Salembier and coworkers [83] which uses hierarchical queues. These two methods are compared to an algorithm developed by Meijster and Wilkinson [57, 58], which is based on the union-find algorithm [88]. The focus is on the area opening since this is one of the simplest cases.

An area opening lowers the grey-value of each pixel that belongs to a connected component that has an area smaller than some threshold λ until the component has an area at least λ. In the binary case, area closings fill all *background* components with an area smaller than λ. The area closing is the dual of the area opening. An area opening (closing) can either remove image components completely, or leave them intact, but never alter their shape.

For the determination of connected components, the algorithm of Vincent is based on a region growing (flooding) algorithm that uses a priority queue. Each regional extremum in the image is processed sequentially, and is expanded with neighboring pixels, until it either satisfies the area criterion, or a pixel with higher grey-value is encountered. In the latter case, the processing of the component is stopped, since it will be re-flooded later during the processing of some other regional extremum. If N is the number of pixels in the image, the computational complexity of the algorithm is $O(N\lambda \log \lambda)$, which becomes $O(N^2 \log N)$ if $\lambda = N$. The latter is the case when computing *pattern spectra* (see section 4.7). Experiments show that the running time of the algorithm depends strongly on the parameter λ, but also on the number of regional extrema.

The algorithm of Salembier of consists of two phases. First a tree, called a *Max-tree*, is constructed from the image, after which the tree can be used for filtering. A Max-Tree is a tree

where the nodes represent the connected components of the image thresholded at various levels. The tree is a rooted tree, of which the regional extrema are the leaves. Each node C in the tree has a parent pointer to another node, which represents the connected component C is contained in when the data set is thresholded at a lower level. Salembier uses a recursive flooding algorithm, to construct the tree. He uses a *hierarchical queue* for the flooding process, which is basically a queue of queues. This queue structure is used for flooding and for processing neighboring components of a regional extremum in order of grey-value. In the nodes of the tree attributes, like area, can be stored. Filtering the image boils down to following parent pointers in the tree, until a node is reached where the filter criterion is satisfied. The computational cost of the Max-tree algorithm is dominated by the flood filling process which is inherently linear in both the number of pixels and in the connectivity. Filtering the Max-tree requires visiting all nodes in the tree, so it is also $O(N)$. Computing the output image is also linear in the number of pixels. One somewhat peculiar feature of the computational complexity of the Max-tree algorithm is its dependency on the number of unoccupied grey levels between parent and child nodes, since a linear search is necessary to find the next occupied grey level.

The key difference between the algorithm based on union-find and the methods of Vincent and Salembier et al. is that multiple extrema are processed simultaneously, rather than sequentially. The algorithm is based on the idea to compute connected components using union-find (see chap. 3), while using Salembier's idea of constructing a tree. The tree is replaced by a forest of smaller trees. Each tree is extended until the root of the tree satisfies the filter criterion.

The union-find algorithm outperforms the other methods on almost all natural and synthetic images tested. The running time of the Max-tree algorithm is independent of λ, while the union-find algorithm shows a very weak dependence. Still, from practical experiments with 2D images, the Max-tree algorithm is typically slower by a factor of 2. The advantage of the union-find algorithm is even greater for the processing of 3-D data sets. This is probably due to its smaller memory requirements, especially given the limited bandwidth of the memory bus of personal computers. In the case of a $128 \times 128 \times 62$ volume, the total memory use of the Max-tree method was 57.5 MB, the priority-queue method needed 45 MB, whereas the union-find algorithm required only 25 MB. Chapter 4 briefly discusses the extension to *attribute openings* and *pattern spectra*. It has been demonstrated that it is possible to compute an area pattern spectrum in the time needed for a single area opening.

Although the attribute opening (closing) of Meijster and Wilkinson is faster in the case in which one wants to filter a data set with a given parameter λ, the Max-tree algorithm does have its merits. It is very useful in the case one wants to filter a data set interactively, by first computing the tree structure, and supplying the user with some graphical user interface to choose the parameter λ. Filtering the tree is a fast operation, while constructing the tree can be done off-line or as a pre-processing step. In view of this application, a parallel algorithm for constructing and filtering Max-trees is discussed in chapter 5.

In chapter 6 an extensive review is given of various existing definitions of the *watershed transform* based on *immersion* or on *shortest paths* with respect to a *topographical distance function*. The watershed transform can be classified as a region-based segmentation approach. The idea is to look upon a grey-scale image as a landscape, where grey levels denote altitude. An intuitive description of the watershed transform goes as follows. Imagine the landscape being

immersed in a lake, with holes pierced in local minima. Basins (also called '*catchment basins*') will fill up with water starting at these local minima, and, at points where water coming from different basins would meet, dams are built. When the water level has reached the highest peak in the landscape, the process is stopped. As a result, the landscape is partitioned into regions or basins separated by dams, called *watershed lines* or simply *watersheds*.

The main sequential algorithms for computing the watershed transform according to both definitions are described. Emphasis is put on the fact that watershed algorithms found in the literature often do not adhere to their definition, or are the implementation of an algorithm without a proper specification.

Strategies for parallel implementation are discussed, distinguishing between distributed memory and shared memory architectures. The watershed algorithm by immersion is hard to parallelize because of its inherently sequential nature. A parallel implementation of this algorithm can be based upon a transformation to a components graph. The distance-based definition allows various parallel implementations. The main ones are based on (ordered) queues, repeated raster scanning, a modified UNION-FIND algorithm, or a combination of these. The main conclusion to be drawn from this review is that, despite all the techniques and architectures used, there is always a stage in the watershed transform which remains a global operation, and therefore in the case of parallel implementation at most modest speedups are to be expected.

7.2 Perspectives

This thesis has presented and reviewed only a small selection of algorithms for commonly used morphological operators. Clearly, there are many algorithms for morphological operators that were not discussed. Some of them are closely related with the ones discussed in this thesis. For example, it would be interesting to investigate if it is possible to extend the distance transform algorithm (chapter 2) for binary images into one that can accept grey-scale images as its input. The idea is to compute for each pixel the distance to the nearest pixel with a lower grey-value, i.e. the lower distance as discussed in chapter 6. Apart from its own merit, such an algorithm would be beneficial when we want to compute the watershed according to Meyer's definition.

The connected components algorithm presented in chapter 3 could possibly be improved by using a computation scheme that resembles the distance transform algorithm. In the case of 2D images, it is possible to scan each row of the image independently, and construct small disjoint set trees per scan-line. These small trees are clearly completely compressed, i.e. each pixel points directly to its parent. Obviously, this processing phase takes linear time, and is trivially parallelized. In a second phase, neighboring scan-lines are merged, using the information coded in the trees obtained from the previous scan. These trees represent a partition of a scan-line in segments, in which the grey-value is constant. We can use this information to skip vertical edges between pixels p and q which both lie in the interior of such a segment. In a way, we could regard this technique as connected component labeling on run-length encoded images. In the case of 2D images this could be worthwhile, especially in images with large (background) components. It is not clear how to extend this idea to 3D data sets.

