Fundamenta Informaticae 41 (2001) 187–228
IOS Press

The Watershed Transform: Definitions, Algorithms and Parallelization Strategies

Jos B.T.M. Roerdink and Arnold Meijster
Institute for Mathematics and Computing Science
University of Groningen
P.O. Box 800, 9700 AV Groningen, The Netherlands
Email: roe@cs.rug.nl,a.meijster@rc.rug.nl

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.

Keywords: Mathematical morphology, watershed transform, watershed definition, sequential algorithms, parallel implementation.

1. Introduction

In grey scale mathematical morphology the watershed transform, originally proposed by Digabel and Lantuéjoul [9,20] and later improved by Beucher and Lantuéjoul [4], is the method of choice for image segmentation [5, 46, 52]. 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 [18].

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 [46]. 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 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. 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 [26]. 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 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. [26,51] for a survey. They can be divided into two classes, one based on the specification of a recursive algorithm by Vincent & Soille [52], and another based on distance functions by Meyer [25]. 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 5.

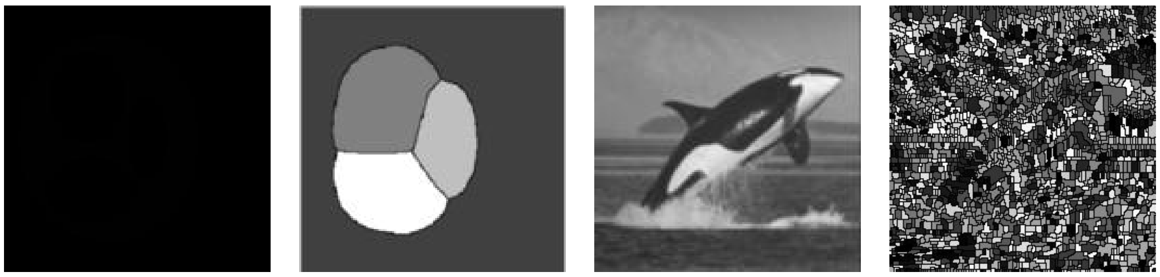
(a) (b) (c) (d)

Figure 1. Examples of watershed segmentation by immersion (see Definition 3.2). (a): synthetic image; (b): watershed transform of (a); (c): natural image; (d): watershed transform of (c). Different basins are indicated by distinct grey values.

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 unwanted 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 continuous 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 [25]. 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. 1(c-d). Several approaches exist to remedy this, such as markers or hierarchical watersheds [3, 26]; also parallellization of marker-based watershed algorithms has been studied [27, 31]. Third, we do not consider dedicated hardware architectures for fast computation of watershed transforms and related operations, see e.g. [19,37]. 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 2 some preliminaries are given. Section 3 presents definitions of the watershed transform, both for the continuous and the discrete case. Sequential watershed algorithms are reviewed in Section 4. Section 5 contains a survey of parallelization strategies for the watershed transform. Conclusions are drawn in Section 6.

2. Preliminaries

This section contains some background material on graphs (see e.g. [8]) and digital images.

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 (or *adjacent* to) 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 $V' \subseteq V$, $E' \subseteq E$, and the elements of E' are incident with vertices from V' only. 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.

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 $\mathbb{Z}^2 \times \mathbb{Z}^2$ 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 non-negative 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).

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

$$T_h = \{p \in D \mid f(p) \leq h\}. \tag{2.1}$$

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 $d_A(a, B) = \textsc{min}_{b \in B}(d_A(a, b))$. 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

$$iz_A(B_i) = \{p \in A \mid \forall j \in [1..k] \backslash \{i\} : d_A(p, B_i) < d_A(p, B_j)\}$$

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

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

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

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

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.

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 [52], and the definition by topographical distance by Meyer [25]. A discussion of algorithms is postponed until Section 4.

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 [25, 36], but other choices have been proposed as well [39].

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 [17, 35]). 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 $T_f(p, A) = \textsc{min}_{a\in A}\, T_f(p, a)$. 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 3.1. (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:*

$$Wshed(f) = D \cap \left(\bigcup_{i\in I} CB(m_i)\right)^c . \tag{3.1}$$

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.

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 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.

7	6	5	4
8	5	4	3
9	4	3	2
0	3	2	**1**

(a) original

B	B	B	B
B	B	B	B
W	B	B	B
A	W	B	B

(b) 4-conn.

B	B	B	B
B	B	B	B
W	W	B	B
A	W	B	B

(c) 8-conn.

W	W	B	B
W	W	B	B
A	W	B	B
A	A	B	B

(d) 4-conn.

B	B	B	B
A	B	B	B
A	A	B	B
A	A	B	B

(e) 8-conn.

Figure 2. Watershed transform on the square grid, for different connectivity. (a): original image (minima indicated in bold); (b-c): results according to immersion (Definition 3.2); (d)-(e): results according to topographical distance (Definition 3.1, with T_f as defined in (3.5)).

3.2.1. Algorithmic definition by immersion

An algorithmic definition of the watershed transform by simulated immersion was given by Vincent and Soille [51, 52] (see also [46, 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. (2.1)) 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 2.4), resulting in an update X_{h+1}. Let MIN_h denote the union of all regional minima at altitude h.

