Parallel Topological Watershed

Joël van Neerbos

August 20, 2010

Contents

Chapter 1

Introduction

In computing science, *computer vision* is a field of study that is concerned with automatically extracting information from images. For example, a prominent application field is medical computer vision, where an artificial system may extract information from a medical images, such as microscopy images or X-ray images. The extracted information can then be used to aid or speed up the work of the medical staff.
A typical process found in many computer vision systems is *image segmentation.* This process partitions an image into multiple regions, where each region contains pixels that share a certain visual characteristic.
An important step in many image segmentation methods is the *watershed transform.* This report introduces a parallel algorithm for the *topological watershed transform*, which is a variant of the watershed transform.

First, Section 1.1 will give an introduction to the watershed transformation. Section 1.2 will introduce to topological watershed, and Section 1.3 the parallelization of the topological watershed.

Chapter 2 will give some definitions of concepts used throughout this report. Chapter 3 will describe how the existing sequential implementation works and Chapter 4 will describe the new parallel implementation. Chapter 5 will show some test results of the parallel implementation, and finally Chapter 6 contains some conclusions and discussion on the results.

1.1 Watershed transformation

The watershed transformation is a tool for segmenting grayscale images, introduced by S. Beucher and C. Lantuéjoul [3]. It can be used to segment an image into regions with similar gray values. To achieve this, the watershed transformation is usually applied to the gradient of the image. The gradient image will have higher gray values where different regions from the original image meet. The watershed transformation puts watershed lines in these lighter areas. The watershed lines can then be used in the original image as the borders between the segments. This concept is illustrated in Figure 1.1.

When the watershed transformation is applied to the gradient of an image and not the image itself, like in the example in Figure 1.1, the image itself is

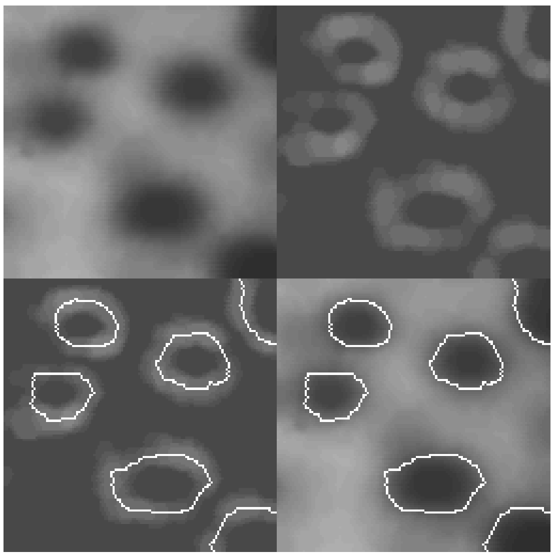


Figure 1.1: The watershed transformation. From top to bottom and from right to left: a) an image b) gradient of the image c) watershed transform applied to the gradient (white shows watershed lines) d) watershed result shown on the original image, where the watershed lines mark the borders of different regions in the image

actually irrelevant to the computation of the watershed. Because this report is about the computation of the watershed transform (or, rather, the *topological* watershed transform), we will only consider the image on which the process is applied, usually being a gradient image. Any original images will be omitted in the remainder of this report.

1.1.1 The flooding paradigm

To explain what the watershed transform does exactly, the *flooding paradigm* (L. Vincent and P. Soille, [13]) is often used. In this paradigm, the input image is considered as a *height map* of a topographical landscape. In a height map, the gray value of each pixel represents the height of a point in the landscape, a higher gray value representing a higher point. For example, Figure 1.3(a) shows the landscape that emerges when Figure 1.2(a) is used as a height map.

In the flooding paradigm, the topographical landscape is then slowly flooded, as if the landscape was punctured in the local minima and is lowered into a body of water. This causes the lowest points to be submerged first, with the water level rising slowly.
As the water keeps on rising, more and more points become submerged and the different bodies of water (called *basins*) become larger and larger. At some point, different existing bodies of water will meet. Instead of merging the bodies of water into one large basin, a dam is built between them. The water keeps on rising, and every time different basins meet a dam is built. When the landscape is completely submerged, only basins and dams remain.
If we convert these dams and basins back to an image, we get the watershed

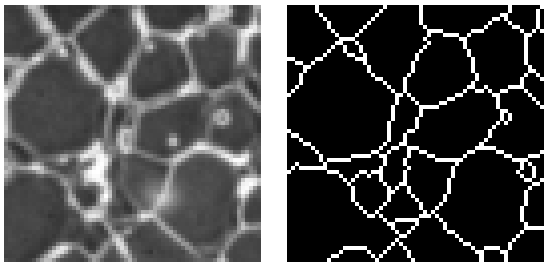

(a) (b)

Figure 1.2: An image (a) and its watershed (b).

transform of the input image, with the dams representing the watershed lines and the bodies of water representing the watershed basins. The flooding process is illustrated in Figure 1.3, which shows the flooding paradigm for the image from Figure 1.2(a), eventually resulting in the watershed of that image, as shown in Figure 1.2(b).

1.2 The topological watershed transform

The watershed transform as described in the previous section has a number of drawbacks. One these drawbacks is that oversegmentation may occur, where regions that should have been one segment are segmented themselves. In general, we only want to keep the watershed lines that correspond to the most significant contours of the original image.
An approach to achieve this was given by J. Angulo and D. Jeulin [1], who proposed a stochastic watershed segmentation. In this approach, the landscape is not punctured at the local minima, but at random locations. The water flowing from a puncture may flood several local minima before meeting another body of water, causing the dams to be built at different locations. This process is repeated several times with punctures at different random locations each time. Some watershed lines will appear in most iterations, while other watershed lines only appear in some of them. Only the watershed lines that appear most often are kept for the final result. This results in watershed lines that lie mostly on significant contours. However, this approach needs to compute the watershed several times, causing the amount of work to increase linearly with the number of iterations.

The approach used in this report is the topological watershed [2], [4]. In the topological watershed, some of the grayscale information from the original image is preserved, which may be useful for further processing, such as reconnection of corrupted contours. Also, this grayscale information can be used to determine the significance of watershed lines. For example, because both basins and watershed lines are assigned gray values, two basins that are separated by a watershed line that has a similar gray value may be merged together, removing the watershed line.
Specifically, the topological watershed preserves the *pass values* [7] between all minima: the heighest point of the lowest path between two minima. This also has the consequence that the connectivity of each lower cross-section is pre-

served. The concepts of connectivity and lower cross-sections will be further explained in Chapter 2.

The pass value of two points in the image is also called the *separation* of the points. The points p and q are said to be *k-separated* if the following conditions apply:

- There exists a path from p to q with maximum value $k - 1$
- There exists no path from p to q with a maximum value lower than $k - 1$
- Both p and q have a value lower than $k - 1$

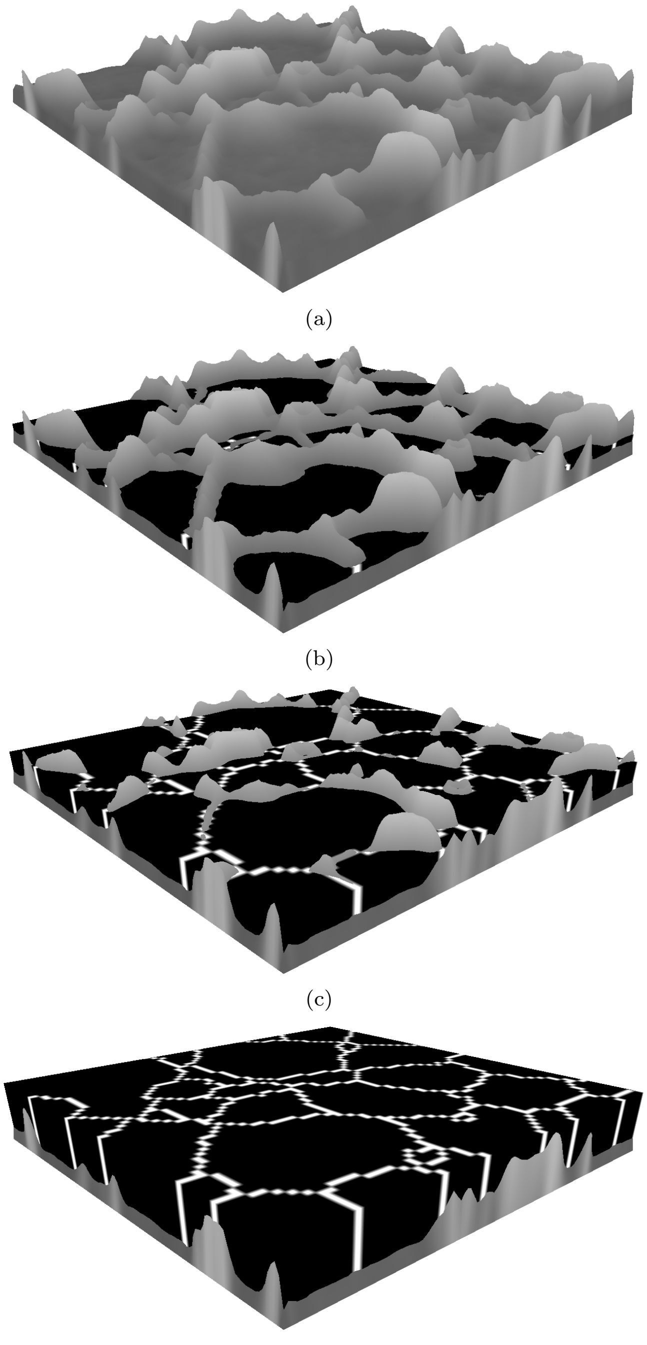
A path that satisfies the first two conditions for some k is called a *lowest path* in this report. If a lowest path from p to q contains no value that is higher than both p and q, then p and q are not separated, but *linked.* Separation is illustrated in Figure 1.2.

The topological watershed is similar to the normal watershed; the locations of the basins and watershed lines are the same. However, the result of the topological watershed is not a binary image, but a grayscale image. The gray values of the pixels are defined as follows:
All pixels in a basin have the same gray value, namely the value of the minimum from the input image that is contained within the basin.
The values of the pixels on the watershed lines are as low as possible, without changing the separation relations between the basins. If two pixels from different basins were k-separated in the input image, they should still be k-separated in the topological watershed of the image.
Both the watershed and the topological watershed of the image from Figure 1.4(a) are shown in Figure 1.5.

1.3 Parallelization of the topological watershed transform

This report proposes a parallel algorithm for the topological watershed. A sequential algorithm has already been implemented by Couprie et al. [5], on which the parallel algorithm will be based.
Also, J.B.T.M. Roerdink and A. Meijster [9] already proposed parallelization strategies for several related watershed transformation variants.

The existing sequential algorithm uses only one processor to compute the topological watershed of the input image. The parallel algorithm will use several processors at the same time, distributing the work over these processors, which should speed up the computation. Ideally, the total running time of the parallel algorithm will be $t_p = t_s/n_p$, where t_p is the execution time of the parallel algorithm, t_s is the execution time of the sequential algorithm and n_p is the number of processors used. In this case the *speedup*, defined as t_s/t_p will be equal to the number of processors. In practice however, this optimal speedup is rarely achieved.

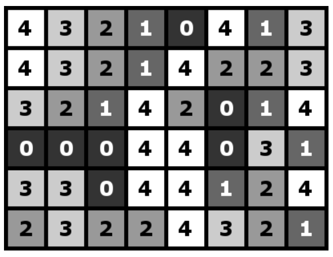

(a)

(b)

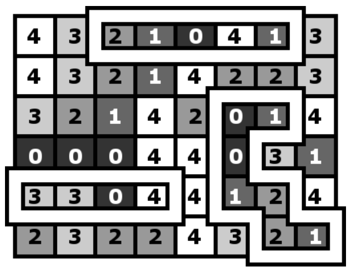
(c)

(d)

Figure 1.3: The flooding paradigm. (a) shows the landscape that emerges when Figure 1.2(a) is used as a height map. When this landscape is flooded, the lower areas are filled up first, forming basins, as shown in (b). When two different basins meet, a dam is built between them, as illustrated in (c). Finally, (d) shows the fully submerged landscape, where only basins and dams are left at the surface. Viewed from above, this situation corresponds to the watershed of the image that was used as a height map, as shown in Figure 1.2(b).

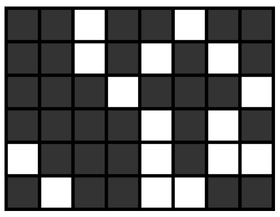


(a) A digital grayscale image

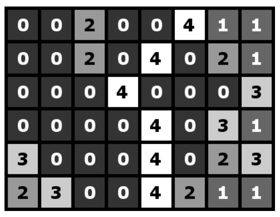


(b) Lowest paths between three pairs of pixels.

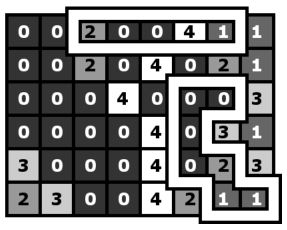
Figure 1.4: Separation. The top path in (b) connects two 5-separated pixels. The left path connects pixels that are not separated (but linked). The right path connects two 3-separated pixels.