The generality of the attribute opening (closing) algorithms of chapters 4 and 5 yields a

perfect and general setup to search for new attributes that can be used to segment and filter large data sets. At this moment we are busy to incorporate the algorithm in a radiotherapy application, that supplies the user with a 2D and 3D view on 3D CT and MRI data sets.

It would be interesting to investigate whether it is possible to use the Max-tree approach for pre-computing a number of texture images for faster interactive visualization for certain ranges of the filter parameter λ. It might also be a good idea to compute textures predictively based on the current value of the filter parameter chosen by the user.

As far as the watershed is concerned, we only discussed algorithms for computing the watershed of an image directly. This is hardly ever useful, since this yields a severe over-segmentation. It would be interesting to investigate if it is possible to construct a concurrent algorithm with which a user can interactively select markers (user selected local minima), such that this over-segmentation is reduced.

Finally, it would be a good idea to incorporate the algorithms presented in this thesis in well known image processing and visualization libraries.

Bibliography

[1] Aho, A., Hopcroft, J., and Ullman, J. *Data structures and algorithms*. Addison Wesley, Reading, MA, 1985.

[2] Alnuweiri, H. M., and Prasanna, V. K. Parallel architectures and algorithms for image component labeling. *IEEE Trans. Patt. Anal. Mach. Intell. 14*, 10 (1992), 1014–1034.

[3] Andrews, G. *Concurrent Programming, principles and practice*. Addison Wesley, Reading, MA, 1991.

[4] Andrews, G., and Olsson, R. *The SR Programming Language, concurrency in practice*. Benjamin/Cummings, 1993.

[5] Apt, K., and Olderog, E.-R. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, New York–Heidelberg–Berlin, 1991.

[6] Bacon, J. *Concurrent systems*. Addison Wesley Longman Ltd, 1998.

[7] Bangham, J. A., Chardaire, P., Pye, C. J., and Ling, P. D. Multiscale nonlinear decomposition: the sieve decomposition theorem. *IEEE Trans. Pattern Anal. Mach. Intell. 18* (1996), 529–538.

[8] Bangham, J. A., Harvey, R., Ling, P. D., and Aldridge, R. V. Morphological scale-space preserving transforms in many dimensions. *J. Electr. Imag. 5* (1996), 283–299.

[9] Bangham, J. A., Harvey, R., Ling, P. D., and Aldridge, R. V. Nonlinear scale-space from n-dimensional sieves. In *Proceedings IEEE ECCV'96* (1996), vol. 1064 of *Lecture Notes in Computer Science*, pp. 189–198.

[10] Bangham, J. A., Ling, P. D., and Harvey, R. Scale-space from nonlinear filters. *IEEE Trans. Pattern Anal. Mach. Intell. 18* (1996), 520–528.

[11] Berge, C. *Théorie des Graphes et ses Applications*. Dunod, Paris, 1958.

[12] Beucher, S. Watershed, hierarchical segmentation and waterfall algorithm. In *Mathematical Morphology and its Applications to Image Processing*, J. Serra and P. Soille, Eds. Kluwer Acad. Publ., Dordrecht, 1994, pp. 69–76.

[13] Beucher, S., and Lantuéjoul, C. Use of watersheds in contour detection. In *Proc. International Workshop on Image Processing, Real-Time Edge and Motion Detection/Estimation, Rennes, september* (1979).

[14] Beucher, S., and Meyer, F. The morphological approach to segmentation: the watershed transformation. In *Mathematical Morphology in Image Processing*, E. R. Dougherty, Ed. Marcel Dekker, New York, 1993, ch. 12, pp. 433–481.

[15] Bieniek, A., Burkhardt, H., Marschner, H., Nölle, M., and Schreiber, G. A parallel watershed algorithm. In *Proc. 10th Scandinavian Conference on Image Analysis (SCIA'97), Lappeenranta, Finland* (1997), pp. 237–244.

[16] Bieniek, A., and Moga, A. A connected component approach to the watershed segmentation. In *Mathematical Morphology and its Applications to Image and Signal Processing*, H. J. A. M. Heijmans and J. B. T. M. Roerdink, Eds. Kluwer Acad. Publ., Dordrecht, 1998, pp. 215–222.

[17] Borgefors, G. Distance transformations in arbitrary dimensions. *Computer Vision, Graphics, and Image Processing 27* (1984), 321–345.

[18] Borgefors, G. Distance transformations in digital images. *Computer Vision, Graphics, and Image Processing 34* (1986), 344–371.

[19] Breen, E. J., and Jones, R. Attribute openings, thinnings and granulometries. *Comp. Vis. Image Understand. 64*, 3 (1996), 377–389.

[20] Butenhof, D. *Programming with POSIX Threads*. Addison-Wesley, Reading, Massachusetts, USA, 1997.

[21] Cheng, F., and Venetsanopoulos, A. N. An adaptive morphological filter for image processing. *IEEE Trans. Image Proc. 1* (1992), 533–539.

[22] Copty, N., Ranka, S., Fox, G., and Shankar, R. A data parallel algorithm for solving the region growing problem on the connection machine. *Journal of Parallel and Distributed Computing 21* (1994), 160–168.

[23] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms*. MIT Press, 1990.

[24] Crespo, J., Schafer, R. W., Serra, J., Gratin, C., and Meyer, F. The flat zone approach: a general low-level region merging segmentation method. *Signal Processing 62* (1997), 37–60.

[25] Danielsson, P. Euclidean distance mapping. *Comput. Graphics Image Process. 14* (1980), 227–248.

[26] Digabel, H., and Lantuéjoul, C. Iterative algorithms. In *Actes du Second Symposium Européen d'Analyse Quantitative des Microstructures en Sciences des Matériaux, Biologie et Médecine, Caen, 4-7 October 1977* (1978), J.-L. Chermant, Ed., Riederer Verlag, Stuttgart, pp. 85–99.

[27] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik 1* (1959), 269–271.

[28] Dijkstra, E. W. Co-operating sequential processes. In *Programming Languages*, F. Genuys, Ed. Academic Press, New York, 1968, pp. 43–112.

[29] Dillencourt, M., Samet, H., and Tamminen, M. A general approach to connected-component labeling for arbitrary image representations. *J. ACM 39* (1992), 253–280.

[30] Dobrin, B. P., Viero, T., and Gabbouj, M. Fast watershed algorithms: analysis and extensions. In *SPIE 1994; Vol. 2180. Proc. IS&T/SPIE Symposium on Electronic Imaging Science & Technology, Nonlinear Image Processing V, February 6-10, 1994, San Jose Convention Center, CA.* (1994), pp. 209–220.