Definition 3.2. (Watershed by immersion) *Define the following recursion:*

$$\begin{cases} X_{h_{min}} &= \{p \in D \mid f(p) = h_{min}\} = T_{h_{min}} \\ X_{h+1} &= \text{MIN}_{h+1} \cup IZ_{T_{h+1}}(X_h), \qquad h \in [h_{min}, h_{max}) \end{cases} \tag{3.2}$$

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. 2(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.

3	2	2
3	1	1
0	1	**0**

(a)

3	2	2
3	1	1
A	1	B

(b) $h = 0$

3	2	2
3	W	B
A	W	B

(c) $h = 1$

3	B	B
3	B	B
A	W	B

(d) $h = 2$

B	B	B
W	B	B
A	W	B

(e) $h = 3$

Figure 3. Watershed transform by immersion on the 4-connected grid, showing relabelling of 'watershed' pixels. (a): Original image; (b-e): labelling steps based on (3.2).

According to the recursion (3.2), 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 [42] is given in Fig. 3, 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 (3.2) is shown in Fig. 3(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 [52] as an implementation of (3.2) in fact does not adhere to this definition, see Section 4.1 below.

3.2.2. Watershed definition by topographical distance

We follow here the presentation in [25]. 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 [26]. 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,

$$LS(p) = \underset{q \in N_G(p) \cup \{p\}}{\mathrm{MAX}} \left(\frac{f(p) - f(q)}{d(p,q)} \right), \tag{3.3}$$

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 (3.3) is defined to be zero). Note that for pixels whose neighbours are all of higher grey value, the lower slope is zero. The *cost* for walking from pixel p to a neighbouring pixel q is defined as

$$cost(p,q) = \begin{cases} LS(p) \cdot d(p,q) & \text{if } f(p) > f(q) \\ LS(q) \cdot d(p,q) & \text{if } f(p) < f(q) \\ \frac{1}{2}(LS(p) + LS(q)) \cdot d(p,q) & \text{if } f(p) = f(q) \end{cases} \tag{3.4}$$

Definition 3.3. *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

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

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

$$T_f(p,q) = \operatorname*{MIN}_{\pi \in [p \rightsquigarrow q]} T_f^\pi(p,q), \tag{3.5}$$

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.

Proposition 3.1. *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 3.1.

It is a consequence of Proposition 3.1 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 3.1. *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. 2(d-e). Note that the result differs from that obtained by immersion according to Definition 3.2. A consequence of Definition 3.1 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. 4. 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.

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 4. Watershed transform on the square grid with 4-connectivity, showing thick watersheds. (a): original image; (b): result according to topographical distance (Definition 3.1, with T_f as defined in (3.5)); (c): result according to immersion (Definition 3.2).

Remark. A distance transform [43] 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 [25]. □

Next we consider images which are not lower complete.

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. Therefore an additional ordering relation between such pixels is required. 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 $\Pi^{\downarrow}_f(p)$ 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 3.4. (Lower completion) *Let f be a digital grey value image with domain D. Define the function $d : D \to \mathbb{N}$ by*

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

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

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

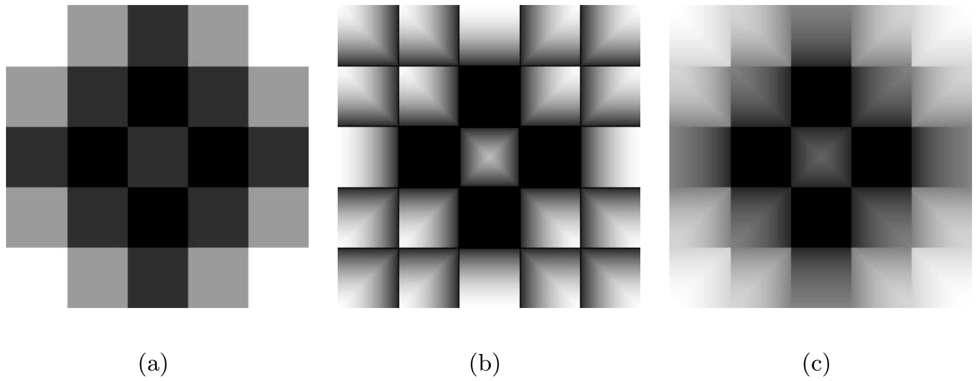

Figure 5. Image (a), lower distance image (b) and lower complete image (c).

The process of lower completion transforms the image f into a lower complete image f_{LC}. An example is given in Fig. 5. 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 4.5).

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

$$x \sqsubset y \iff f_{LC}(x) < f_{LC}(y). \tag{3.6}$$

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* [5,25] as follows.

Definition 3.5. *Let $G = (V, E, f)$ be a digital grey value image. The* lower complete graph *$G' = (V, E')$ is defined as follows. For points p having a lower neighbour,*

$$(p, p') \in E' \iff p' \in \Gamma(p) \tag{3.7}$$

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 3.6. (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:*

$$CB(m_i) = \{p \in D \mid \forall j \in I \backslash \{i\} : f^*(m_i) + T_{f^*}(p, m_i) < f^*(m_j) + T_{f^*}(p, m_j)\}, \tag{3.8}$$

and the watershed of f is defined as in (3.1).