(a) Watershed of Figure 1.4(a)



(b) Topological watershed of Figure 1.4(a)



(c) Lowest paths between the separated pairs of pixels from Figure 1.4(b)

Figure 1.5: Watershed and topological watershed. Figure (c) shows that the separated pixel pairs from Figure 1.4(b) are still respectively 5-separated and 3-separated. The third pair is not shown as it was not separated.

Chapter 2

Definitions

This chapter will introduce some important concepts that we will need to compute the topological watershed. Section 2.1 is about connectivity. Section 2.2 will introduce lower cross-sections, that will be used in Section 2.3 to give a definition of W-destructibility. Section 2.4 will introduce the component tree and the component map, and finally Section 2.5 will introduce the lowest common ancestor.

2.1 Connectivity

The *connectivity* defines for each pixel which pixels in its neighbourhood are its *neighbours*. The examples in this report will use 4-connectivity, meaning that only its 4 nearest pixels are the neighbours of a pixel. The supported connectivities for the algorithms described in this report are 4 and 8 for 2D images and 6, 18 and 26 for 3D volumes. These connectivities are illustrated in Figure 2.1. In the case of border pixels, some neighbours will be absent.

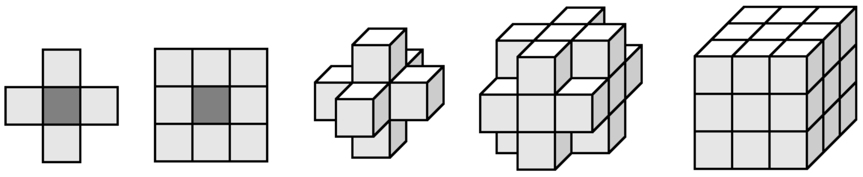


Figure 2.1: Different connectivities: 4 and 8 (2D) and 6, 18 and 26 (3D)

2.2 Lower cross-sections

A *lower cross-section* of an image is a binary image that shows which pixels in the original image have a value lower than some threshold k. The pixels with a value lower than k result in a value of 0 in the lower cross-section, while the pixels with a value equal or higher than k result in a 1. An example is shown in Figure 2.2.
Note that lowering the value of a pixel by 1 would add the pixel to the lower

cross-section for k equal to the old value of the pixel. For example, lowering a pixel with value 4 to value 3 would add a 1 on the location of that pixel in the lower cross-section for $k = 4$.

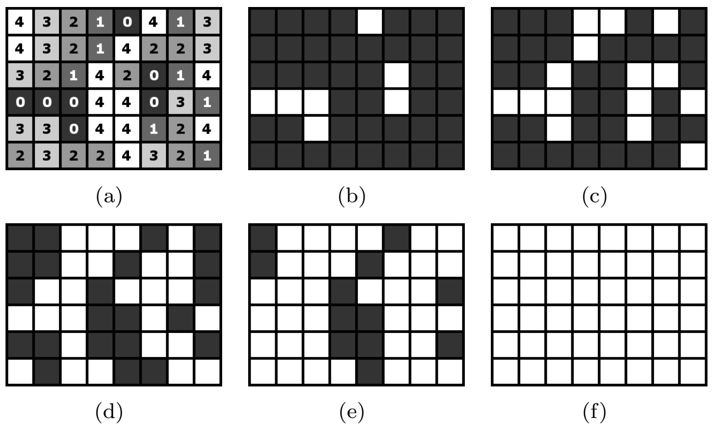


Figure 2.2: An image (a) and its lower cross-sections for $k = 1$ (b), 2 (c), 3 (d), 4 (e) and 5 (f). White pixels have a value of 1, dark pixels have value 0.

2.3 W-destructibility

In the lower cross-sections, we will call a connected region of pixels with value 1 a *connected component*. For example, in Figure 2.3 we have two connected components: A and B.
As described in Section 2.2, lowering the value of a pixel by one will add the pixel to a lower cross-section. Lowering certain pixels can merge different connected components into one connected component. This is illustrated in Figure 2.3.
If multiple connected components are merged, the number of connected components is decreased. However, lowering a pixel can also cause a new component to be formed, causing the number of connected components to be increased.
If a pixel can be lowered without changing the number of connected components at all in any lower cross-section, the pixel is called *W-destructible*.

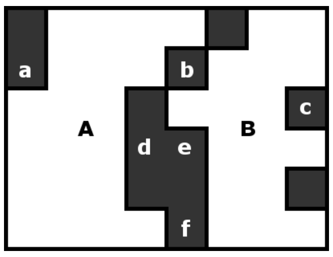


Figure 2.3: A lower cross-section. Pixels a and c can be added to the lower cross-section without merging the connected components A and B, while pixels b and f can not. Either pixel d or e can be added as well, but not both.

After iteratively lowering W-destructible pixels in an image, a *W-thinning* of the image is obtained. A W-thinning of an image that has no W-destructible pixels left, is a topological watershed of the image.

2.4 Component tree & component map

Component trees [6] are based on *Max-trees*, introduced by P. Salembier et al. [10]. The component tree of an image is an abstraction of the landscape that is defined by the image if it is regarded as a heightmap. The root of the component tree represents the entire landscape at its lowest level. At higher levels, the landscape may consist of separate landmasses, which are then represented as different branches in the component tree. Each of these branches may contain its own branches, if there are several smaller landmasses on top of the landmass it represents. The leaves of the tree represent the local maxima or peaks of the landscape. Figure 2.4(g) shows the component tree of Figure 2.4(a). The components of the component tree are shown in Figures 2.4(b) to 2.4(f).
However, for the computation of the topological watershed, we don't need to look at the landscape itself, but at the bodies of water that will form on top of the landscape when it is flooded. Fortunately, we can easily resolve this issue by just computing the component tree of the inverted image (also called the *min-tree* of the image), which gives us the component tree of the body of water that covers the entire landscape when it is completely flooded. Figure 2.4(a) shows the inverse of the image that we used before. Also note that the components of the inverse image shown in Figures 2.4(b) to 2.4(f) are equal to the connected components in the lower cross-sections of the original image (shown in Figure 2.2).
The components of the component tree are stored as a *component map.* The component map stores for each pixel the highest component that contains the pixel. The other components that contain the pixel are the ancestors of the stored component in the component tree. An example of a component map is shown in Figure 2.4(h).

Because the component tree is computed on the inverted image, the levels of the components naturally correspond to the inverted images values. However, we want to use the obtained component tree in combination with the original non-inverted image, so we will invert the component levels to make them correspond to the original image values.
The inverted image and its corresponding component levels only appear during the computation of the component tree and component map, they will not be used in any other step in the computation of the topological watershed.
The levels of the components in the component tree from Figure 2.4(g) are shown in Figure 2.5.

2.5 Lowest Common Ancestor

The *Lowest Common Ancestor* (or LCA) of a set of nodes within a tree is the lowest node in the tree that either has all those nodes as its descendants, or is one of the nodes itself and has all the other nodes as its descendants. Although

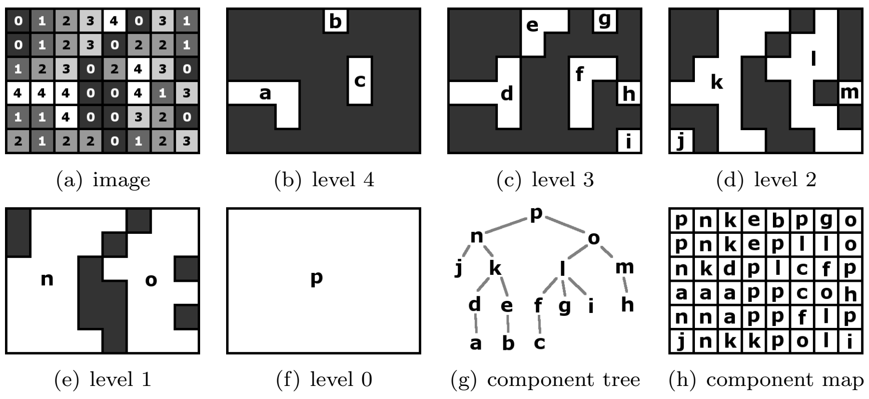


Figure 2.4: An image and its components, component tree and component map. Shown are: (a) the image, which is the inverse of the image shown in Figure 2.2; (b) to (f) the components of (a) on heightlevels 4 to 0; (g) the component tree of (a); (h) the component map of (a).

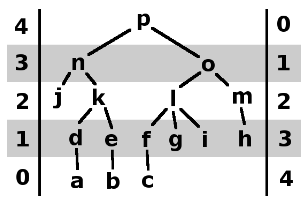


Figure 2.5: The component tree of Figure 2.4(g) with the levels of the components indicated. The values that we will use are shown on the left, the values corresponding to the inverted image are shown on the right.

the LCA is defined for an arbitrary number of nodes, we will only look at the *Binary Lowest Common Ancestor* (or BLCA), which is the LCA of exactly two nodes. Some examples of lowest common ancestors are shown in Figure 2.6

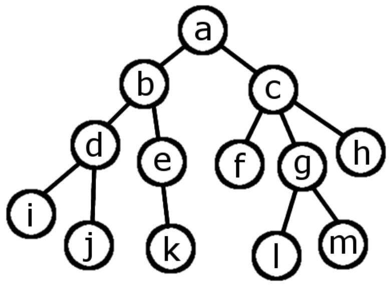


Figure 2.6: A tree. The LCA of i and j in this tree is d, the LCA of b and k is b and the LCA of f and m is c.

Chapter 3

Sequential implementation

In this chapter we will describe the sequential implementation as proposed by [5]. First, we will describe how the sequential algorithm for the topological watershed is implemented, in Section 3.1. This algorithm makes use of the component tree and component map, as well as the BLCA algorithm. Section 3.2 describes how the component tree and component map are computed, and Section 3.3 describes the BLCA implementation.

3.1 Sequential topological watershed

The algorithm that computes the actual topological watershed of an image is described in Section 3.1.3. This algorithm transforms the input image into its topological watershed by lowering the values of W-destructible pixels. To be able to do this, it needs to detect whether or not a pixel is W-destructible. This detection is performed by the `W-Destructible` function described in Section 3.1.2. This function in its turn needs the function `HighestFork` to operate, which we will describe first in Section 3.1.1.

3.1.1 `HighestFork` function

The function `HighestFork` takes a component tree $\mathcal{C}$ and a set of components V from $\mathcal{C}$ as its input, and returns the highest component in the input tree that has at least two children that are either one of the input components or an ancestor of one of the input components. If no such component exists, $\emptyset$ is returned. Figure 3.1 shows some examples of highest forks.
The pseudocode of the algorithm is given below.

```
Function HighestFork (Input $\mathcal{C}, V$)
01. $c_1 \leftarrow \min(V)$; let $c_2 ... c_n$ be the other components from $V$
02. $c \leftarrow c_1$
03. For $i$ From 2 To $n$ Do
04.     $c_{\text{blca}} \leftarrow$ BLCA$(\mathcal{C}, c, c_i)$
05.     If $c_{\text{blca}} \neq c_i$ Then $c \leftarrow c_{\text{blca}}$
06. If $c = c_1$ Then Return $\emptyset$ Else Return $c$
```

On line `01`, the lowest input component is obtained and stored as the intermediate result in line `02`. The loop in lines `03` to `05` loops through the other

components and computes for each one the lowest common ancestor with the intermediate result. If the LCA is not equal to the component itself, it is stored as the new intermediate result.
Finally, if the intermediate result is still equal to the lowest input component, $\emptyset$ is returned. Otherwise, the intermediate result is returned as the final result.

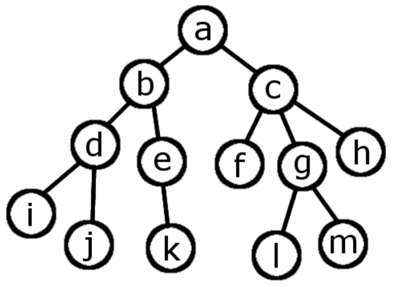


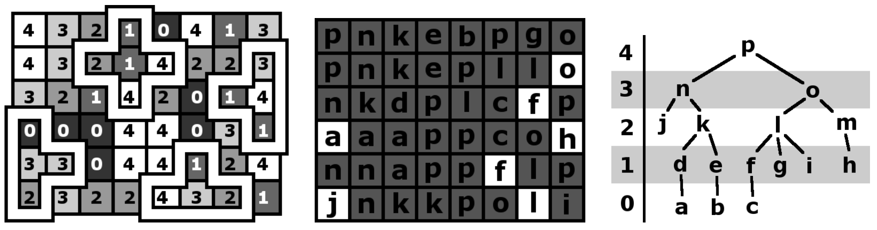
Figure 3.1: A component tree. The highest fork for $\{i, j, k\}$ is b, the highest fork for $\{c, f, g\}$ is c and the highest fork for $\{a, c, l, m\}$ is g. $\{b, d, j\}$ has no highest fork.