[31] Embrechts, H., Roose, D., and Wambacq, P. Component labelling on a mimd multiprocessor. *Comp. Vision Graph. Image Proc. 75*, 2 (1993), 155–165.

[32] Fiorio, C., and Gustedt, J. Two linear time union-find strategies for image processing. *Theoretical Computer Science A 154*, 2 (Feb. 1996), 165–181.

[33] Foster, I. *Designing and Building Parallel Programs*. Addison Wesley, Reading, MA, 1994.

[34] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, MA, 1995.

[35] Guillemin, V., and Pollack, A. *Differential Topology*. Prentice-Hall, Englewood Cliffs, NJ, 1974.

[36] Hambrusch, S. Parallel algorithms for gray-scale digitized picture component labelling on a mesh-connected computer. *Journal of Parallel and Distributed Computing 20* (1994), 56–68.

[37] Haralick, R. M., and Shapiro, L. G. Survey : image segmentation techniques. *Comp. Vision Graph. Image Proc. 29* (1985), 100–132.

[38] Heijmans, H. *Morphological Image Operators*. Academic Press, Boston, 1994.

[39] Heijmans, H. J. A. M. Connected morphological operators for binary images. *Comp. Vis. Image Understand. 73* (1999), 99–120.

[40] Hesselink, W. H., Meijster, A., and Bron, C. Concurrent determination of connected components. *Sci. Comp. Programming 41* (2001), 173–194.

[41] Institute of Electrical and Electronic Engineers, Inc. Information Technology — Portable Operating Systems Interface (POSIX) — Part: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]. IEEE Standard 1003.1c–1995, IEEE, New York City, New York, USA, 1995. also ISO/IEC 9945-1:1990b.

[42] Jackway, P. T., and Deriche, M. Scale-space properties of the multiscale morphological dilation-erosion. *IEEE Trans. Pattern Anal. Mach. Intell. 18* (1996), 38–51.

[43] Kleiman, S., Shah, D., and Smaalders, B. *Programming with Threads*. SunSoft Press. Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1996.

[44] Klein, J. C., Lemonnier, F., Gauthier, M., and Peyrard, R. Hardware implementation of the watershed zone algorithm based on a hierarchical queue structure. In *Proc. IEEE Workshop on Nonlinear Signal and Image processing, June 20-22, Neos Marmaras, Halkidiki, Greece* (1995), I. Pitas, Ed., pp. 859–862.

[45] Kolountzakis, M., and Kutulakos, K. Fast computation of the Euclidean distance maps for binary images. *Information Processing Letters 43* (1992), 181–184.

[46] Lantuéjoul, C. *La squelettisation et son application aux mesures topologiques des mosaïques polycristallines*. PhD thesis, Ecole des Mines, Paris, 1978.

[47] Lantuéjoul, C. Skeletonization in quantitative metallography. In *Issues of Digital Image Processing* (1980), R. Haralick and J. Simon, Eds., Sijthoff and Noordhoff.

[48] Lumia, R., Shapiro, L., and Zuniga, O. A new connected components algorithm for virtual memory computers. *Comp. Vision Graph. Image Proc. 22* (1983), 287–300.

[49] Lynch, N. *Distributed Algorithms*. Morgan Kaufman Publ., San Francisco, CA, 1996.

[50] Matheron, G. *Random Sets and Integral Geometry*. John Wiley, 1975.

[51] Meijster, A., and Roerdink, J. B. T. M. A proposal for the implementation of a parallel watershed algorithm. In *Computer Analysis of Images and Patterns*, V. Hlaváč and R. Šára, Eds., vol. 970 of *Lecture Notes in Computer Science*. Springer-Verlag, New York–Heidelberg–Berlin, 1995, pp. 790–795.

[52] Meijster, A., and Roerdink, J. B. T. M. Computation of watersheds based on parallel graph algorithms. In *Mathematical Morphology and its Applications to Image and Signal Processing*, P. Maragos, R. W. Shafer, and M. A. Butt, Eds. Kluwer Acad. Publ., Dordrecht, 1996, pp. 305–312.

[53] Meijster, A., and Roerdink, J. B. T. M. A disjoint set algorithm for the watershed transform. In *Proc. IX European Signal Processing Conference (EUSIPCO'98), September 8 - 11, 1998, Rhodes, Greece* (1998), S. Theodoridis, I. Pitas, A. Stouraitis, and N. Kalouptsidis, Eds., pp. 1665–1668.

[54] Meijster, A., Roerdink, J. B. T. M., and Hesselink, W. H. A general algorithm for computing distance transforms in linear time. In *Proc. Int. Symp. Math. Morphology (ISMM) 2000* (Palo Alto, CA, June 2000), pp. 331–340.

[55] Meijster, A., Westenberg, M. A., and Wilkinson, M. H. F. Interactive shape preserving filtering and visualization of volumetric data. In *Fourth IASTED Conf Comp. Signal Image Proc., SIP2002* (Kauai, Hawaii, USA, August 12-14 2002), pp. 640–643.

[56] Meijster, A., and Wilkinson, M. H. F. Fast computation of morphological area pattern spectra. In *Int. Conf. Image Proc. 2001* (2001), pp. 668–671.

[57] Meijster, A., and Wilkinson, M. H. F. A comparison of algorithms for connected set openings and closings. *IEEE Trans. Pattern Anal. Mach. Intell. 24*, 4 (2002), 484–494.

[58] Meijster, A., and Wilkinson, M. H. F. An efficient algorithm for morphological area operators. Tech. Rep. 99-9-07, Institute for Mathematics and Computing Science, University of Groningen, submitted.

[59] Meyer, F. Un algorithme optimal de ligne de partage des eaux. In *Proceedings 8th Congress AFCET, Lyon-Villeurbane, France* (1992), vol. 2, pp. 847–859.

[60] Meyer, F. Topographic distance and watershed lines. *Signal Processing 38* (1994), 113–125.

[61] Meyer, F., and Beucher, S. Morphological segmentation. *J. Visual Commun. and Image Repres. 1*, 1 (1990), 21–45.

[62] Moga, A. *Parallel watershed algorithms for image segmentation*. PhD thesis, Tampere University of Technology, Tampere, Finland, Feb. 1997.

[63] Moga, A. N., Cramariuc, B., and Gabbouj, M. Parallel watershed transformation algorithms for image segmentation. *Parallel Computing 24* (1998), 1981–2001.

[64] Moga, A. N., and Gabbouj, M. A parallel watershed algorithm based on the shortest path computation. In *Parallel Programming and Applications*, P. Fritzson and L. Finmo, Eds. IOS Press, 1995.

[65] Moga, A. N., and Gabbouj, M. Parallel image component labeling with watershed transformation. *IEEE Trans. Patt. Anal. Mach. Intell. 19*, 5 (May 1997), 441–450.