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.

16	15	14	13	12	13
14	13	12	11	10	11
15	14	13	9	8	9
1	2	3	7	6	8
0	7	5	**4**	5	**0**

(a)

A	A	A	A	C	C
A	A	A	A	C	C
A	A	A	A	C	C
A	A	A	A	C	C
A	A	A	B	C	C

(b)

16	15	14	13	12	13
14	13	12	11	10	11
15	14	13	9	8	9
1	2	3	7	6	8
0	7	5	**0**	5	**0**

(c)

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

(d)

Figure 6. 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).

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 3.6, depending on the precise implementation. This will be discussed in more detail in Section 4.2.

Lowering the minima values. Meyer states in [25] 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 3.2 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 [50]. An example illustrating this is given in Fig. 6, 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).

'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. 7. 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 [12, 50]. This is a defect of the particular implementation, since, according to Corollary 3.1, 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.

0	1	**0**
1	2	1
0	1	**0**

(a) original

A	W	B
W	2	W
C	W	D

(b) labelled

Figure 7. Watershed according to topographical distance (4-connectivity). (a): original image; (b): Output after labelling pixels with grey values 0 and 1.

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. 4, although the use of a higher connectivity 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. [6, 7], by introducing a *local condition.*

Definition 3.7. *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 3.3). 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 3.6, are now merged by random choice with a basin belonging to some minimum m_k. For an example, see Fig. 8.

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 5.2.3).

5	4	5
4	3	4
6	2	6
0	1	**0**

(a)

W	W	W
W	W	W
A	W	B
A	W	B

(b)

A	A	A
A	A	A
A	A	B
A	A	B

(c)

B	B	B
B	B	B
A	B	B
A	B	B

(d)

Figure 8. Watershed transform on the 4-connected square grid. (a): original image; (b): result according to topographical distance (Definition 3.6); (c-d): two watershed labellings consistent with the local condition (Definition 3.7).

For an input image which would contain watershed pixels according to Definition 3.6, the output of a watershed algorithm based on Definition 3.7 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.

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.

4.1. Watershed algorithms by immersion

4.1.1. Vincent-Soille algorithm

An implementation of the watershed transform of Definition 3.2 was presented by Vincent & Soille [52]. Since we want to discuss this implementation in some detail, we reproduce their algorithm here in pseudocode, see Algorithm 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 [8], 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 [52], the time complexity of Algorithm 4.1 is linear in the number of pixels of the input image.