3.1.2 `W-Destructible` function

The function `W-Destructible` determines whether a pixel is W-destructible or not. If the pixel is W-destructible, the function returns the component to which the pixel can be added. If the pixel is not W-destructible, it returns $\emptyset$.
As its input, the function needs a map F that maps each pixel to its value, the pixel p that is to be tested for W-destructibility, the component tree $\mathcal{C}(\overline{F})$ of the inverse image and the component map Ψ, that maps each pixel to a component in $\mathcal{C}(\overline{F})$.

```
Function W-Destructible (Input $F, p, \mathcal{C}(\overline{F}), \Psi$)
01. $V \leftarrow$ the components pointed at by $\Psi$ for all lower neighbours of $p$
02. If $V = \emptyset$ Then Return $\emptyset$
03. $c \leftarrow$ HighestFork($\mathcal{C}(\overline{F})$, $V$)
04. If $c = \emptyset$ Then Return $\min(V)$
05. If level of $c < F(p)$ Then Return $c$ Else Return $\emptyset$
```

In line `01` of the algorithm, the components of all neighbouring pixels that have a value strictly lower than the value of p, are obtained using the component map. If there are no lower neighbours, and therefore no corresponding components, $\emptyset$ is returned in line `02`, signifying that the pixel is not W-destructible.
Otherwise, the highest fork of the obtained components in the component tree is computed in line `03`. If they have no highest fork, this means that p is not a watershed pixel and that it can be lowered to the level of its lowest neighbour, so the component of the lowest neighbour is returned (line `04`).
If there is a highest fork, p is a watershed pixel. In this case, the highest fork is returned if its level is lower than the value of the pixel, or $\emptyset$ is returned if it is not.
The four different cases of this algorithm are illustrated in Figure 3.2.



(a) An image with four pixels and their neighbours marked

(b) The lower neighbours from (a) marked in the component map of the component tree in (c)

(c) The component tree of the inverse of the image in (a), with the levels inverted back

Figure 3.2: The four cases of the `W-Destructible` algorithm. The top example shows a pixel that has no lower neighbours and is therefore not W-destructible. The lower neighbours of the bottom example map to components f and l, that have no highest fork in the component tree, so the lowest component is returned, in this case component f. The right example has lower neighbours that map to components f, h and o. Their highest fork is component o, of which the level is lower than the level of the pixel, so component o is returned. Finally, the left example has lower neighbours that map to components a and j, that have highest fork n. However, the level of component n is not lower than the level of the pixel itself, so the left example is not W-destructible.

3.1.3 `TopologicalWatershed` procedure

Now that we have the `W-Destructible` algorithm, we could just loop through all pixels, lowering them if they are W-destructible. However, since the W-destructibility of a pixel is dependent on the values of its neighbours, a pixel that is not W-destructible may become W-destructible if one or multiple of its neighbours are lowered. Also, a pixel that has already been lowered may become W-destructible again if one of its neighbours is lowered afterwards. To reduce the time-complexity of the algorithm, we want each pixel to be lowered at most once. This can be achieved by giving the highest priority to W-destructible pixels that may be lowered down to the lowest possible value.
The procedure `TopologicalWatershed` implements this idea, using priority queues to determine which W-destructible pixel should be lowered next. The procedure takes the map F that holds the pixel values of the image, the component tree $\mathcal{C}(\overline{F})$ of the inverse image and the corresponding component map Ψ as its input, and it produces a modified version of the map F, that then holds the pixel values of the topological watershed of the image.

```
Procedure TopologicalWatershed (Input $F, \mathcal{C}(\overline{F}), \Psi$; Output $F$)
01. For $k$ From $k_{\min}$ To $k_{\max} - 1$ Do $L_k \leftarrow \emptyset$
02. For All pixels $p$ Do
03.     $c \leftarrow$ W-Destructible($F, p, \mathcal{C}(\overline{F}), \Psi$)
04.     If $c \neq \emptyset$ Then
05.         $i \leftarrow$ level of $c$; $L_i \leftarrow L_i \cup \{p\}$
06.         $K(p) \leftarrow i; H(p) \leftarrow$ pointer to $c$
```

```
07. For $k$ From $k_{\min}$ To $k_{\max} - 1$ Do
08.     While $\exists p \in L_k$ Do
09.         $L_k = L_k \backslash \{p\}$
10.         If $K(p) = k$ Then
11.             $F(p) \leftarrow k; \Psi(p) \leftarrow H(p)$
12.             For All $q \in$ neighbourset of p, $k < F(q)$ Do
13.                 $c \leftarrow$ W-Destructible$(F, q, \mathcal{C}(\overline{F}), \Psi)$
14.                 If $c = \emptyset$ Then $K(q) \leftarrow \infty$
15.                 Else
16.                     $i \leftarrow$ level of $c$
17.                     If $K(q) \neq i$ Then
18.                         $L_i \leftarrow L_i \cup \{q\}; K(q) \leftarrow i$
19.                         $H(q) \leftarrow$ pointer to $c$
```

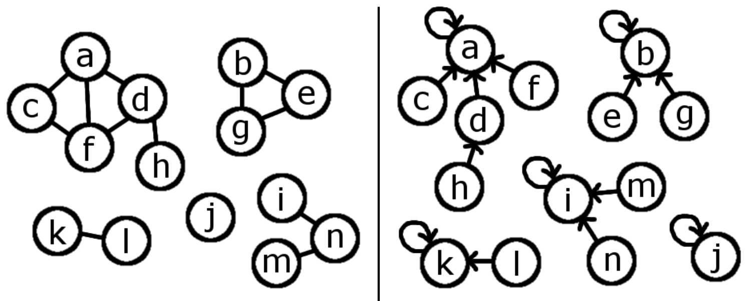
First, an empty priority queue L, which uses the graylevels as the priorities, is created in line 01. Then, the algorithm loops through all pixels and runs the function W-Destructible on each one. If the pixel turns out to be W-destructible, it is added to the priority queue with its priority set to the level to which the pixel may be lowered. Also, this new level is stored in the map K and a pointer to the component to which the pixel may be added is stored in the map H. The pixel itself is not lowered yet.
Next, the algorithm runs through the priority queue, starting with the priority that corresponds with the lowest graylevel, in the loop that starts on line 07. A pixel is removed from the queue in line 09, and the algorithm checks whether the pixel should still be lowered to the level of the current priority, using the map K. If the map K holds a different value for the pixel than the level of the current priority, this means that the situation of the pixel has changed since the beginning of the algorithm, and its value should no longer be changed to the graylevel that corresponds to the current priority. No actions need to be performed in that case. If the map K still holds the value of the current priority, the pixel is updated with its new value in the map F, and the component map is updated accordingly, using the pointer stored in the map H (line 11).
Now the image has changed, the W-destructibility of each possibly affected pixel (i.e. the neighbours of the changed pixel) is recomputed. If a neighbour is not W-destructible, this is stored in the map K on line 14, as it may have been W-destructible before. Otherwise, the algorithm checks whether the neighbour is already to be lowered to the correct level using the map K. If it is not, the maps K and H are given their new values and the pixel is added to the priority queue with its priority equal to the level to which it may be lowered.

3.2 The component tree

The sequential component tree algorithm described here is the algorithm proposed by L. Najman and M. Couprie in [8], which runs in quasi-linear time. This algorithm is based on the Union-find algorithm proposed by R.E. Tarjan [12], that will be described first.

3.2.1 Tarjan's Union-find

The *Union-find* algorithm is used to label the connected components in a graph, making it possible for a vertex in the graph to quickly determine to which connected component it belongs. Each connected component in the graph is represented by one of its vertices, called the *canonical element* of the component. The vertices in a connected component are stored as a tree, where each vertex has a parent. The canonical element is the root of the tree, being its own parent. An example input graph and the corresponding output trees resulting from the Union-find algorithm are shown in Figure 3.3.



(a) A graph

(b) Result of Tarjan's Union-find on the graph of (a)

Figure 3.3: Input and the corresponding output of Tarjan's Union-find algorithm. Figure (a) shows a graph with 5 separate connected components. Figure (b) shows the vertices from (a) organized in trees, where the root or canonical element of each tree functions as the label of its vertices. The connected components from (a) now have labels a, b, i, j and k.

The first step in computing the result of the Union-find algorithm is to create a single-vertex tree out of every vertex from the input graph. This is done using the procedure `MakeSet`. Each vertex has a parent, which is stored in the map `Par`. The map `Rnk` stores the rank of each vertex, although only the rank of vertices that are canonical nodes is used. Ranks will be discussed later in this section.
The procedure `MakeSet` is shown below. The input of the procedure is the vertex that is to be represented as a tree.

```
Procedure MakeSet (Input $x$)
01. Par($x$) ← $x$; Rnk($x$) ← 0
```

The parent of the vertex is set to the vertex itself, making the vertex a root and therefore a canonical node. The rank is initially set to zero.

Now that each vertex is represented as a tree, we want to be able to merge trees that contain vertices of the same connected component. Merging two trees X and Y is done by making the root of Y the parent of the root of X. This means that each vertex from X is now part of Y, with the depth of each vertex increased by one. The depth of the vertices in Y does not change, so the new depth of Y is equal to the maximum of the depth of X plus one and the old depth of Y. Because we want to keep the depth of our trees as small as possible for speed reasons, we want X to be the input tree with the smallest depth.
In the implementation, we determine which input tree to take for X and which for Y based on the *rank* of each tree. The rank of a tree is initially equal to the depth of the tree, but we will see that the depth of individual vertices may decrease during run-time, so the depth of the tree may actually be smaller than its rank.
The merging of two trees is implemented with the function `Link`. Its input consists of two vertices x and y, being the roots of the two trees. Vertex y is returned at the end, which is then the root of the merged tree.

```
Function Link (Input $x$, $y$)
01. If Rnk($x$) $>$ Rnk($y$) Then exchange($x$,$y$)
02. If Rnk($x$) $=$ Rnk($y$) Then Rnk($y$) $\leftarrow$ Rnk($y$) + 1
03. Par($x$) $\leftarrow y$
04. Return $y$
```

On line `01`, the algorithm makes sure that the root of the tree with the smallest rank is stored in the x variable.
Assuming that the ranks are still equal to the depths of the trees, the depth of the tree of y can only increase if the two trees have equal depths, so the rank of y is increased in line `02` if they have the same ranks.
Line `03` sets the parent of x to y, and y is returned on line `04`.
The function `Link` is illustrated in Figure 3.4.

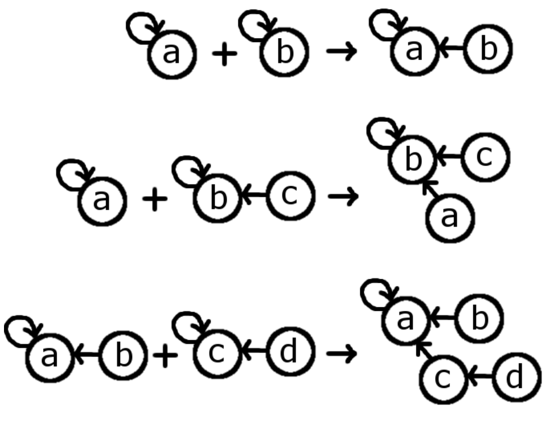


Figure 3.4: Some examples of the `Link` function. Top: two single-vertex trees with rank 0 are merged, the result has rank 1. Middle: a tree with rank 0 is merged with a tree with rank 1, resulting in a tree with rank 1. Bottom: two trees with rank 1 are merged, the result has rank 2.

For determining to which tree a vertex belongs, we use the function `Find`. The function needs a vertex x for its input, and returns the vertex that is the root of the tree that contains x.

```
Function Find (Input $x$)
01. If $\mathrm{Par}(x) \neq x$ Then $\mathrm{Par}(x) \leftarrow$ Find($\mathrm{Par}(x)$)
02. Return $\mathrm{Par}(x)$
```

The function `Find` is a recursive function, that calls itself unless the input vertex is a root. In that case, it can just return the input vertex (or its parent, since a root is its own parent). In any other case it sets its parent, as well as the parent of all its ancestors, to be the root, and returns this root as the result of the function.
The effect of this function is illustrated in Figure 3.5.

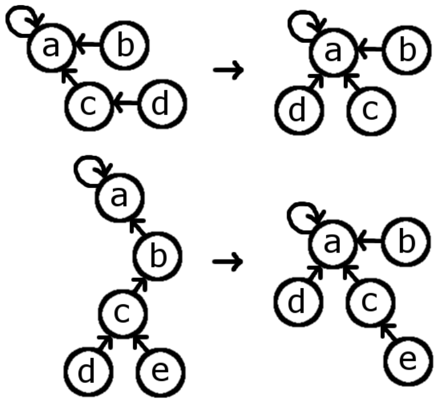


Figure 3.5: Some examples of the `Find` function. Top: the function is called on vertex d, and is recursively called on c and a. The parent of both c and d is set to a, and a is returned. Bottom: the function is called with vertex d as its argument. Recursively, the function is also called with arguments c, b and a. The function sets the parent of the vertices b, c and d to a and vertex a is returned. The parent of vertex e remains unchanged.