[66] Moga, A. N., and Gabbouj, M. Parallel marker-based image segmentation with watershed transformation. *Journal of Parallel and Distributed Computing 51*, 1 (1998), 27–45.

[67] Moga, A. N., Viero, T., Dobrin, B. P., and Gabbouj, M. Implementation of a distributed watershed algorithm. In *Mathematical Morphology and its Applications to Image Processing*, J. Serra and P. Soille, Eds. Kluwer Acad. Publ., Dordrecht, 1994, pp. 281–288.

[68] Moga, A. N., Viero, T., Gabbouj, M., Nölle, M., Schreiber, G., and Burkhardt, H. Parallel watershed algorithm based on sequential scanning. In *Proc. IEEE Workshop on Nonlinear Signal and Image processing, June 20-22, Neos Marmaras, Halkidiki, Greece* (1995), I. Pitas, Ed., pp. 991–994.

[69] Moore, E. F. The shortest path through a maze. In *Proc. Intern. Symp. on Theory of Switching, 1957* (1959), vol. 30 of *Annals of the computation laboratory of Harvard University*, pp. 285–292.

[70] *Message Passing Interface (MPI) standard*. http://www-unix.mcs.anl.gov/mpi.

[71] Nackman, L. R. Two-dimensional critical point configuration graphs. *IEEE Trans. Patt. Anal. Mach. Intell. 6*, 4 (1984), 442–450.

[72] Najman, L., and Schmitt, M. Watershed of a continuous function. *Signal Processing 38* (1994), 99–112.

[73] Noguet, D., Merle, A., and Lattard, D. A data dependent architecture based on seeded region growing strategy for advanced morphological operators. In *Mathematical Morphology and its Applications to Image and Signal Processing*, P. Maragos, R. W. Shafer, and M. A. Butt, Eds. Kluwer Acad. Publ., Dordrecht, 1996, pp. 235–243.

[74] Nölle, M., Schreiber, G., and Schulz-Mirbach, H. PIPS–a general purpose parallel image processing system. In *Proceedings 16th DAGM-Symposium Mustererkennung, Vienna* (Sept. 1994), G. Kropatsch, Ed., Reihe Informatik XPress, Springer-Verlag, New York–Heidelberg–Berlin, pp. 271–309.

[75] Pavel, S., and Akl, A. Efficient algorithms for the Euclidean distance transform. *Parallel Processing Letters 5* (1995), 205–212.

[76] Preteux, F. On a distance function approach for gray-level mathematical morphology. In *Mathematical Morphology in Image Processing*, E. R. Dougherty, Ed. Marcel Dekker, New York, 1993, ch. 10, pp. 323–349.

[77] *PVM: Parallel Virtual Machine, a user's guide and tutorial for networked parallel computing*, 1994.

[78] Quinn, M. *Parallel Computing. Theory and Practice*. McGraw-Hill, New York, NY, 1994.

[79] Roerdink, J. B. T. M., and Meijster, A. Segmentation by watersheds: definition and parallel implementation. In *Advances in Computer Vision*, F. Solina, W. G. Kropatsch, R. Klette, and R. Bajcsy, Eds. Springer, Wien, New York, 1997, pp. 21–30.

[80] Roerdink, J. B. T. M., and Meijster, A. The watershed transform: definitions, algorithms, and parallellization strategies. *Fundamenta Informaticae 41* (2000), 187–228.

[81] Rosenfeld, A., and Pfaltz, J. Distance functions on digital pictures. *Pattern Recognition 1* (1968), 33–61.

[82] Rosenfeld, A., and Pfaltz, J. L. Sequential operations in digital picture processing. *J. Ass. Comp. Mach. 13* (1966), 471–494.

[83] Salembier, P., Oliveras, A., and Garrido, L. Anti-extensive connected operators for image and sequence processing. *IEEE Trans. Image Proc. 7* (1998), 555–570.

[84] Salembier, P., and Serra, J. Flat zones filtering, connected operators, and filters by reconstruction. *IEEE Trans. Image Proc. 4* (1995), 1153–1160.

[85] Samet, H. Connected component labeling using quadtrees. *J. Ass. Comp. Mach. 28*, 3 (1981), 487–501.

[86] Serra, J. *Image Analysis and Mathematical Morphology*, 2 ed., vol. 1. Academic Press, New York, 1982.

[87] Sofou, A., Tzafestas, C., and Maragos, P. Segmentation of soilsection images using connected operators. In *Int. Conf. Image Proc. 2001* (2001), pp. 1087–1090.

[88] Tarjan, R. E. Efficiency of a good but not linear set union algorithm. *J. ACM 22* (1975), 215–225.

[89] Tarjan, R. E. *Data Structures and Network Algorithms*. SIAM, 1983.

[90] Tarjan, R. E., and van Leeuwen, J. Worst-case analysis of set union algorithms. *J. Ass. Comp. Mach. 31*, 2 (1984), 245–281.

[91] Tel, G. *Distributed Algorithms*. Cambridge University Press, 1994.

[92] Urbach, E. R., and Wilkinson, M. H. F. Shape distributions and decomposition of grey scale images. IWI-report 2000-9-15, Institute for Mathematics and Computing Science, University of Groningen, 2001.

[93] Urbach, E. R., and Wilkinson, M. H. F. Shape-only granulometries and grey-scale shape filters. In *Proc. Int. Symp. Math. Morphology (ISMM) 2002* (2002), pp. 305–314.

[94] van de Snepscheut, J. L. A. *What Computing is all about*. Springer-Verlag, New York–Heidelberg–Berlin, 1993.

[95] Verbeek, P. W., and Verwer, B. J. H. Shading from shape, the eikonal equation solved by gray-weighted distance transform. *Pattern Recognition Letters 11* (1990), 681–690.

[96] Viero, T. *Algorithms for image sequence filtering, coding and image segmentation*. PhD thesis, Tampere University of Technology, Tampere, Finland, Jan. 1996.

[97] Vincent, L. *Algorithmes Morphologiques a Base de Files d'Attente et de Lacets. Extension aux Graphes*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, Fontainebleau, 1990.

[98] Vincent, L. Grayscale area openings and closings, their efficient implementation and applications. In *Proc. EURASIP Workshop on Mathematical Morphology and its Application to Signal Processing* (Barcelona, Spain, 1993), pp. 22–27.

[99] Vincent, L. Morphological area openings and closings for grey-scale images. In *Shape in Picture: Mathematical Description of Shape in Grey-level Images*, Y.-L. O, A. Toet, D. Foster, H. J. A. M. Heijmans, and P. Meer, Eds. NATO, 1993, pp. 197–208.

[100] Vincent, L. Morphological grayscale reconstruction in image analysis: application and efficient algorithm. *IEEE Trans. Image Proc. 2* (1993), 176–201.

[101] Vincent, L. Granulometries and opening trees. *Fundamenta Informaticae 41* (2000), 57–90.