Algorithm 4.1 Vincent-Soille watershed algorithm [52].

```
procedure Watershed-by-Immersion
INPUT: digital grey scale image $G = (D, E, im)$.
OUTPUT: labelled watershed image $lab$ on $D$.
#define INIT $-1$                    (* initial value of $lab$ image *)
#define MASK $-2$                    (* initial value at each level *)
#define WSHED 0                      (* label of the watershed pixels *)
#define FICTITIOUS $(-1, -1)$        (* fictitious pixel $\notin D$ *)
$curlab \leftarrow 0$                (* $curlab$ is the current label *)
$fifo\_init(queue)$
for all $p \in D$ do
   $lab[p] \leftarrow$ INIT ; $dist[p] \leftarrow 0$    (* $dist$ is a work image of distances *)
end for
SORT pixels in increasing order of grey values (minimum $h_{min}$, maximum $h_{max}$)

(* Start Flooding *)
for $h = h_{min}$ to $h_{max}$ do          (* Geodesic SKIZ of level $h-1$ inside level $h$ *)
   for all $p \in D$ with $im[p] = h$ do          (* mask all pixels at level $h$ *)
              (* these are directly accessible because of the sorting step *)
      $lab[p] \leftarrow$ MASK
      if $p$ has a neighbour $q$ with ($lab[q] > 0$ or $lab[q] =$ WSHED) then
         (* Initialize queue with neighbours at level $h$ of current basins or watersheds *)
         $dist[p] \leftarrow 1$ ; $fifo\_add(p, queue)$
      end if
   end for
   $curdist \leftarrow 1$ ; $fifo\_add($FICTITIOUS$, queue)$
   loop          (* extend basins *)
      $p \leftarrow fifo\_remove(queue)$
      if $p =$ FICTITIOUS then
         if $fifo\_empty(queue)$ then
            BREAK
         else
            $fifo\_add($FICTITIOUS$, queue)$ ; $curdist \leftarrow curdist + 1$ ;
```

```
33:             $p \leftarrow fifo\_remove(queue)$
34:           end if
35:         end if
36:         for all $q \in N_G(p)$ 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:                 $lab[p] \leftarrow lab[q]$
42:               else if $lab[p] \neq lab[q]$ then
43:                 $lab[p] \leftarrow$ WSHED
44:               end if
45:             else if $lab[p] = $ MASK then
46:               $lab[p] \leftarrow$ WSHED
47:             end if
48:           else if $lab[q] = $ MASK and $dist[q] = 0$ then          (* $q$ is plateau pixel *)
49:             $dist[q] \leftarrow curdist + 1$ ; $fifo\_add(q, queue)$
50:           end if
51:         end for
52:       end loop
53:       (* detect and process new minima at level $h$ *)
54:       for all $p \in D$ with $im[p] = h$ do
55:         $dist[p] \leftarrow 0$          (* reset distance to zero *)
56:         if $lab[p] = $ MASK then          (* $p$ is inside a new minimum *)
57:           $curlab \leftarrow curlab + 1$ ;          (* create new label *)
58:           $fifo\_add(p, queue)$ ; $lab[p] \leftarrow curlab$
59:           while not $fifo\_empty(queue)$ do
60:             $q \leftarrow fifo\_remove(queue)$
61:             for all $r \in N_G(q)$ do          (* inspect neighbours of $q$ *)
62:               if $lab[r] = $ MASK then
63:                 $fifo\_add(r, queue)$ ; $lab[r] \leftarrow curlab$
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 (3.2), for the following reasons (the line numbers mentioned refer to the pseudocode of Algorithm 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 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 [52] 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 4.1 in order to implement the recursion (3.2) 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 $lab[q] = \text{WSHED}$ in line 20), the resetting of distances (line 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.

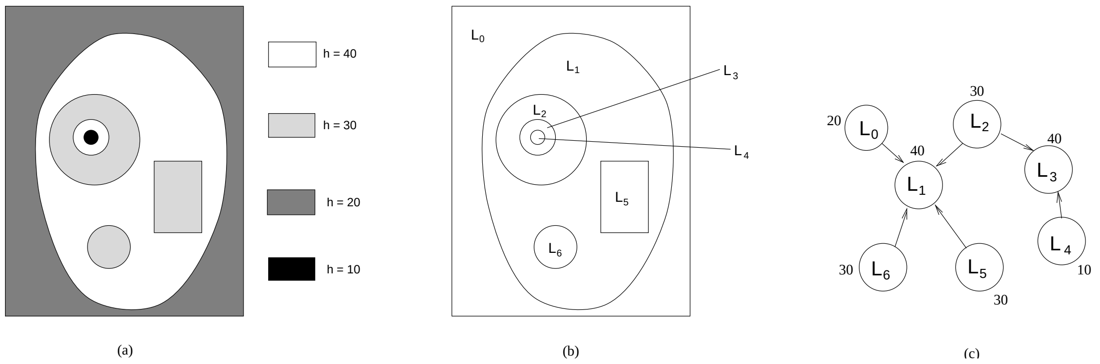

Figure 9. (a) input image. (b) labelled level components. (c) components graph, with grey values of the nodes indicated.

Remark. In [42] we tried to formalize what the Vincent-Soille algorithm computes by defining a modified recursion as follows:

$$\begin{cases} X_{h_{min}} &= \{p \in D \mid f(p) = h_{min}\} \\ X_{h+1} &= X_h \cup \text{MIN}_{h+1} \cup (IZ_{T_{h+1}}(X_h) \backslash T_h), \qquad h \in [h_{min}, h_{max}) \end{cases} \tag{4.1}$$

The '$\backslash T_h$' term in (4.1) 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. 3, the pixel in the second row, second column remains labelled as WSHED according to (4.1). However, also this modified recursion does not always correctly represent the implementation of Algorithm 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 4.1. □

4.1.2. Components graph algorithm

A straightforward parallel implementation of the Vincent-Soille algorithm is difficult when plateaus occur. Therefore, an alternative approach was proposed in [21], 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.

Algorithm 4.2 Watershed transform w.r.t. topographical distance based on image integration via the Dijkstra-Moore shortest paths algorithm.

```
procedure ShortestPathWatershed;
INPUT: lower complete digital grey scale image $G = (V, E, im)$ with cost function $cost$.
OUTPUT: labelled image $lab$ on $V$.
#define WSHED 0                    (* label of the watershed pixels *)
(* Uses distance image $dist$. On output, $dist[v] = im[v]$, for all $v \in V$. *)

for all $v \in V$ do        (* Initialize *)
  $lab[v] \leftarrow 0$ ; $dist[v] \leftarrow \infty$
end for
for all local minima $m_i$ do
  for all $v \in m_i$ do
    $lab[v] \leftarrow i$ ; $dist[v] \leftarrow im[v]$        (* initialize distance with values of minima *)
  end for
end for
while $V \neq \emptyset$ do
  $u \leftarrow GetMinDist(V)$        (* find $u \in V$ with smallest distance value $dist[u]$ *)
  $V \leftarrow V \backslash \{u\}$
  for all $v \in V$ with $(u, v) \in E$ do
    if $dist[u] + cost[u, v] < dist[v]$ then
      $dist[v] \leftarrow dist[u] + cost(u, v)$
      $lab[v] \leftarrow lab[u]$
    else if $lab[v] \neq$ WSHED and $dist[u] + cost[u, v] = dist[v]$ and $lab[v] \neq lab[u]$ then
      $lab[v] =$ WSHED
    end if
  end for
end while
```

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. 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. 9.
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.

Algorithm 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: $S \leftarrow \{p \in V | \exists q \in N_G(p) : im[p] \neq im[q]\}$        (* interior pixels of minima excluded *)
9: while not $empty(S)$ do
10:   select point $p \in S$ with minimal grey value
11:   remove $p$ from $S$
12:   for all $q \in \Gamma^{-1}(p) \cap S$ do        (* label steepest upper neighbours of $p$ *)
13:     if $lab[q]$ = MASK then
14:       $lab[q] \leftarrow lab[p]$
15:     else if $lab[q] \neq$ WSHED and $lab[q] \neq lab[p]$ then
16:       $lab[q]$ = WSHED
17:     end if
18:   end for
19: end while
```

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 [5, 25, 26].