With the procedure `MakeSet` and the functions `Link` and `Find`, we can define the `Union-find` algorithm, that creates trees from an input graph, where the root of each tree functions as its label, as was shown in Figure 3.3.
The procedure `Union-find` needs a graph (V, E) as its input, where V is the set of vertices and E the set of edges stored as tuples (p, q), where a tuple $(p, q) \in E$ means that there is an edge between the vertices p and q.

```
Procedure Union-find (Input $(V, E)$)
01. For All $p \in V$ Do MakeSet($p$)
02. For All $p \in V$ Do
03.     $\mathrm{tree}_p \leftarrow$ Find($p$)
04.     For All $q$ with $(p, q) \in E$ Do
05.         $\mathrm{tree}_q \leftarrow$ Find($q$)
06.         If $\mathrm{tree}_p \neq \mathrm{tree}_q$ Then $\mathrm{tree}_p \leftarrow$ Link($\mathrm{tree}_p$, $\mathrm{tree}_q$)
```

The algorithm first creates single-vertex trees for all vertices in V using `MakeSet` on line `01`. For every vertex it then finds the root of the tree it belongs to in line `03` and the root of the tree of each of its neighbours in line `05`. If two vertices are connected with an edge but they are not part of the same tree, then the roots of their trees are linked in line `06`.

When the `Union-find` procedure is done, the label of each vertex can simply be found using the `Find` function with the vertex as its argument.

3.2.2 Computing the component tree

The Union-find algorithm maintains a collection of sets of vertices, in which some sets may be merged during the execution of the algorithm, as described in the previous section. The algorithm from [8] that is described here, uses two such collections: $\mathcal{Q}_{node}$ and $\mathcal{Q}_{tree}$.
$\mathcal{Q}_{node}$ will contain the actual component tree. Each set in $\mathcal{Q}_{node}$ will represent a node in this component tree. To make a set into a node, its children in the component tree need to be stored, as well as its level. The function `MakeNode` is used to initialize a node. It needs the level of the node as its input, and creates a new node with the given level and an empty list of children.

Function MakeNode (**Input** *level*)

```
01. Allocate a new node $n$ with an empty list of children
02. $n$.level ← $level$
03. Return $n$
```

The nodes created with the function `MakeNode` are stored in the array `nodes`. This array is constructed in such a way that the node with index x will belong to the set in $\mathcal{Q}_{node}$ with the same index or label x.
When two sets in $\mathcal{Q}_{node}$ are merged, the corresponding nodes need to be merged as well. This is done with the function `MergeNodes`, that is shown below. The input consists of the two array indices of the nodes that need to be merged.

Function MergeNodes (**Input** *node1*, *node2*)

```
01. $node$ ← $\textbf{Link}_{\text{node}}(node1, node2)$
02. If $node = node2$ Then
03.         Add the list of children of $node1$ to the list of children of $node2$
04. Else
05.         Add the list of children of $node2$ to the list of children of $node1$
06. Return $node$
```

On line `01`, the corresponding sets in $\mathcal{Q}_{node}$ are merged using the function `Link`. The subscript `node` indicates that the function `Link` only operates on the collection $\mathcal{Q}_{node}$, and not on $\mathcal{Q}_{tree}$. Because the indices of the nodes correspond to the indices of the sets, there is no need to differentiate between them.
On line `02`, the function checks if the canonical node of the merged set corresponds to the canonical node of the set of the second input node. If they correspond, the lists of children of the two input nodes are merged and stored in the second input node. Otherwise, the lists are merged and stored in the first input node.
Finally, the merged node is returned in line `06`. There is no need to change the level of the node, because only nodes of the same level will be merged in the `BuildComponentTree` algorithm.

The other collection of sets that the algorithm uses is $\mathcal{Q}_{tree}$. This collection is used to determine which nodes need to be merged.

The sets in $\mathcal{Q}_{tree}$ are constructed just like in the unmodified Union-find, with the image as its input graph, where the pixels are the vertices and the edges connect each pixel with its neighbours. However, the sets in $\mathcal{Q}_{tree}$ are merged in a specific order. The pixels are visited in decreasing order of level, and each pixel is only merged with neighbours that have been visited before. This causes the highest peaks in the image to merge into multiple-vertex sets first, and the lower pixels are added to them gradually. In the end, the entire graph will be merged into one set. This process resembles an inverted flooding process, where the water level is slowly decreasing. First only the peaks appear, then more and more land emerges and finally the entire landscape is visible.

The procedure `BuildComponentTree` is shown below. It needs a vertex-weighted graph (V, E, F) as its input. For this graph, the pixels of the image are used for the vertices V, E contains all tuples (p, q) where p and q are neighbours in the image and the map F contains all image values, where $F(p)$ gives the value of pixel p.
Internally the procedure also uses the array of nodes called `nodes` and a map `lowestNode`, that stores the index of one of its lowest nodes for each set in $\mathcal{Q}_{tree}$.
In the final lines of the algorithm, the root of the component tree is stored in `Root` and the component map is stored in the map `M`.

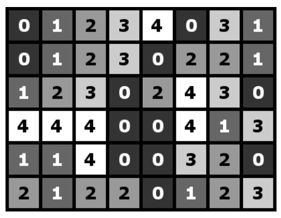
```
Procedure BuildComponentTree (Input $(V, E, F)$)
01. Sort the points in decreasing order of level $F$
02. For All $p \in V$ Do
03.     MakeSet$_{\text{tree}}(p)$; MakeSet$_{\text{node}}(p)$
04.     $nodes[p] \leftarrow$ MakeNode$(F(p))$; $lowestNode[p] \leftarrow p$
05. For All $p \in V$ in decreasing order of level $F$ Do
06.     curTree $\leftarrow$ Find$_{\text{tree}}(p)$
07.     curNode $\leftarrow$ Find$_{\text{node}}(lowestNode[\text{curTree}])$
08.     For All already processed neighbours $q$ of $p$ with $F(q) \geq F(p)$ Do
09.         adjTree $\leftarrow$ Find$_{\text{tree}}(q)$
10.         adjNode $\leftarrow$ Find$_{\text{node}}(lowestNode[\text{adjTree}])$
11.         If curNode $\neq$ adjNode Then
12.             If $nodes$[curNode].level = $nodes$[adjNode].level Then
13.                 curNode $\leftarrow$ MergeNodes(adjNode, curNode)
14.             Else
15.                 $nodes$[curNode].addChild(adjNode)
16.             curTree $\leftarrow$ Link$_{\text{tree}}$(adjTree, curTree)
17.             $lowestNode$[curTree] $\leftarrow$ curNode

18. $Root \leftarrow lowestNode[$Find$_{\text{tree}}($Find$_{\text{node}}(0))]$
19. For All $p \in V$ Do $M(p) \leftarrow$ Find$_{\text{node}}(p)$
```

The `BuildComponentTree` algorithm first sorts the pixels in decreasing order of value in line `01`, and initializes the sets in $\mathcal{Q}_{tree}$ and $\mathcal{Q}_{node}$. Line `03` initializes the nodes with their levels set to their corresponding values in the image, and fills the `lowestNode` map with its initial values.
The main loop starts on line `05`. In each iteration of the loop, one pixel is processed. The pixels are processed in decreasing order.
First, the set in $\mathcal{Q}_{tree}$ that the pixel belongs to is found on line `06`. Then, the set from $\mathcal{Q}_{node}$ that belongs to the lowest node of the set we just found is computed

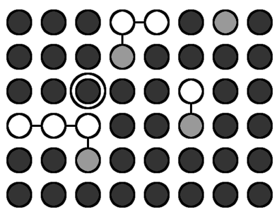
on line `07`. The same is done for each already processed neighbour on lines `09` and `10`.
If the two sets from $\mathcal{Q}_{node}$ of a pixel and its neighbour are not the same (line `11`), the levels of the corresponding nodes are checked. If they are the same, then the two nodes are merged (line `13`). Otherwise, the node that belongs to the neighbour is added to the children of the node that belongs to the currently processed pixel (line `15`). Finally, the two sets in $\mathcal{Q}_{tree}$ are merged and the lowest node of the merged set is set to the node that is being processed in lines `16` and `17`.
Two examples of an iteration of the main loop are illustrated in Figures 3.6 and 3.7.

On line `18`, the root of the component tree is obtained by finding the tree that belongs to the node of an arbitrary (in this case: the first) pixel, and looking up its lowest node. The component map can be computed by finding for each pixel the node that it belongs to.

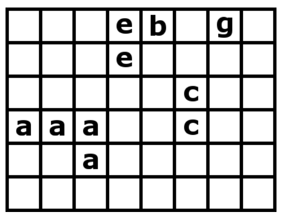


(a) image

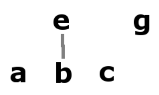
Situation before main loop iteration:



(b) $\mathcal{Q}_{tree}$

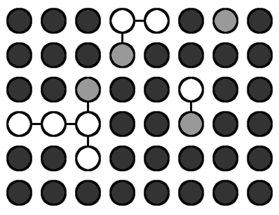


(c) $\mathcal{Q}_{node}$

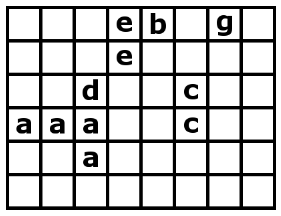


(d) component tree

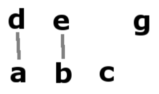
Situation after main loop iteration:



(e) $\mathcal{Q}_{tree}$

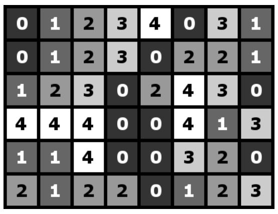


(f) $\mathcal{Q}_{node}$

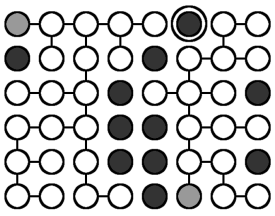


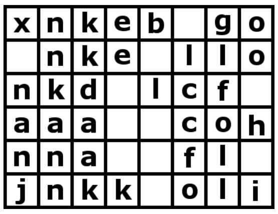
(g) component tree

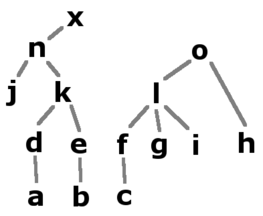
Figure 3.6: The initial and final situation of an iteration of the main loop of the `BuildComponentTree` algorithm. (a) The input image. (b) $\mathcal{Q}_{tree}$ at the beginning of the loop iteration. White and light gray nodes have been processed by the algorithm, dark nodes have not. Connected nodes belong to the same sets in $\mathcal{Q}_{tree}$, with the light gray node being the `lowestNode` of the set. The node marked with a circle around it is the node that will be processed in this iteration. (c) $\mathcal{Q}_{node}$ at the beginning of the iteration. Nodes with the same label belong to the same set in $\mathcal{Q}_{node}$. (d) The component tree at the beginning of the iteration, here consisting of four partial trees. The nodes in the tree correspond to the sets with the same label in (c). (e) $\mathcal{Q}_{tree}$ at the end of the loop iteration. The node that was marked in (b) is now merged with the set next to it, and the newest node in the set is marked as the `lowestNode`. (f) The old `lowestNode` of the newly merged set (marked light gray in (b)) and the new node do not match in value (see (a), where they have values 4 and 3), so the new node is not merged with set a of the old `lowestNode` in $\mathcal{Q}_{node}$. Instead, set a becomes its child in the component tree, as shown in (g).


(a) image

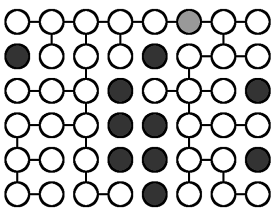
Situation before main loop iteration:

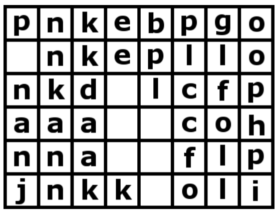

(b) $\mathcal{Q}_{tree}$


(c) $\mathcal{Q}_{node}$


(d) component tree

Situation after main loop iteration:


(e) $\mathcal{Q}_{tree}$


(f) $\mathcal{Q}_{node}$

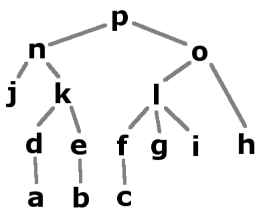

(g) component tree

Figure 3.7: The initial and final situation of another iteration of the main loop of the `BuildComponentTree` algorithm. The same example from Figure 3.6 is used here, at a later iteration. See the caption of Figure 3.6 for more details. In the initial situation, $\mathcal{Q}_{tree}$ contains two processed sets, each with its own lowest node, as shown in (b). Because the node that is to be processed has both sets as its neighbours, we need to look at the sets in $\mathcal{Q}_{node}$ that contain the two lowest nodes: set x with level 0 and set o with level 1. (Sets shown in (c), levels shown in (a).) The node we are processing has level 0, just like set x, so we merge the node with set x into set p (shown in (f)). Set o has a different level, so set o is added to the children of set p (shown in (g)). Finally, the new node is set as the lowest node of the merged set in $\mathcal{Q}_{tree}$, as shown in (e). Note that the result of the algorithm does not contain component m, that was previously shown in the component tree of this image between components o and h. This is because there are no pixels that map to component m. Because the component tree is only used in combination with the component map, the absence of component m has no effect on the output of the algorithm.