[102] Vincent, L., and Soille, P. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *IEEE Trans. Patt. Anal. Mach. Intell. 13*, 6 (1991), 583–598.

[103] Wilkinson, M. H. F., and Roerdink, J. B. T. M. Fast morphological attribute operations using Tarjan's union-find algorithm. In *Proc. Int. Symp. Math. Morphology (ISMM) 2000* (Palo Alto, CA, June 2000), pp. 311–320.

[104] Wilkinson, M. H. F., and Westenberg, M. A. Shape preserving filament enhancement filtering. In *Proc. MICCAI'2001* (2001), W. J. Niessen and M. A. Viergever, Eds., vol. 2208 of *Lecture Notes in Computer Science*, pp. 770–777.

Publications

Papers in scientific journals

Meijster, A., and Wilkinson, M. H. F. A comparison of algorithms for connected set openings and closings. *IEEE Trans. Pattern Anal. Mach. Intell. 24*, 4 (2002), 484–494.

Hesselink, W. H., Meijster, A., and Bron, C. Concurrent determination of connected components. *Sci. Comp. Programming 41* (2001), 173–194.

Roerdink, J. B. T. M., and Meijster, A. The watershed transform: definitions, algorithms, and parallellization strategies. *Fundamenta Informaticae 41* (2000), 187–228.

Full papers in conference proceedings

Meijster, A., Westenberg, M. A., and Wilkinson, M. H. F. Interactive shape preserving filtering and visualization of volumetric data. In *Fourth IASTED Conf. Comp. Signal Image Proc., SIP2002* (Kauai, Hawaii, USA, August 12-14, 2002), pp. 640–643.

Meijster, A., and Wilkinson, M. H. F. Fast computation of morphological area pattern spectra. In *Int. Conf. Image Proc. 2001* (2001), pp. 668–671.

Meijster, A., Roerdink, J. B. T. M., and Hesselink, W. H. A general algorithm for computing distance transforms in linear time. In *Proc. Int. Symp. Math. Morphology (ISMM) 2000* (Palo Alto, CA, June 2000), pp. 331–340.

Meijster, A., and Wubs, F. Towards an implementation of a Multilevel ILU Preconditioner on Shared-Memory Computers. In *Proc. HPCN Europe 2000*, May 8-10, Amsterdam, Netherlands, Lect. Notes Comp. Science 1823, 2000, pp. 109–118.

Meijster, A., and Roerdink, J. B. T. M. A disjoint set algorithm for the watershed transform. In *Proc. IX European Signal Processing Conf. (EUSIPCO'98), September 8 - 11, 1998, Rhodes, Greece* (1998), S. Theodoridis, I. Pitas, A. Stouraitis, and N. Kalouptsidis, Eds., pp. 1665–1668. *Note: Received best young author paper award for EUSIPCO-98.*

Roerdink, J. B. T. M., and Meijster, A. Segmentation by watersheds: definition and parallel implementation. In *Advances in Computer Vision*, F. Solina, W. G. Kropatsch, R. Klette, and R. Bajcsy, Eds. Springer, Wien, New York, 1997, pp. 21–30.

Meijster, A., and Roerdink, J. B. T. M. Computation of watersheds based on parallel graph algorithms. In *Mathematical Morphology and its Applications to Image and Signal Processing*, P. Maragos, R. W. Shafer, and M. A. Butt, Eds. Kluwer Acad. Publ., Dordrecht, 1996, pp. 305–312.

Meijster, A., and Roerdink, J. B. T. M. The implementation of a parallel watershed algorithm. In *Proc. Computing Science in the Netherlands*, November 27-28, Utrecht, Netherlands, 1995, pp. 134–142.

Meijster, A., and Roerdink, J. B. T. M. A proposal for the implementation of a parallel watershed algorithm. In *Computer Analysis of Images and Patterns*, V. Hlaváč and R. Šára, Eds., vol. 970 of *Lecture Notes in Computer Science*. Springer-Verlag, New York–Heidelberg–Berlin, 1995, pp. 790–795.

Chapters in Books

Roerdink, J. B. T. M., and Meijster, A. The watershed transform: definitions, algorithms, and parallellization strategies. In *Mathematical Morphology*, J. Goutsias and H.J.A.M. Heijmans (eds.), IOS Press, 2000, pp. 187–228.

Other publications

Meijster, A., and Wilkinson, M. H. G. An efficient algorithm for morphological area operators. Technical report 99-9-07, Institute for Mathematics and Computing Science, University of Groningen.

Meijster, A., and Roerdink, J. B. T. M. An alternative algorithm for computing watersheds on shared memory parallel computers. Summer School on Morphological Image and Signal Processing, 27-30 September, Zakopane, Poland, 1995, pp. 37–42.

Samenvatting

Het onderwerp van dit proefschrift is efficiënte algoritmen voor morfologische beeldverwerking. Ieder hoofdstuk behandelt een algoritme uit de mathematische morfologie. Er worden overzichten gegeven van reeds bestaande algoritmen. Tevens worden nieuwe algoritmen of verbeteringen van reeds bestaande algoritmen geïntroduceerd. In de meeste hoofdstukken ligt de nadruk op het ontwerp en de implementatie van parallelle algoritmen. De interesse voor parallelle algoritmen wordt met name gevoed door de toenemende vraag naar snelle algoritmen voor real-time beeldverwerkingstaken en de verwerking van drie-dimensionale volume data. Parallelle algoritmen voor de verwerking van 3D volume data zijn met name interessant in de medische beeldverwerking, waar het gebruik van grote MRI en CT volumes tegenwoordig de dagelijkse praktijk is.

In hoofdstuk 2 wordt een generiek algoritme voor de berekening van zgn. *afstandtransformaties* in lineaire tijd van een binair beeld gepresenteerd. Een afstandtransformatie berekent een grijswaarde beeld waarvan de waarde in iedere pixel zijn afstand tot het dichtstbijzijnde voorgrond pixel in het binaire beeld representeert. Het gepresenteerde algoritme is generiek in de zin dat verschillende metrieken gebruikt kunnen worden. De metrieken die worden besproken in hoofdstuk 2 zijn de *Manhattan* of *City-block metriek* (L_1-metriek), de *schaakbord metriek* (L_∞-metriek) en de *Euclidische metriek* (L_2-metriek).

Het algoritme kan als volgt worden samengevat. In een eerste fase wordt iedere kolom C_x (gedefiniëerd als de punten (x,y) met x vast) afzonderlijk doorlopen. Voor ieder punt (x,y) uit C_x wordt de afstand $G(x,y)$ van (x,y) tot de dichtstbijzijnde voorgrond pixel uit dezelfde kolom van het binaire beeld bepaald. In een tweede fase wordt iedere rij R_y (gedefiniëerd als de punten (x,y) met y vast) afzonderlijk doorlopen, waarbij voor ieder punt (x,y) uit R_y het minimum van $(x-x')^2+G(x',y)^2$ voor de L_2-metriek, $|x-x'|+G(x',y)$ voor de L_1-metriek, en $|x-x'|$ **max** $G(x',y)$ voor de L_∞-metriek wordt bepaald, waarbij (x',y) de rij R_y doorloopt.