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 [10] and Moore [34].
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 (3.4). On output, the distance value of a pixel p equals $f(p)$, where f is the input image. The pseudocode is given in Algorithm 4.2, which is described in more detail in Section 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 4.3, see Section 4.2.3 for details.

Unordered algorithms. The shortest path algorithm of Berge [2] 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 [49]. The implementation is based on an iterative algorithm [25] which integrates the lower slope of the input image, see

Algorithm 4.4. In [25] a variant is mentioned based on propagation of labelled pixels to steepest upper neighbours, as in hill climbing.

Algorithm 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 $v \in D$. *)
 6:
 7: for all $v \in D$ do        (* Initialize *)
 8:    $lab[v] \leftarrow 0$ ; $dist[v] \leftarrow \infty$
 9: end for
10: for all local minima $m_i$ do
11:    for all $v \in m_i$ do
12:       $lab[v] \leftarrow i$ ; $dist[v] \leftarrow im[v]$        (* initialize distance with values of minima *)
13:    end for
14: end for
15: $stable \leftarrow$ 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 $v \in N_G(u)$ in the future (w.r.t. scan order) of $u$ do
27:    if $dist[u] + cost[u, v] < dist[v]$ then
28:       $dist[v] \leftarrow dist[u] + cost(u, v)$
29:       $lab[v] \leftarrow lab[u]$
30:       $stable \leftarrow$ false
31:    else if $lab[v] \neq$ WSHED and $dist[u] + cost[u, v] = dist[v]$ and $lab[v] \neq lab[u]$ then
32:       $lab[v] =$ WSHED
33:       $stable \leftarrow$ false
34:    end if
35: end for
```

In [25] slightly different versions of the above algorithms are presented which do not produce watershed labels (lines 21-22 in Algorithm 4.2, lines 14-15 in Algorithm 4.3 and lines 30-32 in Algorithm 4.4 are omitted), and therefore are not exact implementations of Definition 3.6. 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 [25]. In fact, those algorithms are in agreement with the local definition of the watershed transform, as discussed in Section 3.2.3.

Algorithm 4.5 Algorithm for lower completion using a FIFO queue.

```
procedure LowerCompletion
INPUT: digital grey scale image $G = (D, E, im)$.
OUTPUT: lower complete image $G' = (D, E, lc)$.

$fifo\_init(queue)$
for all $p \in D$ do        (* Initialize queue with pixels that have a lower neighbour *)
    $lc[p] \leftarrow 0$
    if $p$ has a lower neighbour then
        $fifo\_add(p, queue)$
        $lc[p] \leftarrow -1$
    end if
end for
$dist \leftarrow 1$        (* $dist$ is an integer variable *)
$fifo\_add(\text{FICTITIOUS}, queue)$        (* insert fictitious pixel *)
while not $fifo\_empty(queue)$ do
    $p \leftarrow fifo\_remove(queue)$
    if $p = \text{FICTITIOUS}$ then
        if not $fifo\_empty(queue)$ then
            $fifo\_add(\text{FICTITIOUS}, queue)$
            $dist \leftarrow dist + 1$
        end if
    else
        $lc[p] \leftarrow dist$
        for all $q \in N_G(p)$ with $(im[q] = im[p]$ and $lc[q] = 0)$ do
            $fifo\_add(q, queue)$
            $lc[q] \leftarrow -1$        (* to prevent from queueing twice *)
        end for
    end if
end while

for all $p \in D$ do        (* Put the lower complete values in the output image *)
    if $lc[p] \neq 0$ then
        $lc[p] = dist \cdot im[p] + lc[p] - 1$
    else
        $lc[p] = dist \cdot im[p]$
    end if
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 [8] to propagate distances, cf. Algorithm 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.