3.3 Lowest Common Ancestor

The lowest common ancestor (see Section 2.5 for its definition) is needed in the `HighestFork` function that was described in Section 3.1.1. In this function, we need to compute the LCA of two different components in the component tree. Because the level of each component is stored as well, we can easily find the LCA by comparing the ancestors of the two components at each level. The lowest component that is an ancestor of both components is the lowest common ancestor of the two.
The method for finding the LCA that is described in the previous paragraph is easy to implement, but its execution will have a time complexity that is linear to depth of the component tree. Since the LCA algorithm is needed very often, a faster implementation will have a large impact on the overall performance of the algorithm.
Fortunately, faster algorithms are available. Because the component tree is only computed once and does not change afterwards, we can preprocess the tree to make it easier to find the LCA of its components. A very straightforward preprocessing would be to precompute the LCA of all pairs of components, so the `HighestFork` function can obtain the LCA of any two components in constant time. However, just storing the LCA of all pairs of component will take an amount of storage space that is quadratic to the number of components. This is a problem especially with larger images.
Schieber and Vishkin [11] found an algorithm that can compute the LCA in constant time as well, but needs only linear computation time and storage. Their algorithm will be described here.

The algorithm of Schieber and Vishkin makes use of two special kinds of trees, in which the LCA can be computed in constant time: simple paths and complete binary trees. Both are illustrated in Figure 3.8.
If a tree is a simple path, we can traverse the tree during preprocessing in linear time, storing the level of each pixel. To compute the LCA of two nodes at a later time, we only have to compare the levels of the nodes and return the node with the lowest level value as the lowest common ancestor. See Figure 3.8(a) for an illustration.
For a full binary tree the computation is a bit more complex. The method described by Schieber and Vishkin is explained in the next section.

3.3.1 Computing the LCA in a full binary tree

Before we start computing lowest common ancestors, we need to label the nodes with their inorder number. These numbers can be computed in linear time during preprocessing. We then need to look at the binary representations of these inorder numbers (see Figure 3.8(b)). To find the lowest common ancestor of two nodes, we will compute three bit positions: i_1: the position of the rightmost `1` in the first number; i_2: the position of the rightmost `1` in the second number and i_3: the position of the leftmost bit that is different in both numbers. The bit positions are numbered from right to left, with the rightmost bit having position 0. We take the leftmost of the three positions we just computed, which is the maximum of i_1, i_2 and i_3, and store it as i. Now that we have i, we can obtain the LCA of the two nodes by taking either one of their inorder numbers, and then adapt it by setting the bit at position i to `1` and the bits at positions 0 to $i - 1$ to value `0`. The bits $i + 1$ and above remain equal to the bits at the corresponding positions in the inorder numbers of both input nodes (both nodes have the same bit values at those positions). The resulting number is the inorder number of the LCA of the two nodes. Some examples are shown in Figure 3.8.

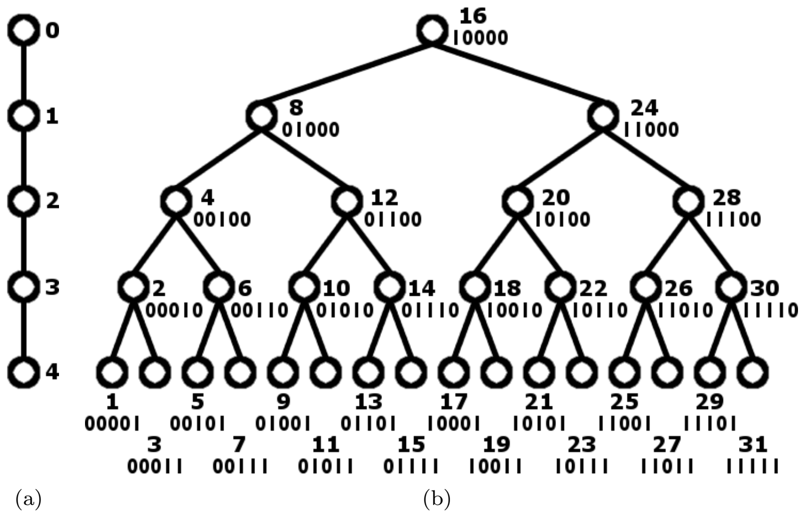


Figure 3.8: Two trees in which the the LCA can be computed in constant time. (a) shows a tree that is a simple path, with the level of each node computed. The LCA of any two nodes is the node that has the lowest level. (b) shows a full binary tree, with the inorder number of each node shown in both decimal and binary notation. If we want to know the LCA of nodes 18 and 29, we have $i_1 = 1$, $i_2 = 0$ and $i_3 = 3$, so $i = 3$. This means that bits 0, 1 and 2 of the LCA have value `0`, bit 3 has value `1` and the value of remaining bit 4 is equal to the value of bit 4 in both 18 and 29: `1`. Together this produces the binary number `11000`, that corresponds to node 24: the LCA of nodes 18 and 29. For nodes 12 and 15, we have $i_1 = 2$, $i_2 = 0$ and $i_3 = 1$, so $i = 2$. This gives us value `0` at bits 0 and 1, value `1` at bit 2 and values `1` and `0` at bits 3 and 4. Together they become binary number `01100`, or node 12, which is the LCA of nodes 12 and 15.

To implement a function that computes the LCA of two nodes in a tree in constant time, we need a machine that can perform multiplication, division, powers-of-two computation, bitwise AND, base-two discrete logarithm, and bitwise exclusive OR in constant time. If these operations can not be computed in constant time, look-up tables for the missing operations can be computed in linear time.
With the ability to perform these operation in constant time, we can construct the following functions that will perform some basic operations, that we will need in our implementation.

The first one is the function `RightmostOne`, that returns the position of the rightmost `1` in the binary representation of the number x:

```
Function RightmostOne (Input $x$)
01. Return $\lfloor \log_2(x - (x \text{ AND } x - 1)) \rfloor$
```

For example, if x is 44, or `101100` in binary representation, the rightmost `1` is computed as follows. First, the binary AND of 44 and $44 - 1$ (43, in binary: `101011`) is computed: `101100` AND `101011` gives `101000`. This number is then subtracted from x: `101100 - 101000` gives `000100`, which is 4 in decimal representation. The $\log_2$ of this number is computed, giving 2.0 as a result. The floor of 2.0 is 2, our final result. The rightmost `1` in `101100` is the third bit from the right, which has position number 2, so this result is correct.

The next function we will use is the function `LeftmostDiff`, that returns the leftmost differing bit of the numbers x and y:

```
Function LeftmostDiff (Input $x, y$)
01. Return $\lfloor \log_2(x \text{ XOR } y) \rfloor$
```

If, for example, x and y are 44 and 37, or `101100` and `100101` in binary, the following steps are performed. First, we take the binary exclusive OR of the two numbers: `101100` XOR `100101` gives `001001` (9 in decimal). The $\log_2$ of 9 is approximately 3.2. The function returns the floor of this number: 3. The leftmost bit that does not correspond in x and y is indeed the bit at position 3, the fourth bit from the right.

The third function, `ZeroRightBits`, sets the n rightmost bits of the number x to zero:

```
Function ZeroRightBits (Input $x, n$)
01. Return $2^n \lfloor x/2^n \rfloor$
```

Note that dividing a number by 2^n (and taking the floor of the result) is equal to shifting its bits n places to the right, and multiplying a number by 2^n is equal to shifting its bits n places to the left. So if, for example, x is 43 (`101011`) and n is 3, x is first divided by $2^3 = 8$, giving $43/8 \approx 5.4$. The floor of this result is 5, or `000101` in binary. Finally we multiply this result again with 2^3, giving $5 * 8 = 40$, or `101000` in binary. This result is equal to x with the 3 rightmost bits set to zero.

Using the three functions we just defined, we can now implement the function `FullBinaryTreeLCA`, that computes the LCA of two nodes x and y in a full binary tree in constant time:

```
Function FullBinaryTreeLCA (Input $x, y$)
01. $i_1$ = RightmostOne($x$); $i_2$ = RightmostOne($y$)
02. $i_3$ = LeftmostDiff($x, y$)
03. $i = \max(i_1, \max(i_2, i_3))$
04. Return ZeroRightBits($x, i + 1$) + $2^i$
```

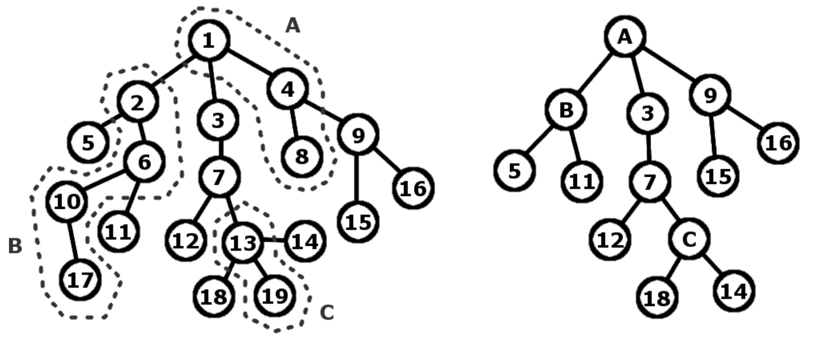
The first three lines do exactly what was described before in this section: they compute the positions of the rightmost `1` in x and y, and the position of the leftmost bit that does not correspond in x and y. The maximum of these three positions is stored as i.
In line `04`, the bit at position i and the bits on the right of bit i are set to zero.

After that, bit i is set to one by adding 2^i to the result of the `ZeroRightBits` function. The resulting number is then returned as the LCA of x and y.

3.3.2 Computing the LCA in arbitrary trees

Now we know how to compute the LCA in constant time in trees that are simple paths or full binary trees, we can use this to compute the LCA in arbitrary trees in constant time as well. This section will describe how. Throughout this section, variables like `inlabel`, `ascendant`, `level` and `head` will be used. All these variables are computed during the preprocessing stage in linear time. The next section will describe how the preprocessing stage is implemented. In this section, we can just assume that the values for all these variables are available for use in constant time.

The general idea is to merge paths in our tree in a way such that the result resembles a full binary tree. We will only merge paths between a node and one of its descendants. Merging paths like this does not change the lowest common ancestors of nodes, as shown in Figure 3.9.



(a) (b)

Figure 3.9: Merging paths from nodes to their descendants in a tree. (a) shows a tree with three paths marked, (b) shows the same tree with the paths merged into one node each. The LCA relations between nodes remain intact after the paths have been merged. For example, nodes 11 and 17 had LCA 6. Nodes 6 and 17 have been merged into node B, so both node 17 and 6 have been replaced by node B. For the LCA relation to remain intact, nodes 11 and B should have LCA B, which is the case in the tree shown in (b). The same goes for nodes 12 and 19 that had LCA 7. Node 19 is merged into node C, but nodes 12 and C still have LCA 7. This holds for all combinations of nodes.

Before we decide which paths to merge, we label each node with its preorder number. We then assign an `inlabel` to each node, which is defined as the preorder number of its descendant that would come the highest in the full binary tree. If the node has no descendant that would come higher in the full binary tree than the node itself, then the `inlabel` is set to its own preorder number. Note that nodes that have the same `inlabel` always form a path. These paths are shown in Figure 3.10(a), their `inlabel` values are shown in Figure 3.10(b). See Figure 3.8(b) for a full binary tree.