Beide fases bestaan uit twee scans, een voorwaartse en een achterwaartse scan. De tijdscomplexiteit van het algoritme is lineair in het aantal pixels. Het algoritme kan eenvoudig worden geparallelliseerd voor executie op een parallelle computer met gemeenschappelijk geheugen, doordat de berekening per rij (kolom) onafhankelijk is van de berekening in andere rijen (kolommen). Een zgn. barrier dient gebruikt te worden als een synchronisatiepunt tussen de twee fases.

Het algoritme blijkt erg efficiënt te zijn en schaalt bijna perfect op computerarchitecturen met gemeenschappelijk geheugen. Het kan eenvoudig worden uitgebreid zodat d-dimensionale afstandtransformaties ($d > 2$) kunnen worden uitgerekend door het te splitsen in d fasen, waarvan ieder een één-dimensionaal probleem oplost.

Hoofdstuk 3 beschrijft het ontwerp van een parallel algoritme voor de bepaling van *sa-*

menhangcomponenten in binaire en grijswaarde beelden. De punten van een beeld worden beschouwd als verbonden als zij buren van elkaar zijn en dezelfde grijswaarde bezitten. Een verzameling punten C is een samenhangcomponent als voor ieder punt uit C een verbonden pad van punten uit C bestaat naar ieder ander punt in de verzameling en geen punten aan C kunnen worden toegevoegd waarbij deze eigenschap behouden blijft. Een parallelle versie van Tarjan's UNION-FIND algoritme voor de bepaling van equivalentieklassen in een graaf wordt gebruikt.

We presenteren het algoritme voor ongerichte grafen, hoewel we geïnteresseerd zijn in de toepassing op beelden. Het ontwerp gaat door vier stadia. Eerst beschrijven we Tarjan's sequentiële algoritme. Distributie van de knopen van de graaf over een aantal processen leidt tot een message passing algoritme. Dit ontwerp wordt vervolgens afgebeeld op een architectuur met gemeenschappelijk geheugen d.m.v. wederzijdse uitsluiting en synchronisatie. Tenslotte wordt de wederzijdse uitsluiting en synchronisatie geïmplementeerd m.b.v. POSIX thread primitieven.

Het resultaat is een parallel algoritme, waarvan de hoeveelheid communicatie wordt bepaald gedurende de executie. Het gevolg is dat processen niet mogen stoppen als ze klaar zijn met één of andere taak, omdat andere processen mogelijk nieuwe taken zullen sturen. Dit leidt tot een terminatie detectie probleem, dat wordt opgelost d.m.v. het tellen van het aantal nog te verwerken taken.

Tenslotte beschrijven we de toepassing van het algoritme voor de bepaling van samenhangcomponenten in beelden. Omdat grafen dimensieloos zijn, is het algoritme toepasbaar voor beelden van iedere omvang en dimensie. We beschrijven een optimalisatie die de hoeveelheid communicatie behoorlijk kan reduceren. Deze optimalisatie maakt gebruik van het feit dat beelden meestal lokaal min of meer dezelfde grijswaarden bevatten. Uit experimenten blijkt dat de parallellisatie effectief is en dat de versnelling vaak bijna lineair is in het aantal processoren.

In hoofdstuk 4 wordt een overzicht gegeven van algoritmen voor zgn. *connected set openings* en *closings*. De nadruk ligt op *attribute openings* en *closings*. De twee meest bekende algoritmen voor deze filters worden vergeleken met een nieuwe algoritme dat is ontwikkeld door Meijster en Wilkinson. Het eerste algoritme is de priority-queue methode voor filtering op basis van oppervlakte gepubliceerd door Vincent (zie [99]) en later uitgebreid door Breen en Jones tot attribuut-filters (zie [19]). Het andere algoritme is gebaseerd op de zgn. *Max-tree* methode van Salembier en co-auteurs (zie [83]). De methode van Meijster en Wilkinson is gebaseerd op een variant van het UNION-FIND algoritme van Tarjan.

In dit hoofdstuk ligt de nadruk op de berekening van *area openings*, wat één van de eenvoudigste attribuut openingen is. Een area opening kent aan ieder pixel p de hoogste grijswaarde h toe waarbij de samenhangcomponent (waartoe p behoort) van het binaire beeld dat ontstaat na drempeling met drempelwaarde h een oppervlakte heeft van tenminste λ pixels. De area closing is het duale filter. In het geval van binaire beelden verwijdert een area opening alle samenhangcomponenten met een oppervlakte kleiner dan λ, terwijl een area closing alle achtergrond-componenten met een oppervlakte kleiner dan λ vult. Een area opening (closing) kan componenten uit een beeld volledig verwijderen, of ze volledig intact laten, maar nooit de vorm van een component veranderen.

Vincent gebruikt een priority-queue bij de implementatie van een breadth-first algoritme voor de bepaling van samenhangcomponenten. Alle regionale extrema van het beeld worden na elkaar verwerkt. Ieder extremum wordt uitgebreid met aangrenzende pixels totdat aan het oppervlak-

tecriterium is voldaan of een pixel met een hogere grijswaarde is bereikt. In het laatste geval wordt de verwerking van de betreffende component gestaakt, omdat deze later opnieuw bereikt zal worden bij de verwerking van een ander extremum.

Als N het aantal pixels in het image representeert dan is de tijdscomplexiteit van het algoritme $O(N\lambda\log\lambda)$, wat neerkomt op $O(N^2\log N)$ als $\lambda = N$. Deze laatste situatie doet zich voor bij de berekening van zgn. *pattern spectra* (zie sectie 4.7). Uit experimenten blijkt dat de executietijd van het algoritme sterk afhangt van de parameter λ, maar ook van het aantal regionale extrema.

Het algoritme van Salembier en co-auteurs bestaat uit twee fasen. Eerst wordt een boomstructuur, een zgn. *Max-tree*, geconstrueerd, waarna vervolgens deze boom wordt gebruikt tijdens een tweede filterfase. Een Max-tree is een boom waarvan de knopen de samenhangcomponenten representeren van de binaire beelden die ontstaan door het input beeld op alle mogelijke niveaus te drempelen. Iedere knoop C in de boom heeft een verwijzing naar een andere knoop die de samenhangcomponent is waarin C bevat is, als de dataset op een lagere grijswaarde wordt gedrempeld. De wortel van de boom is een samenhangcomponent die bestaat uit alle pixels van het beeld, terwijl de bladeren van de boom bestaan uit de regionale maxima.