4.2.1. Minima detection

Usually a flooding algorithm based on FIFO queues is used for minima detection [22, 29, 32]. However, the UNION-FIND algorithm for implementing disjoint sets [48], see also [8, 47], 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 4.6 Computing level components by breadth-first search using a FIFO queue.

```
procedure LevelComponents
INPUT: digital grey scale image $G = (V, E, im)$.
OUTPUT: image $lab$ on $V$, with labelled level components.
#define INIT $-1$                    (* initial value of $lab$ image *)

for all $p \in D$ do
    $lab[p] \leftarrow$ INIT
end for
$curlab \leftarrow 1$      (* $curlab$ is the current label *)
$fifo\_init(queue)$

for all $p \in V$ with $lab[p] =$ INIT do
    $lab[p] \leftarrow curlab$
    $fifo\_add(p, queue)$
    while not $fifo\_empty(queue)$ do
        $s \leftarrow fifo\_remove(queue)$
        for all $q \in N_G(s)$ with $im[s] = im[q]$ do
            if $lab[q] =$ INIT then
                $lab[q] \leftarrow curlab$
                $fifo\_add(q, queue)$
            end if
        end for
    end while
    $curlab \leftarrow curlab + 1$
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 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, 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.

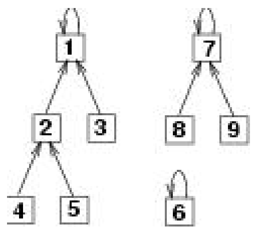

Figure 10. Disjoint set forest of sets of integers $\{1,2,3,4,5\}$, $\{6\}$, $\{7,8,9\}$.

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. 10 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 [47] 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 [47] 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 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 4.7), and p gets the label of its representative.

This algorithm can be used for the computation of connected components in images of any dimension, size and connectivity, in contrast to the algorithm of Rosenfeld-Pfaltz [44], which works only for 2-dimensional images using 4-connectivity. The same restriction holds for the UNION-FIND algorithm in [14] 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.

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 [8, 10]. 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 4.4, where *lab*$[v]$ is the index of the minimum nearest to v, and *dist*$[v]$ is the distance to this minimum [22]. 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. (3.8). 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 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 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 [5, 24]. The OQ processes lower grey

Algorithm 4.7 Scan-line algorithm for labelling level components based on disjoint sets.

```
procedure UNION-FIND-ComponentLabelling
INPUT: grey scale image $im$ on digital grid $G = (D, E)$.
OUTPUT: image $lab$ on $D$, with labelled level components.
(* Uses array $parent$ of pointers. *)

(* First pass *)
for all $p \in D$ in lexicographical order do
    $r \leftarrow p$
    for all $q \in N_G(p)$ with $q \prec p$ do
        if $im[q] = im[p]$ then
            $r \leftarrow r$ min FindRoot $(q)$        (* min denotes minimum w.r.t. lexicographical order *)
        end if
    end for
    $parent[p] \leftarrow r$
    for all $q \in N_G(p)$ with $q \prec p$ do        (* compress paths *)
        if $im[q] = im[p]$ then
            PathCompress$(q, r)$
        end if
    end for
end for

(* Second pass *)
$curlab \leftarrow 1$        (* $curlab$ is the current label *)
for all $p \in D$ in lexicographical order do
    if $parent[p] = p$ then        (* $p$ is a root pixel *)
        $lab[p] = curlab$
        $curlab = curlab + 1$
    else
        $parent[p] = parent[parent[p]]$        (* Resolve unresolved equivalences *)
        $lab[p] = lab[parent[p]]$
    end if
end for

function FindRoot $(p : pixel)$
while $parent[p] \neq p$ do
    $r \leftarrow parent[p]$ ; $p \leftarrow r$
end while
 return $r$

procedure PathCompress$(p : pixel, r : pixel)$
while $parent[p] \neq r$ do
    $h \leftarrow parent[p]$ ; $parent[p] \leftarrow r$ ; $p \leftarrow h$
end while
```

levels 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 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. □

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 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. (3.3)), 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.

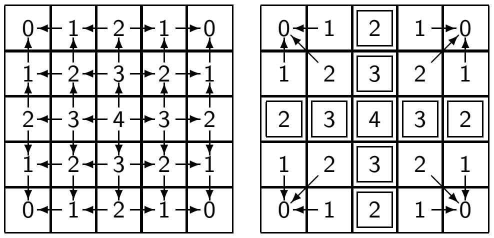

Figure 11. Left: image and its corresponding DAG; right: graph after resolving (watershed pixels are surrounded by a box).

4.2.4. Watershed transform by UNION-FIND algorithm

The UNION-FIND algorithm described in Section 4.2.1 can be modified to compute the watershed transform itself [23], 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 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 3.5), which is a directed acyclic graph (DAG). See Fig. 11 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 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 pseudocode of the resolving algorithm, which closely resembles Tarjan's *FindRoot* operation [47], see Algorithm 4.8.

This technique computes the exact watershed transform by topographical distance [25]. A similar approach was developed by Bieniek et al. [7], based on earlier work [6] on parallel implementation of the watershed transform. However, these authors use the local condition in which no watershed pixels are computed (see Sections 3.2.3, 5.2.3); when several steepest 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.

Algorithm 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 $p \in V$ do          (* give $p$ the label of its representative *)
10:    $rep \leftarrow$ Resolve ($p$)
11:    if $rep \neq$ W then
12:       $lab[p] \leftarrow lab[rep]$
13:    else
14:       $lab[p] \leftarrow$ 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: $i \leftarrow 1$ ; $rep \leftarrow (0, 0)$        (* some value such that $rep \neq$ W *)
22: while ($i \leq$ CON) and ($rep \neq$ W) do          (* CON indicates the connectivity *)
23:    if ($sln[p, i] \neq p$) and ($sln[p, i] \neq$ W) then
24:       $sln[p, i] \leftarrow$ Resolve ($sln[p, i]$)
25:    end if
26:    if $i = 1$ then
27:       $rep \leftarrow sln[p, 1]$
28:    else if $sln[p, i] \neq rep$ then
29:       $rep \leftarrow$ W
30:       for $j \leftarrow 1$ to CON do
31:          $sln[p, j] \leftarrow$ W
32:       end for
33:    end if
34:    $i \leftarrow i + 1$
35: end while
36: return $rep$
```

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.