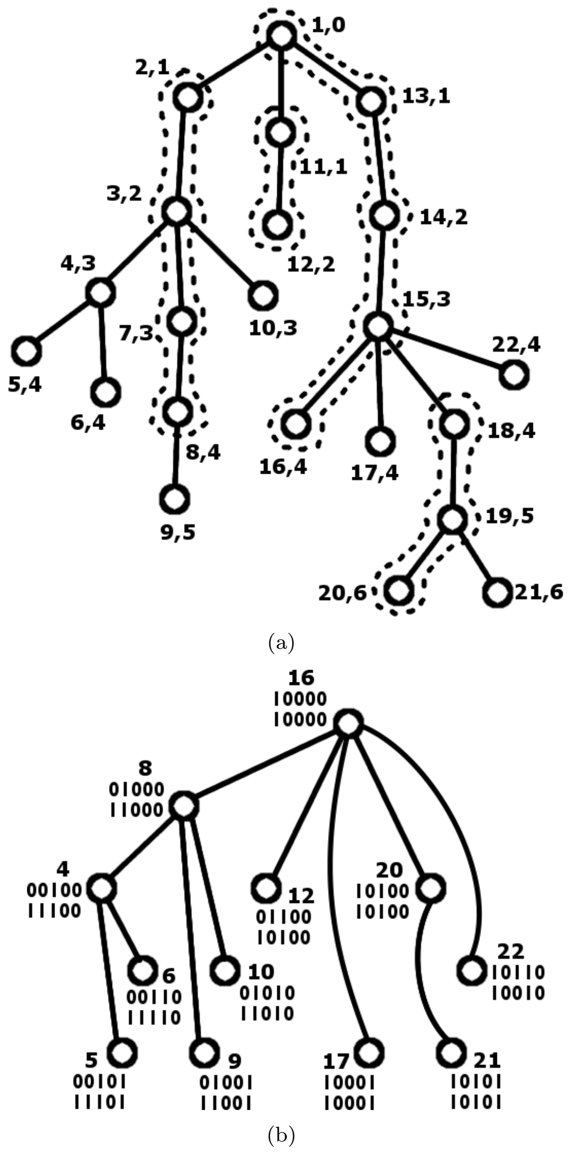


Figure 3.10: (a) A tree. Every node is labelled with its `preorder` number followed with its `level`. Every marked path contains nodes with corresponding `inlabel`s, being the preorder number of the lowest node in the path. All nodes that are not part of a marked path have an `inlabel` that is equal to their own preorder number. (b) The tree of (a), where all nodes with the same `inlabel` have been merged. Every node (or group of merged nodes) is labelled with its `inlabel` in decimal notation, its `inlabel` in binary notation and the value of its `ascendant` variable, respectively. The nodes are located at the place where their `inlabel` value would be in a full binary tree (see Figure 3.8(b)). Note that the tree of (b) is never actually stored in memory. Instead, every node in the tree of (a) stores its own `inlabel` and `ascendant` values, where all nodes in a marked path have the same `inlabel` and `ascendant` values. This figure is a modified version of Figure 3.1 from [11]

If we would merge all nodes with the same `inlabel`, we would get an `inlabel`-tree, as shown in Figure 3.10(b). An especially useful property of this `inlabel`-tree is that for every node in this tree, its set of ancestors is a subset of the set of ancestors of that same node in the full binary tree. For example, the set of ancestors of node 5 in the full binary tree in Figure 3.8(b) is $\{16, 8, 4, 6, 5\}$ (ordered from top to bottom), and the set of ancestors of node 5 in our `inlabel`-tree is its subset $\{16, 8, 4, 5\}$. Note that each node is part of its own ancestor set. We can store the ancestors of each node in the `inlabel`-tree as a binary number, where each bit represents the presence of a node from the ancestor set of the full binary tree. In the example of node 5, the first three elements in both ancestor sets (nodes 16, 8 and 4) correspond, which is stored as three ones. The fourth element of the ancestor set in the full binary tree, node 6, is missing in the ancestor set of the `inlabel`-tree, which is stored as a zero. The final element, node 5, corresponds again, so another `1` is stored. Together, this produces the binary number `11101`, which we store in node 5 as its `ascendant` variable. Other nodes, for example node 12, have smaller ancestor sets in the full binary tree and therefore need less bits to store their ancestors. In those cases, the leftmost bits are used to store which ancestors are present, while the other bits are set to zero. In the case of node 12, the first three bits store its ancestor set, while the other two are set to zero, giving its `ascendant` variable the value `10100`. The `ascendant` values of all nodes in the `inlabel` tree are shown in Figure 3.10(b).

Note that in the full binary tree, the position of the rightmost `1` in the inorder labels is unique for each level. If the rightmost `1` is at position 0 (the rightmost position), the node is at the lowest possible level in the tree, and if the rightmost `1` is at the leftmost bit position, the node is the root, at the highest level in the tree. Because the ancestors of a node all have a different level, they also all have their rightmost `1` at a different position. The `ascendant` variable uses this fact to store at each bit position whether the ancestor that has its rightmost `1` at that same bit position is still an ancestor in the `inlabel`-tree. For example, node 5 in Figure 3.8(b) has ancestors 16, 8, 4 and 6, with their rightmost `1`s at positions 4, 3, 2 and 1. Node 5 itself has its rightmost `1` at position 0. In the `inlabel`-tree of Figure 3.10(b), it only has ancestors 16, 8 and 4 left, with rightmost `1`s at positions 4, 3 and 2. Node 5 itself still has its rightmost `1` at position 0, so its `ascendant` variable has `1`s at positions 4, 3, 2 and 0, producing `ascendant` value of `11101`.

The next step in computing the LCA of two nodes in our original tree of Figure 3.10(a), is finding the LCA of their `inlabel`s in the `inlabel`-tree. To do this, we first want to know the level of their LCA in the full binary tree. We computed this level before in the `FullBinaryTreeLCA` algorithm, where it was stored as i: the rightmost `1` of the LCA. Because the LCA of the two nodes in the full binary tree has its rightmost `1` at position i, we can look at the bit at position i in the `ascendant` values of both nodes to find out if this LCA is still an ancestor of both nodes. If the `ascendant` values of both nodes have a `1` at position i, then the LCA of the two nodes in the `inlabel`-tree is equal to the LCA of the two nodes in the full binary tree. However, if one of them has a `0` at position i, the full binary tree LCA is no longer an ancestor of both nodes and therefore can not be the lowest common ancestor either. The bits to the left of bit i represent the other common ancestors of the two nodes, so if we take

the closest bit to the left of bit i that has value `1` in both `ascendant` values, we have the bit that represents the LCA of the two nodes in the `inlabel`-tree. We store the position of this bit as j. This position j stores, just like position i did for the full binary tree, the position of the rightmost `1` of the LCA in the `inlabel`-tree. We can now use j to construct the label of the LCA in the same way we used i in the `FullBinaryTreeLCA` algorithm: take the label of one of the two nodes, set bit j to `1` and bits 0 to $j-1$ to value `0`.
As an illustration, we will compute the LCA of nodes 9 and 10 in the `inlabel`-tree of Figure 3.10(b). First we find i, the rightmost `1` of the LCA in the full binary tree. The LCA of nodes 9 and 10 in the full binary tree is node 10, which has its rightmost `1` at position 1 in binary notation. So we have $i = 1$. We then look at bit 1 in the `ascendant` values of both nodes 9 and 10, that have values `0` and `1`, respectively. They do not both have value `1`, so we look at the first bit to the left of bit i that does have value `1` for both nodes, which is bit 3. We therefore set j to 3. Finally, we have to take the binary number of either two nodes, `01010` for node 10 or `01001` for node 9 and set bits 0 to $j-1$ to value `0` and bit j to `1`, i.e. bits 0, 1 and 2 to value `0` and bit 3 to value `1`. This results in the binary number `01000`, or the number 8 in decimal notation, which is the LCA of nodes 9 and 10 in our `inlabel`-tree.

For two nodes in our original tree, we can now find the LCA of their `inlabel`s in the `inlabel`-tree. Because the `inlabel`-tree is the original tree with some paths merged, and merging paths in a tree does not change LCA relations as shown in Figure 3.9, this LCA in the `inlabel`-tree should be the `inlabel` of the actual LCA of the two nodes in our original tree. To find this final LCA of our two nodes, the only thing we still have to do is finding which node in its `inlabel`-path is our LCA.
The first step in finding our LCA in its `inlabel`-path is finding for both input nodes their lowest ancestor in the `inlabel`-path. For example, for nodes 5 and 9 in Figure 3.10(a), the `inlabel` of their LCA is `01000`, or 8 in decimal notation. The path of nodes that has `inlabel` 8 contains the nodes 2, 3, 7 and 8. For node 5, its lowest ancestor in this path is node 3, and for node 9 it is node 8. With these lowest ancestors in the `inlabel` path known, we have reduced the problem to finding the LCA in a tree that is a simple path. The LCA is simply the node with the lowest `level` value, node 3 in this case.
However, finding the lowest ancestor in the `inlabel`-path for both input nodes in constant time is not that easy. To achieve this, we will first, for both ancestors, find the son that is also an ancestor of the corresponding input node. In our last example, the son of node 3 that is also an ancestor of node 5 is node 4, and the son of node 8 that is also an ancestor of node 9 is node 9 itself.
To find these sons, we will first compute their `inlabel`s, using the `ascendant` variables of the input nodes themselves and the position of the rightmost `1` in the `inlabel` of their LCA, which we stored before as j. We are looking for the highest ancestor of a node in the `inlabel`-tree that is still below the calculated LCA in the `inlabel`-tree. We can find the rightmost `1` in the `inlabel` of this ancestor by taking the leftmost `1` that is still to the right of bit position j. We store the position of this bit as k. With k, we can construct the entire `inlabel` of the son we are looking for using the method we used before: by taking the `inlabel` of the node itself, setting bit k to value `1` and the bits to the right of bit k to value `0`. For example, if one of the input nodes would be node 21 from

Figure 3.10(b) and the LCA of both input nodes would be node 16, we would be looking for the highest ancestor of node 21 that is still below node 16. To find it, we first need j, the position of the rightmost `1` in the `inlabel` of node 16, which has value 4. So we look at the position of the leftmost `1` to the right of bit 4 in the `ascendant` value of node 21, and store it as k. Node 21 has `ascendant` value `10101`, so k gets value 2. We then construct the `inlabel` we are looking for by taking the `inlabel` of node 21, which is also `10101`, setting bit 2 to value `1` and the bits right of it to value `0`. The obtained `inlabel` is `10100`, which is the `inlabel` of node 20, the highest ancestor of node 21 below node 16.
Now we know how to compute the `inlabel`s of the sons we are looking for, we need to find the sons themselves. Because both sons have an `inlabel` that is different from that of its parent (that has the `inlabel` of the LCA), they should both be at the head of their `inlabel`-path. For this purpose we compute the head of each `inlabel`-path during preprocessing (in linear time) and store it in a look-up table called `head`. We can then use this table to find the sons we are looking for. Once we have both sons, we can find the lowest ancestors of the two input nodes in the `inlabel`-path of their LCA simply by taking the parents of the sons we just found.

As a final example, we will now find the LCA of nodes 10 and 19 in the tree of Figure 3.10(a). Their `inlabel`s are 10 and 20, that have node 16 as their LCA in the `inlabel`-tree. For this LCA, j, the position of it rightmost `1`, has value 4.
The first `1` right of bit 4 in the `ascendant` value of node 10 (`11010`) is bit 3 so $k = 3$ for the first son. By taking the `inlabel` of node 10 (`01010`) and transforming it to have its rightmost `1` at position 3, we obtain the `inlabel` `01000`, or 8, the `inlabel` of the first son. In the look-up table `head` we find that the head of its `inlabel`-path is node 2: the first son.
The first `1` to the right of bit 4 in the `ascendant` value of node 19 (`10100`) is bit 2, so $k = 2$ for the second son. We then transform the `inlabel` of node 19 (also `10100`) to have the rightmost `1` at position 2, obtaining the `inlabel` `10100`, or 20. We find the second son by finding the head of the `inlabel`-path with value 20, which gives us node 18.
Finally, we take the parents of both sons, i.e. the parents of nodes 2 and 18, giving us the lowest ancestors of nodes 10 and 19 in their `inlabel`-path: nodes 1 and 15. We then only have to compute the LCA of nodes 1 and 15 within the `inlabel`-path, which we do simply by looking at their `level`-values. This gives us the LCA of nodes 10 and 19: node 1.

3.3.3 Implementation of the LCA

The implementation of the LCA for arbitrary trees uses the same functions that were used by the implementation of the LCA in full binary tree: the functions `RightMostOne`, `LeftmostDiff` and `ZeroRightBits`. These functions have not changed and will therefore not be described again here.
In addition to those functions, the function `FirstOneRightOf` is used, that computes the position of the leftmost `1` right of position p in the binary number x. It is implemented as follows:

```
Function FirstOneRightOf (Input $x, p$)
01. Return $\lfloor \log_2(x \text{ AND } 2^p - 1) \rfloor$
```

If, for example, $x = 43$ and $p = 3$, we are looking for the first `1` right of bit 3 in the binary number `101011`. We first calculate $2^3 - 1$, which is 7 or `000111` in binary notation. We then compute the binary AND of x and this number: `101011` AND `000111` produces `000011`, or 3 in decimal notation. The $\log_2$ of 3 is about 1.6, and the floor of 1.6 is 1. This 1 is returned, so the leftmost `1` to the right of bit 3 in `101011` should be at position 1, which is correct.