Salembier's algoritme voor de constructie van de boom is recursief, waarbij gebruikt wordt gemaakt van een zgn. *hiërarchische queue*. Een hiërarchische queue is in feite een priority queue van queues. Deze queuestructuur wordt gebruikt voor bepaling van samenhangcomponenten, en voor het verwerken van aangrenzende componenten van een regionaal extremum in volgorde van grijswaarde. In de knopen van de boom worden attributen, zoals oppervlakte, opgeslagen. Het filteren van het beeld komt nu neer op het volgen van de verwijzingen in de boom, totdat een knoop is bereikt die aan het filter criterium voldoet. Alle afstammelingen van deze knoop krijgen nu de grijswaarde die behoort bij deze knoop.

De rekentijd van het Max-tree algoritme wordt gedomineerd door de bepaling van de samenhangcomponenten. Deze fase heeft een lineaire tijdscomplexiteit in het aantal pixels en de connectiviteit van het beeld. Het filteren vereist dat alle knopen in de boom bezocht worden, en dit kost ook $O(N)$ tijd. Het bepalen van het uitvoerbeeld is eveneens een berekening met lineaire tijdscomplexiteit. Een opmerkelijke eigenschap van de tijdscomplexiteit van het Max-tree algoritme is zijn afhankelijkheid van het aantal onbezette grijswaarden tussen de knopen van de boom, omdat lineair zoeken naar de eerstvolgende bezette grijswaarde nodig is.

Het belangrijkste verschil tussen het algoritme gebaseerd op UNION-FIND en de algoritmen van Vincent en Salembier is het feit dat het UNION-FIND algoritme meerdere extrema gelijktijdig verwerkt, terwijl de andere twee algoritmen de extrema één voor één behandelen. Het algoritme is gebaseerd op het idee uit hoofdstuk 3 om de samenhangcomponenten m.b.v. UNION-FIND te bepalen, en het idee van Salembier om een boom te construeren. De boom is vervangen door een bos van kleinere bomen. Iedere boom wordt uitgebreid totdat de wortel van de boom aan het filter criterium voldoet.

Het UNION-FIND algoritme blijkt sneller te zijn dan de andere methoden voor vrijwel alle natuurlijke en kunstmatige beelden. De tijdscomplexiteit van het Max-tree algoritme is onafhankelijk van λ, terwijl het UNION-FIND algoritme een zwakke afhankelijkheid vertoont. Desalniettemin blijkt uit experimenten met 2D beelden dat het Max-tree algoritme ongeveer een factor 2 langzamer is dan het UNION-FIND algoritme. Het voordeel blijkt zelfs groter bij de verwerking van 3D volumes. Dit is waarschijnlijk te verklaren uit het feit dat het UNION-FIND minder ge-

heugen gebruikt, wat met name van belang is vanwege de beperkte bandbreedte van de memory bus van PCs. In het geval van een $128 \times 128 \times 62$ volume gebruikt het Max-tree algoritme 57.5 MB geheugen, het priority-queue algoritme gebruikt 45 MB, terwijl slechts 25 MB geheugen nodig is voor het UNION-FIND algoritme.

In hoofdstuk 4 wordt tevens kort besproken hoe de algoritmen uitgebreid kunnen worden, zodat ze gebruikt kunnen worden voor het berekenen van *attribute openings* en *pattern spectra*. Het blijkt mogelijk om een area pattern spectrum uit te rekenen in dezelfde tijd die nodig is voor het berekenen van een enkele area opening.

Hoewel het attribute opening (closing) algoritme van Meijster en Wilkinson sneller blijkt te zijn dan het Max-tree algoritme in het geval van een gegeven dataset en een vaste λ, heeft het Max-tree algoritme toch zijn voordelen. Het is namelijk erg geschikt voor het interactief filteren van een dataset door eerst de boomstructuur uit te rekenen, waarna een gebruiker d.m.v. een grafisch user interface de gewenste parameter λ kan kiezen. Het filteren van de boomstructuur is een snelle operatie, terwijl de constructie van de boom de tijdsintensieve stap is. Deze kan echter offline plaatsvinden. Met het oog op deze toepassing behandelt hoofdstuk 5 een algoritme voor de parallelle constructie en filtering van Max-trees.

In hoofdstuk 6 wordt een uitgebreid en kritisch overzicht gegeven van verscheidene definities van de zgn. *waterscheidingstransformatie* (*watershed transform*) gebaseerd op *onderdompeling* (*immersion*) of kortste paden m.b.t. een zgn. *topografische afstandsfunctie*.

De waterscheidingstransformatie kan geclassificeerd worden als een zgn. *region-based* segmentatiemethode. Het idee is om een grijswaarden beeld te beschouwen als een landschap, waarvan de grijswaarden hoogten representeren. Een intuïtieve beschrijving van het waterscheidingsalgoritme gaat dan als volgt. Stel je voor dat kleine gaatjes geprikt worden in de lokale minima, waarna het gehele landschap langzaam wordt ondergedompeld in een groot meer. Er zullen expanderende bassins (zgn. *catchment basins*) vanuit de gaatjes ontstaan. Op de plaatsen waar water uit verschillende bassins elkaar gaat raken, wordt een dam gebouwd. Het proces eindigt als het water de hoogste piek in het landschap heeft bereikt. Het resultaat is een landschap dat volledig is gepartitioneerd in gebieden (bassins), gescheiden door dammen, de zgn. *waterscheidingslijnen* (*watershed lines*).

In dit hoofdstuk worden de belangrijkste sequentiële algoritmen voor het berekenen van de watershed transform beschreven. Er wordt nadruk gelegd op het feit dat vele watershed algoritmen uit de literatuur niet voldoen aan hun definitie, of de implementatie zijn van een algoritme zonder een goede specificatie.

Strategieën voor parallel implementaties worden beschreven voor zowel gedistribueerde als gemeenschappelijke geheugen systemen. Het waterscheiding algoritme gebaseerd op simulatie van onderdompeling is erg moeilijk te parallelliseren, vanwege de inherente sequentiële aard van het onderdompelingsproces. Een parallelle implementatie van dit algoritme is mogelijk d.m.v. een transformatie in een graaf van samenhangcomponenten, waarna een watershed van de graaf wordt uitgerekend.

De definitie van de watershed die gebaseerd is op afstanden laat verscheidene parallelle implementaties toe. De belangrijkste zijn gebaseerd op (hiërarchische) queues, herhaald raster scanning, een variant op het UNION-FIND algoritme, of een combinatie van deze mogelijkheden.

De belangrijkste conclusie die getrokken kan worden uit dit overzicht is dat, ondanks alle

technieken en architecturen die gebruikt worden, er altijd een stadium in het watershed algoritme blijft dat een globale operatie behelst, waardoor er in het geval van parallellisatie een slechts bescheiden versnelling verwacht mag worden.

Dankwoord

Op deze plaats wil ik graag enkele mensen bedanken, die mij geholpen hebben bij het tot stand brengen van dit proefschrift.