5.1. General considerations

5.1.1. Parallel computer systems

A standard classification of parallel computer systems into four types is due to Flynn, see [15,41] 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.

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 [11] 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.

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.

5.1.4. Speed versus scalability

Let N be number of processors used. Define $T(N)$ to be the running time between the moment that the first processor starts and the moment that the last processor finishes. *Speedup* of the parallel algorithm is measured by:

$$S_P(N) = \frac{T_1}{T(N)},$$

where T_1 is the execution time of the fastest serial algorithm on one processor. Often, T_1 is replaced by the time needed to execute the algorithm, which formed the starting point for parallelization, on one processor; then one speaks about *relative speedup*. *Efficiency* is defined as

$$E(N) = S_P(N)/N.$$

A quality measure for the efficiency of a parallel algorithm is how close the efficiency is to unity, i.e., how well the speedup curve approximates the linear function $S_P(N) = N$. Speedup depends critically upon the amount of sequential computation. If f is the fraction of such sequential operations, then *Amdahl's law* states that the maximum speedup achievable obeys [41]:

$$S_P(N) \leq \frac{1}{f + \frac{1-f}{N}}.$$

This implies that a small number of sequential operations can drastically limit the achievable speedup, since $S_P(N) \leq 1/f$, no matter how many processors are used.

Usually, speedup is an increasing function of the problem size, since overhead costs, such as creating processes, input/output and process synchronization are constant or increase slower than grain size. Note that an algorithm can be slow but at the same time have good scaling properties.

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 $D_i, i = 1, \dots, N$ 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.

5.2.1. Hill climbing by ordered queues

Parallellization of the watershed transform by ordered queues is discussed in [27]. 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 [32] 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 [27] 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 [27, 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) [38]. (An initial implementation on a loosely coupled cluster of workstations using the PVM (Parallel Virtual Machine) package [40] resulted in marginal speedups with efficiency deteriorating quickly as the number of processes was increased [32]). The second scheme was implemented on a Cray T3D MIMD distributed memory architecture with 256 nodes using MPI (Message Passing Interface) [16]. 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.

5.2.2. Hill climbing and rainfalling after lower completion

Hill climbing, with lower completion as preprocessing, was considered by Moga et al. [27, 28], effectively using, but not explicitly introducing, the local condition of Definition 3.7. In addition, the *rainfalling* algorithm was studied, see Section 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 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 [38] and on Cray T3D under MPI. Speedup curves are rather similar for rainfalling and hillclimbing, with rainfalling having shorter running times but somewhat 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 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.

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. [6] using the local condition of Definition 3.7, and by Moga et al. [27, 30]. We first describe the former approach [6].

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 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 [1, 13, 45].

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

$$D_i^{temp} = \{p \in D_i | \Gamma(p) \cap (D_i^+ \setminus D_i) \neq \emptyset\}. \tag{5.1}$$

3. Produce a watershed segmentation consistent with Definition 3.7, 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 3.7.
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 4.2.1.

As an example, consider Fig. 12. Figure 12(b) shows a watershed segmentation of the image in Fig. 12(a) consistent with Definition 3.7 (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 3.7) is obtained.

Of course, the real problem is to treat images with plateaus. To do this, the following procedure is proposed in [6]. 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 3.7 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 (5.1), $\Gamma(p)$ is replaced by $\Gamma'(p)$, the set D_i^{temp} as defined in (5.1) 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 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 3.7 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 [6], 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.

0	1	2	1	**0**
1	2	3	2	1
5	3	7	4	3
4	5	8	6	2
3	4	6	2	1
0	2	3	4	**0**

(a)

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

(b)

A	1	2	1	B
1	2	3	2	1
5	3	7	4	3
4	E	8	6	2
3	4	6	2	1
C	2	3	4	D

(c)

A	A	A	B	B
A	A	A	B	B
A	A	A	B	B
C	E	E	D	D
C	C	D	D	D
C	C	D	D	D

(d)

Figure 12. Watershed according to Definition 3.7. (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.

A similar approach was used by Moga et al. [27,30]. The difference with the approach in [6] 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 [6] on a Parsytec Supercluster 128 under PIPS and in [30] 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 5.2.2. Also, the data dependence is less severe, although it is still present due to the plateau correction step.

5.2.4. Parallel watershed transform based on sequential scanning

Finally we mention a watershed algorithm based on sequential scannings [27,33] (see Section 4.2). Although the sequential algorithm is very slow, the method has good scaling properties 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 [33], and under MPI on a Cray T3D [27, 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.