Finally, the function that computes the LCA of two nodes x and y in an arbitrary tree looks as follows:

```
Function ComputeLCA (Input $x, y$)
01. If $inlabel[x] = inlabel[y]$ Then
02.         If $level[x] \leq level[y]$ Then Return $x$ Else Return $y$
03. $i_1$ = RightmostOne($inlabel[x]$); $i_2$ = RightmostOne($inlabel[y]$)
04. $i_3$ = LeftmostDiff($inlabel[x]$,$inlabel[y]$)
05. $i = \max(i_1, \max(i_2,i_3))$
06. common = ZeroRightbits($ascendant[x]$ AND $ascendant[y]$, $i$)
07. $j$ = RightmostOne(common)
08. inlabelz = ZeroRightBits($inlabel[x], j+1$) + $2^j$
09. If $inlabel[x]$ = inlabelz Then $\hat{x} = x$ Else
10.         $k$ = FirstOneRightOf($ascendant[x], j$)
11.         inlabelw = ZeroRightBits($inlabel[x], k+1$) + $2^k$
12.         $\hat{x}$ = parent($head$[inlabelw])
13. If $inlabel[y]$ = inlabelz Then $\hat{y} = y$ Else
14.         $k$ = FirstOneRightOf($ascendant[y], j$)
15.         inlabelw = ZeroRightBits($inlabel[y], k+1$) + $2^k$
16.         $\hat{y}$ = parent($head$[inlabelw])
17. If $level[\hat{x}] \leq level[\hat{y}]$ Then Return $\hat{x}$ Else Return $\hat{y}$
```

On line `01`, the function checks if the `inlabel`s of x and y are equal. If this is the case, the problem is reduced to finding the LCA in a simple path, and the result of this LCA is returned in line `02`.
Otherwise, the position of the rightmost `1` of the LCA of x and y is computed in lines `03` to `05`, and stored as i. These lines are exactly the same as lines `01` to `03` in the `FullBinaryTreeLCA` algorithm that was described before.
The rightmost `1` of the LCA of `inlabel`$[x]$ and `inlabel`$[y]$ is then computed in lines `06` and `07` and its position is stored as j. In line `06`, the common ancestors of `inlabel`$[x]$ and `inlabel`$[y]$ are found by computing the binary AND of `ascendant`$[x]$ and `ascendant`$[y]$ and setting the bits that represent ancestors below their full binary tree LCA (i.e. the i rightmost bits) to zero. In line `07`, the bit that represents the lowest common ancestor is stored as j.
The LCA of `inlabel`$[x]$ and `inlabel`$[y]$ in the `inlabel`-tree, which is equal to the `inlabel` of the LCA of x and y (called z), is then constructed from j on line `08`. This is done by taking the `inlabel` of x, setting bits j and the bits right of it to zero and then adding 2^j, setting bit j to 1.
Lines `09` to `12` compute $\hat{x}$: the lowest ancestor of x that has its `inlabel` equal to `inlabel`$[z]$, the `inlabel` of the LCA of x and y. Line `09` checks if `inlabel`$[x]$ is equal to `inlabel`$[z]$. If so, $\hat{x}$ is simply x. Otherwise, the `inlabel` of w is computed, where w is the son of $\hat{x}$ that is an ancestor of x. The rightmost `1` of this `inlabel` is computed in line `10` by finding the leftmost `1` right of position

j in ascendant$[x]$. Line 11 then constructs inlabel$[w]$ from this position and inlabel$[x]$. Finally, w is found in line 12 by looking up the head of its inlabel-path, and its parent is stored as $\hat{x}$.
Lines 13 to 16 do exactly the same for $\hat{y}$: the lowest ancestor of y that has its inlabel equal to inlabel$[z]$.
Line 17 returns $\hat{y}$ as the LCA of x and y if $\hat{y}$ is higher up in the tree (i.e. has a lower level value) than $\hat{x}$, or returns $\hat{x}$ otherwise.

3.3.4 The preprocessing stage

Before we can perform the computations described in the previous section, we need to do the preprocessing. The preprocessing algorithm takes a tree T as its input and produces four arrays as its output. For each node, its inlabel, ascendant and level values are stored in the corresponding arrays. The array head stores the head of each inlabel-path.
The implementation of the preprocessing stage is shown below:

```
Procedure PreprocessLCA (Input $T$;
   Output $inlabel$, $ascendant$, $level$, $head$)
01. Compute $preorder$ and $size$ by preorder traversal of $T$

02. For All nodes $v$ Do
03.     $i \leftarrow$ LeftmostDiff($preorder[v] - 1$, $preorder[v] + size[v] - 1$)
04.     $inlabel[v] \leftarrow$ ZeroRightBits($preorder[v] + size[v] - 1$, $i$)

05. Compute $level$ by breadth-first traversal of $T$

06. $ascendant[\mathrm{root}(T)] \leftarrow 2^{l-1}$
07. For All nodes $v$, in breadth-first traversal order Do
08.     If $inlabel[v] = inlabel[\mathrm{parent}(v)]$ Then
09.         $ascendant[v] \leftarrow ascendant[\mathrm{parent}(v)]$
10.     Else
11.         $i \leftarrow$ RightmostOne($inlabel[v]$)
12.         $ascendant[v] \leftarrow ascendant[\mathrm{parent}(v)] + 2^i$

13.         $head[inlabel[v]] \leftarrow v$
14. $head[inlabel[\mathrm{root}(T)]] \leftarrow \mathrm{root}(T)$
```

First, two additional arrays, preorder and size, are created to store the preorder number and size of each node, where the size of a node is defined as the number of its descendants including the node itself. The values of both arrays can easily be found by a preorder traversal of T.
In lines 02 to 04, the inlabel of each node is computed. The inlabel of a node v in a preorder numbered tree is defined as the number of its descendant that would come the highest in a (inorder numbered) full binary tree, or, equivalently, the number of its descendant that has the most rightmost zeroes. The descendants of node v range from preorder$[v]$ (node v itself) to preorder$[v]$ + size$[v] - 1$. In line 03 we find i, the position of the rightmost 1 of the number with the most rightmost zeroes in that range. Bit i has value 0 in all numbers before the number we are looking for, and value 1 in the number itself and the numbers after it. All bits left of bit i should have values that remain equal in all numbers in the range we are looking at, so we can find bit i by looking at the leftmost bit that changes in that range. In the case that node v itself has

the rightmost zeroes, bit i will have value `1` in all numbers, so we add the node before v to be sure at least one node will have value `0` at bit position i. For example, in Figure 3.10(a), the node with preorder number 15 has size 8. For this node, `preorder`$[v]-1 =$ `01110` (14) and `preorder`$[v]$ + `size`$[v]-1 =$ `10110` (22). The leftmost differing bit is bit 4, so $i = 4$. The node with preorder number 8 has size 2. For that node, `preorder`$[v] - 1 =$ `00111` (7) and `preorder`$[v]$ + `size`$[v] - 1 =$ `01001` (9), so $i = 3$. Line `04` constructs the `inlabel` from i and the descendant of v with the highest preorder number (`preorder`$[v]$ + `size`$[v] - 1$). Because we are already sure that bit i in this highest preorder number has value `1`, we only have to set the bits to the right of it to value `0`.
In line `05`, the values of the `level` array are found by traversing T in breadth-first order. This traversal may be combined with the loop that starts in line `07`. The `ascendant` array is computed in lines `06` to `12`. In line `06`, the `ascendant` value for the root is constructed by setting the leftmost bit, bit $l - 1$, to value `1`, while the others remain zero. The variable l (length) represents the total number of bits needed to represent the labels of T. With the `ascendant` value of the root known, the loop in line `07` visits all nodes in T in breadth-first order. If the `inlabel` of the node is equal to the `inlabel` of its parent, the `ascendant` values should be the same as well, so the `ascendant` value of the node is set to the value of its parent (lines `08` and `09`). Otherwise, the bit position that represents the current node (i.e. the position of its rightmost `1`) is set to value `1` in its `ascendant` value, as shown in lines `11` and `12`.
The values of the `head` array can be found by setting each node that has an `inlabel` that is different than the `inlabel` of its parent to be the head of its own `inlabel`-path. Because this condition corresponds to the condition of the `Else` branch that starts in line `10`, the initialization of the `head` array is combined with it, as shown in line `13`. The only exception in calculating the heads of the `inlabel`-paths this way is the `inlabel`-path of the root, because the root has no parent with a different `inlabel` (since the root is defined as its own parent). However, we already know that the head of this `inlabel`-path should be the root itself, so this is implemented in line `14`.

Chapter 4

Parallel Implementation

This chapter will describe the proposed parallel algorithm for the topological watershed. The new parallel algorithm is based on the sequential algorithm from Chapter 3, the parallel Max-tree implementation by Wilkinson et al. [14] and the parallel LCA implementation by B. Schieber and U. Vishkin [11].

The sequential implementation of the topological watershed, as described in Chapter 3, used the function `W-Destructible` to determine in constant time whether a pixel was W-destructible. For this function to operate, it needed a component tree and some preprocessing on this component tree to be able to find the lowest common ancestor of any two of its components. To parallelize the computation of the topological watershed, we need a parallel version of all these elements: the main algorithm, that uses the `W-Destructible` function, the computation of the component tree and the preprocessing stage of the LCA.
Section 4.1 will describe the parallel version of the main algorithm, Section 4.2 will describe the parallel component tree algorithm and finally the parallel implementation of the preprocessing will be explained in Section 4.3.

4.1 Parallel topological watershed

The key element of the sequential topological watershed algorithm is the function `W-Destructible`. The algorithm, described in Section 3.1, basically just calls this function for all pixels in the image, using a specific order to keep the execution time linear. Because the function `W-Destructible` is a local function (it only needs information from one pixel and its neighbours), we can parallelize the sequential topological watershed algorithm for n threads simply by dividing the image into n tiles and assigning one tile to each thread. An example division for a 2D image is illustrated in Figure 4.1(a) and a division for a 3D volume is shown in Figure 4.1(d).
However, since the `W-Destructible` function is dependent on the neighbours of the pixel it is examining, problems may arise when examining border pixels, because their neighbours in other tiles can be changed at any time by their assigned threads, which may produce incorrect results. To prevent this from happening, we need to prevent adjacent pixels in different tiles to be processed at the same time. We can achieve this by letting each thread process its tile in different stages, and synchronizing all threads after each stage. Each tile is

divided into subtiles, and a different subtile is processed in each stage. The tiles are divided in such a way that no two adjacent subtiles need to be processed at the same time. Whether or not two subtiles are adjacent depends on the connectivity (described in Section 2.1). Subtile divisions for each connectivity are shown in Figure 4.1.

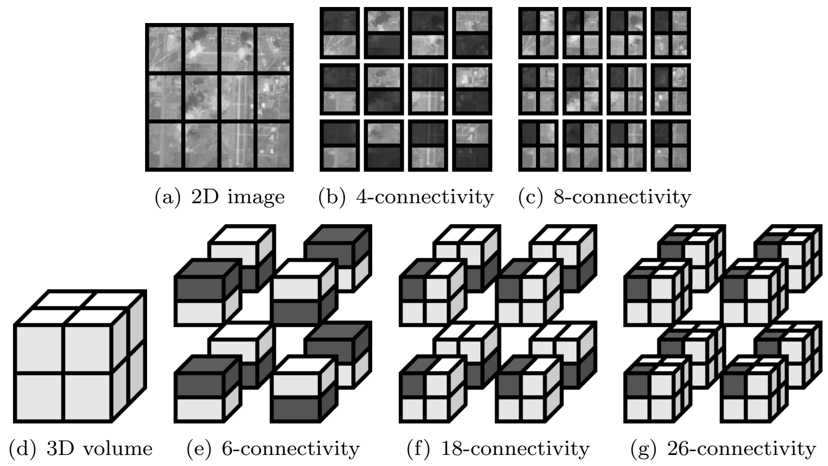
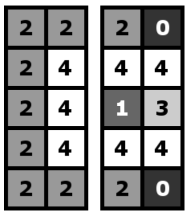


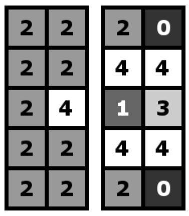
Figure 4.1: (a) shows an image divided into 12 tiles for 12 threads. With 4-connectivity, each tile should be divided into 2 subtiles, as shown in (b). The dark subtiles represent the subtiles that are processed in the first stage. Note that no two dark subtiles are neighbours of each other. However, they would be neighbours with 8-connectivity, so 4 subtiles are used for 8-connectivity, as shown in (c). (d) shows a 3D volume divided into 8 'tiles' for 8 threads. For 6-, 18- and 26-connectivity, 2, 4 and 8 subtiles are needed, respectively. This is illustrated in Figures (e), (f) and (g). Again, the dark subtiles represent the subtiles to be processed in the first stage.

Now each thread knows which subtile it should process in each stage, an algorithm much like the sequential topological watershed algorithm from Section 3.1 can be used to process each subtile. However, a few adjustments have to be made.
For example, a certain pixel x may need to be lowered to the value of some local minimum y to obtain a topological watershed of the input image. However, if pixels x and y are part of different subtiles, it is possible that pixel x will not get the value of y the first time its subtile is processed. Multiple iterations may be needed to obtain the desired result, as shown in Figure 4.2.

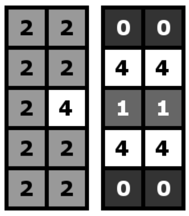
The sequential main topological watershed algorithm from Section 3.1 processes all pixels in the order that is determined by the priority queue. In the beginning of the algorithm, all pixels are added to this priority queue. In the parallel implementation, this may be a good approach when the pixels are processed for the first time, but visiting each pixel in every later iteration as well is not necessary. Instead, we will keep track of the pixels that have been changed in the most recent iteration, and only add those pixels to the priority queue that are adjacent to pixels in other subtiles that changed recently. For this purpose we will use the binary map `pxChanged`, that will store for each pixel whether or



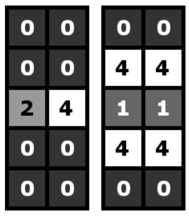
(a) before processing



(b) first iteration, first stage complete



(c) first iteration, both stages complete



(d) second iteration, first stage complete

Figure 4.2: An example where multiple iterations are needed. The original image, divided into two subtiles, is shown in (a). The left subtile is processed in the first stage. Only pixels from the left subtile are changed, but values of adjacent pixels from the right subtile may be used to determine if a pixel in the left subtile is W-destructible. (b) shows the result after the first stage. The right subtile is processed in the second stage, completing the first iteration. However, as displayed in (c), the result is not yet a topological watershed. A second iteration is needed to obtain a correct result, as shown in (d).

not its value has changed recently. The use of this map is illustrated in Figure 4.3.