Om te beginnen mijn eerste promotor, Jos Roerdink. Ik wil hem bedanken voor zijn begeleiding en vertrouwen in het afronden van dit proefschrift. Ik heb zelf altijd graag een proefschrift willen schrijven dat vanaf het eerste tot het laatste hoofdstuk een samenhangend boekwerk is en niet een proefschrift bestaande uit artikelen. Op advies van Jos, heb ik uiteindelijk besloten te capituleren en toch het proefschrift uit artikelen te laten bestaan. Jos stelde ”Een proefschrift is niet je definitieve levenswerk. Het moet ook ooit eens af zijn.”. Mijn werk bij het rekencentrum bleek in de praktijk niet te combineren met het afronden van het oorspronkelijk geplande boekwerk en nood breekt dan wet. Jos, ik denk dat je uiteindelijk gelijk hebt en hoop dat we in de toekomst nog eens samen een aantal artikelen kunnen schrijven als spin-off van dit proefschrift, over een aantal ideeën die ik nog heb en noodgedwongen niet in dit proefschrift heb kunnen opnemen. Verder wil ik je bedanken voor het zeer nauwgezet lezen van stukken tekst die ik heb geschreven en je LaTeX-hulp in het finale stadium van het afronden van dit boekje.

Een andere zeer grote bijdrage aan dit proefschrift is geleverd door mijn tweede promotor Wim Hesselink. Een aantal algoritmen uit dit proefschrift is ontstaan uit intuïtieve ideeën, die ik vervolgens testte op geconstrueerde invoerbeelden. Meestal bevatte zo’n eerste implementatie wel een aantal fouten. Vervolgens riep ik de hulp van Wim in om zijn licht over mijn oplossing te laten schijnen. Het resultaat was dan meer dan eens een volledig nieuw algoritme, waar met veel moeite mijn oorspronkelijke idee in terug te vinden was. Wim is een meester in het vertalen van een intuïtief gesteld probleem in een formele specificatie, om dan vanuit deze specificatie stap voor stap een programma met correctheidsbewijs af te leiden. We hebben vaak gezamenlijk op het bord in zijn kamer op deze wijze aan mijn algoritmen gewerkt en ik vond dit zeer plezierig, hoewel het voor mij een harde leerschool is om geen plaatjes te mogen tekenen ter ondersteuning van mijn argumenten. Het leidde tot een aantal gezamenlijke publicaties, waarop we beide onze stempels hebben gedrukt. De signatuur van de artikelen, waarbij Wim co-auteur is, is duidelijk formeler dan die van de overige artikelen, terwijl de context toch duidelijk beeldverwerking blijft. Ik vind het jammer dat het in dit vakgebied zo moeilijk is om mensen geïnteresseerd te krijgen voor deze formele benadering van het ontwikkelen van algoritmen voor beeldbewerkingstaken, want ik ben nu toch wel erg vaak algoritmen in de beeldverwerkingliteratuur tegen gekomen die enige (kleine) foutjes bevatten.

Nicolai Petkov wil ik bedanken voor het dragen van de initiële verantwoordelijkheid voor mijn begeleiding en zijn adviezen. Naast het wetenschappelijk werk heb ik Nicolais groepsevenementen (dagjes zeilen) erg gewaardeerd. Ik kan dit soort activiteiten iedere onderzoeksgroep aanraden.

Harm Bakker, Jan-Eppo Jonker, en Jan Jongejan wil ik bedanken voor de geregelde discussies die we hebben gehad over de algoritmen in dit proefschrift en de gezellige tijd die we hebben besteed achter de toetsenborden van onze terminals, terwijl we weer eens één van mijn ideeën probeerden te programmeren. Uit deze activiteiten is me meer dan eens gebleken dat programmeren teamwork is en niet iets voor een enkel persoon, die zich opsluit op zijn werkkamer. Een team maakt minder fouten dan een individu. Dit verklaart waarschijnlijk ook onze gezamenlijke passie voor de jaarlijkse programmeerwedstrijd van het IWI.

Met Michael Wilkinson heb ik drie papers gepubliceerd. Alle drie over de implementatie van connected set filters en hun toepassingen. Michael, bedankt voor je enthousiaste samenwerking en de vele uren gezellig gebabbel. Ik hoop dat we in de toekomst nog vaker tot leuke algoritmen kunnen komen, door eenvoudigweg te brainstormen met een whiteboard en wat stiften.

Toen ik nog op het IWI werkte als AIO deelde ik een kamer met Peter Kruizinga en Tino Laurens. Hen wil ik bedanken voor de gezellige werksfeer en de discussies die we hebben gevoerd over onze onderzoeken. Ditzelfde geldt voor Michel Westenberg, die zijn eigen paar vierkante meters aan de andere zijde van de gang ter beschikking stelde als een soort AIO-kantine waar we onder het genot van een 'bakkie pleur' (dit is koffie, zelfs mijn spellingchecker snapt dit niet) o.a. onze worstelingen op het pad naar het schrijven van onze proefschriften bespraken. Alle drie hebben ze ruimschoots van mij gewonnen, door enige jaren eerder te promoveren. Ineke Kruizinga wil ik bedanken voor de vele gezellige uurtjes die vele collegae van het IWI en ik bij haar koffie drinkend hebben doorgebracht, en voor haar support als het even wat minder ging. Tenslotte wil ik nog vermelden dat ik met zeer veel plezier op het IWI heb gewerkt als studentassistent, AIO en docent. Ik wil al mijn ex-collegae op het IWI bedanken voor de plezierige samenwerking, i.h.b. Coen Bron, Eelco Dijkstra, Rudy Moddemeijer, Jos Nijhuis, Harm Paas, Roelof Sijtsma, en Arthur Veldman.

Mijn collegae op het Rekencentrum, m.n. Rob de Bruin en Laurens Voerman, wil ik bedanken voor de vele malen dat ik ze lastig mocht vallen met dit proefschrift. Als ik een nieuw idee had, dan waren Rob en Laurens nooit te beroerd om wat tijd uit te trekken om een fout in mijn redenering te zoeken.

Mijn paranimfen Ger Battjes en Harrie Bos wil ik bedanken voor hun bereidheid mij te steunen tijdens de verdediging van dit proefschrift.

Mijn familie wil ik bedanken voor hun belangstelling en steun. In het bijzonder wil ik mijn ouders bedanken voor hun materiële en immateriële steun tijdens mijn studie en in de daaropvolgende AIO-periode.

Last, but not least, wil ik mijn partner Ernee Bos bedanken voor haar geduld en begrip. Ik vermoed dat het schrijven van een proefschrift niet alleen een proeve van bekwaamheid is van de promovendus, maar tevens een zeer grondige test van het doorzettingsvermogen van zijn of haar partner. Lieve Ernee, bedankt.