5.2.5. Conclusions

Summarizing the performance results for the various parallel implementations discussed in Section 5.2.1-Section 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.

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 5.1.

5.3.1. Components graph algorithm

A parallelization of the components graph algorithm (cf. Section 4.1.2) is proposed in [21]. 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.

5.3.2. Topographical distance algorithm by ordered queue

The first step of the algorithm is detecting local minima, see Section 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 3.2.2). Computation of the watershed transform on the graph is based on Algorithm 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.

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 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.

5.3.4. Results and conclusions.

Timing results on a Cray-J932 shared memory computer with 16 processors for the algorithm discussed in Section 5.3.2 are reported in [22], where only static mapping was used. At 16 processors, efficiency for the complete watershed computation ranges from 20% for an image with only a few minima to about 60% for an image with a moderate amount of minima. For images with very many minima efficiency deteriorates. 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. Work by the present authors on the approach of Section 5.3.3 is in progress.

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.

References

[1] 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.

[2] Berge, C. *Théorie des Graphes et ses Applications.* Dunod, Paris, 1958.

[3] 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.

[4] 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).

[5] 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.

[6] 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.

[7] 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.

[8] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms.* MIT Press, 1990.

[9] 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.

[10] Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische Mathematik 1* (1959), 269–271.

[11] Dijkstra, E. W. Co-operating sequential processes. In *Programming Languages*, F. Genuys, Ed. Academic Press, New York, 1968, pp. 43–112.

[12] 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.

[13] Embrechts, H., Roose, D., and Wambacq, P. Component labelling on a mimd multiprocessor. *Comp. Vis. Graph. Im. Proc. 75*, 2 (1993), 155–165.

[14] Fiorio, C., and Gustedt, J. Two linear time union-find strategies for image processing. *Theoretical Computer Science A 154*, 2 (Feb. 1996), 165–181.

[15] Foster, I. *Designing and Building Parallel Programs.* Addison Wesley, Reading, MA, 1994.

[16] Gropp, W., Lusk, E., and Skjellum, A. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* MIT Press, Cambridge, MA, 1995.

[17] Guillemin, V., and Pollack, A. *Differential Topology.* Prentice-Hall, Englewood Cliffs, NJ, 1974.

[18] Haralick, R. M., and Shapiro, L. G. Survey : image segmentation techniques. *Comp. Vis. Graph. Im. Proc. 29* (1985), 100–132.

[19] 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.

[20] Lantuéjoul, C. *La squelettisation et son application aux mesures topologiques des mosaïques polycristallines.* PhD thesis, Ecole des Mines, Paris, 1978.

[21] 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.

[22] 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.

[23] 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.

[24] Meyer, F. Un algorithme optimal de ligne de partage des eaux. In *Proceedings* 8*th Congress AFCET, Lyon-Villeurbane, France* (1992), vol. 2, pp. 847–859.

[25] Meyer, F. Topographic distance and watershed lines. *Signal Processing 38* (1994), 113–125.

[26] Meyer, F., and Beucher, S. Morphological segmentation. *J. Visual Commun. and Image Repres. 1*, 1 (1990), 21–45.

[27] Moga, A. *Parallel watershed algorithms for image segmentation.* PhD thesis, Tampere University of Technology, Tampere, Finland, Feb. 1997.

[28] Moga, A. N., Cramariuc, B., and Gabbouj, M. Parallel watershed transformation algorithms for image segmentation. *Parallel Computing 24* (1998), 1981–2001.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] Nackman, L. R. Two-dimensional critical point configuration graphs. *IEEE Trans. Patt. Anal. Mach. Intell. 6*, 4 (1984), 442–450.

[36] Najman, L., and Schmitt, M. Watershed of a continuous function. *Signal Processing 38* (1994), 99–112.

[37] 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.

[38] 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.

[39] 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.

[40] *PVM: Parallel Virtual Machine, a user's guide and tutorial for networked parallel computing*, 1994.

[41] Quinn, M. *Parallel Computing. Theory and Practice.* McGraw-Hill, New York, NY, 1994.

[42] 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.

[43] Rosenfeld, A., and Pfaltz, J. Distance functions on digital pictures. *Pattern Recognition 1* (1968), 33–61.

[44] Rosenfeld, A., and Pfaltz, J. L. Sequential operations in digital picture processing. *J. Ass. Comp. Mach. 13* (1966), 471–494.

[45] Samet, H. Connected component labeling using quadtrees. *J. Ass. Comp. Mach. 28*, 3 (1981), 487–501.

[46] Serra, J. *Image Analysis and Mathematical Morphology.* Academic Press, New York, 1982.

[47] Tarjan, R. E. *Data Structures and Network Algorithms.* SIAM, 1983.

[48] Tarjan, R. E., and van Leeuwen, J. Worst-case analysis of set union algorithms. *J. Ass. Comp. Mach. 31*, 2 (1984), 245–281.

[49] 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.

[50] Viero, T. *Algorithms for image sequence filtering, coding and image segmentation.* PhD thesis, Tampere University of Technology, Tampere, Finland, Jan. 1996.

[51] 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.

[52] 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.