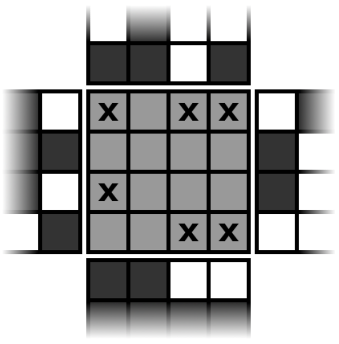


Figure 4.3: An example of a `pxChanged` map in an iteration after the first. The white pixels in the four adjacent subtiles are pixels that are marked as 'changed' in the `pxChanged` map. The pixels in the current subtile that are adjacent to such a changed pixel, marked with x in this figure, are added to the priority queue of the sequential algorithm.

The `TopologicalWatershed` procedure from Section 3.1 roughly consists of two parts: first the priority queue L is initialized (lines `01` to `06`) followed by the main loop of the algorithm (lines `07` to `19`). The parallel version of the algorithm is a bit more complex, so the algorithm is distributed over two procedures: the procedure `InitializeQueue`, which corresponds to the first part of the sequential algorithm, and the procedure `TopologicalWatershedTile`, that corresponds to the second part. Both procedures are given below.

The `InitializeQueue` procedure needs the image F, the component tree of its inverse $\mathcal{C}(\overline{F})$ and the corresponding component map Ψ as its input, just like in the sequential algorithm. Additionally, it also needs the subtile T in which it should operate, and the current state of the `pxChanged` map. The output consists of the priority queue L, the maps K and H as explained in Section 3.1, and the updated `pxChanged` map. The queue L and the maps K and H are all local, but the map `pxChanged` is global, and may be read and modified by other threads while this procedure is being executed. However, each thread will only write in the part of the `pxChanged` map that corresponds to its current subtile, and will only read in adjacent subtiles that are processed in a different stage, so no conflicts will emerge.

```
Procedure InitializeQueue (Input $F, \mathcal{C}(\overline{F}), \Psi, T, pxChanged$;
   Output $L, K, H, pxChanged$)
01. For $k$ From $k_{\min}$ To $k_{\max} - 1$ Do $L_k \leftarrow \emptyset$
02. For All pixels $p \in T$ Do $pxChanged[p] \leftarrow$ false
03. If first iteration Then
04.      For All pixels $p \in T$ Do
05.          $c \leftarrow$ W-Destructible$(F, p, \mathcal{C}(\overline{F}), \Psi)$
06.          If $c \neq \emptyset$ Then
07.              $i \leftarrow$ level of $c$; $L_i \leftarrow L_i \cup \{p\}$
08.              $K(p) \leftarrow i; H(p) \leftarrow$ pointer to $c$
09. Else
10.      For All border pixels $p$ of $T$ Do
11.          addP $\leftarrow$ false
12.          For All neighbours $q$ of $p$ Do
13.              If $pxChanged[q]$ = true Then addP $\leftarrow$ true
14.          If addP = true Then
15.              $c \leftarrow$ W-Destructible$(F, p, \mathcal{C}(\overline{F}), \Psi)$
16.              If $c \neq \emptyset$ Then
17.                  $i \leftarrow$ level of $c$; $L_i \leftarrow L_i \cup \{p\}$
18.                  $K(p) \leftarrow i; H(p) \leftarrow$ pointer to $c$
```

The algorithm starts by initializing the priority queue L in line `01`. It then proceeds by setting the `pxChanged` map to `false` for every pixel in subtile T. If the procedure is run during the first iteration, then lines `04` to `08` are executed. In the first iteration, L is initialized exactly like in the original sequential algorithm, apart from the fact that only the pixels within T are processed instead of all pixels in F.
If the procedure is called after the first iteration, lines `09` to `18` are executed. In these lines the algorithm checks all border pixels for changed neighbours, and tests the pixels for W-destructibility if any changed neighbours are found. If a pixel turns out to be W-destructible, it is added to the priority queue, and the maps K and H are updated as before.

The second procedure, `TopologicalWatershedTile`, needs the same input as the procedure `InitializeQueue`. The output consists of the updated image F, the update `pxChanged` map, and the binary variable `anyChanges`. The variable `anyChanges` is used to quickly determine if any changes have occurred in the subtile during the execution of this procedure.

```
Procedure TopologicalWatershedTile (Input $F, \mathcal{C}(\overline{F}), \Psi$, $T$, pxChanged;
    Output $F$, pxChanged, anyChanges)
01. anyChanges $\leftarrow$ false
02. InitializeQueue($F, \mathcal{C}(\overline{F}), \Psi$, $T$, pxChanged)
03. For $k$ From $k_{\min}$ To $k_{\max} - 1$ Do
04.         While $\exists p \in L_k$ Do
05.             $L_k = L_k \backslash \{p\}$
06.             If $K(p) = k$ Then
07.                 $F(p) \leftarrow k; \Psi(p) \leftarrow H(p)$
08.                 pxChanged[p] $\leftarrow$ true; anyChanges $\leftarrow$ true
09.                 For All neighbours $q$ of $p$ within $T$, with $k < F(q)$ Do
10.                     $c \leftarrow$ W-Destructible($F, q, \mathcal{C}(\overline{F}), \Psi$)
11.                     If $c = \emptyset$ Then $K(q) \leftarrow \infty$
12.                     Else
13.                         $i \leftarrow$ level of $c$
14.                         If $K(q) \neq i$ Then
15.                             $L_i \leftarrow L_i \cup \{q\}; K(q) \leftarrow i$
16.                             $H(q) \leftarrow$ pointer to $c$
```

The procedure `TopologicalWatershedTile` starts by initializing the value of `anyChanges`. The function call on line `02` produces an initialized priority queue L, as well as initialized maps K and H. The rest of the function is mostly the same as the second part of the `TopologicalWatershed` procedure from Section 3.1. Line `08` is added, where the `pxChanged` map and the `anyChanges` variable are updated. Also, a new restriction is added to line `09`, saying that only neighbours of p that lie within the subtile T should be added to the priority queue.

With the procedure `TopologicalWatershedTile` implemented, we can now define the main parallel algorithm: the procedure `ParallelTW`. It needs the pixel mapping F, the component tree of the inverse image $\mathcal{C}(\overline{F})$ and the corresponding component map Ψ as its input, just like the sequential algorithm. Because each thread will run this procedure independently, the global map `pxChanged` needs to be provided to each thread as well. However, no initial values need to be stored in it. Additionally, each thread is provided its identifier `id`. The first thread gets `id` value `0`, the second gets value `1` and so on. The output of the procedure is the updated map F, that now contains the topological watershed of the input image.

```
Procedure ParallelTW (Input $F, \mathcal{C}(\overline{F}), \Psi$, pxChanged, id; Output $F$)
01. done $\leftarrow$ false
02. While done = false Do
03.         anyChangesThr[id] $\leftarrow$ false
04.         For All stages $s$ Do
05.             $T \leftarrow$ current subtile, based on id and $s$
06.             TopologicalWatershedTile($F, \mathcal{C}(\overline{F}), \Psi$, $T$, pxChanged)
07.             If anyChanges = true Then anyChangesThr[id] $\leftarrow$ true
08.             Barrier()
```

```
09.     If $id = 0$ Then
10.         $anyChangesAtAll$ ← false
11.         For All threads $t$ Do
12.             If $anyChangesThr[t]$ = true Then
13.                 $anyChangesAtAll$ ← true
14.     Barrier()
15.     If $anyChangesAtAll$ = false Then done ← true
```

The algorithm keeps looping until a topological watershed of the input image is found. In each iteration, the loop that starts on line `02` is executed once by each thread. Line `04` starts a loop that visits all stages. The number of stages is equal to the number of subtiles assigned to each thread, as shown in Figure 4.1. Line `05` then determines the location and dimensions of the subtile that should be processed by the current thread in the current stage. Some examples of the subtiles that should be processed by each thread in the first stage are displayed in Figure 4.1. In the other stages the subtiles that are processed should have a similar pattern, always assuring that no two adjacent subtiles are processed at the same time.
The topological watershed of the tile is then computed on line `06`. If the algorithm returns that there have been some changes, then this is stored for the current thread in the (global) `anyChangesThr` array. After this, a standard barrier function is called, that just waits until all threads have reached this barrier and then lets all thread continue. This is done to ensure that no thread will start with the next stage until all threads are done with the current one.
When all threads have finished processing all their subtiles, the first thread will check if there have been any changes in any of the threads. If there have not been any changes at all, a topological watershed has been found and all threads will terminate. Otherwise, each thread will go to the next iteration by starting again with the main loop from line `02`.

4.2 The component tree

Wilkinson et al. [14] described how to parallelize the computation of a component tree, their method will be described here. Because the algorithm from [14] also deals with features of the component tree that we won't use, its implementation is simplified somewhat in this report.
Basically, the sequential algorithm is parallelized by letting multiple threads each compute the component tree of a different part of the input image, and merging the component trees of the parts afterwards. The actual connecting of component trees is done with the procedure `Connect`. The main algorithm, the procedure `ParallelComponentTree`, uses the procedure `Connect` to merge all component trees into one component tree, which also happens in parallel. Finally, the component tree and component map are simplified with the procedure `CompressTree`. But first of all, we will look at an additional function, called `LevelRoot`.

The function `LevelRoot` returns the *level root* of a component, which is defined as the highest ancestor of that component in the component tree that still has the same level as the component itself. For example, if both a component x and its parent y have the same level, but z, the parent of y has a different level,

then y is the level root of x, and y is also its own level root. Initially, each node in a component tree is its own level root, but while merging two component trees this may change, as we will see in the implementation of the procedure `Connect`.
The function `LevelRoot` needs a component c, the map F and the component tree $\mathcal{C}(\overline{F})$ as its input, and returns the level root of c. It is implemented as follows.

```
Function LevelRoot (Input $c$, $F$, $\mathcal{C}(\overline{F})$)
01. If $c = \text{root}(\mathcal{C}(\overline{F})) \vee F(c) \neq F(\text{parent}(c))$ Then
02.     Return $c$
03. Else
04.     $\text{parent}(c) \leftarrow$ LevelRoot$(\text{parent}(c), \mathcal{C}(\overline{F}), F)$
05.     Return $\text{parent}(c)$
```

Line `01` checks if the component c is the root, or if the parent of c has a different level than c itself. In both cases, c is returned as its own level root. Otherwise, the level root of the parent of c is computed recursively, stored and then returned.
Technically, the map F maps pixels to their corresponding gray levels, so using the map to retrieve the level of a component should technically be impossible. However, all pixels in a component have the same level and the component is represented by a canonical element (which does correspond to one pixel), using the map F to obtain the level of the component c should not be a problem in practice.

With the `LevelRoot` function defined, we can now implement the procedure `Connect`. This procedure takes two components x and y as its input, and merges the path from x to the root of its tree with the path from y to the root of its tree. To do this, it uses three iterators: x, y and z. Initially, iterator x points to component x and iterator y points to component y. The rest of the input consists of the map F, the component tree $\mathcal{C}_x$ that contains the component x and the component tree $\mathcal{C}_y$ that contains the component y. The output of the algorithm is the merged component tree, that contains both component x and y. Also note that this algorithm uses a different way of handling roots of trees than we did before. Here, the root has parent $\perp$, instead of being its own parent. This difference has no significant impact on how the algorithm works, but is kept here to keep the code consistent with the code from [14]. The algorithm is implemented as follows:

```
Procedure Connect (Input $x$, $y$, $F$, $\mathcal{C}_x$, $\mathcal{C}_y$; Output $\mathcal{C}_{x,y}$)
01. $x \leftarrow$ LevelRoot$(x, F, \mathcal{C}_x)$; $y \leftarrow$ LevelRoot$(y, F, \mathcal{C}_y)$
02. If $F(y) > F(x)$ Then $\text{swap}(x, y)$
03. While $(x \neq y) \wedge (y \neq \perp)$ Do
04.     If $\text{parent}(x) \neq \perp$ Then $z \leftarrow$ LevelRoot$(\text{parent}(x))$ Else $z \leftarrow \perp$
05.     If $z \neq \perp \wedge F(z) \geq F(y)$ Then
06.         $x \leftarrow z$
07.     Else
08.         $\text{parent}(x) \leftarrow y$
09.         $x \leftarrow y$; $y \leftarrow z$
```

On line `01`, the iterators x and y are set to the level roots of the components they were pointing to. Line 2 makes sure iterator x has the highest level by swapping the iterators if y has a higher level initially. Again, the map F is used to obtain the level of a component instead of the level of a pixel, as explained before in this section. The algorithm then starts looping until either y has reached the root of its tree or x is equal to y. Each loop iteration starts with z being set to the level root of the parent of x, unless x is the root of its tree, in which case z is set to $\perp$. If the value of the component pointed at by z is higher than or equal to that of y, then x is set to z. Otherwise, the component of y is set to be the parent of the component of x, x is set to y and y is set to z. What this means in practice is illustrated in Figures 4.4 and 4.5.

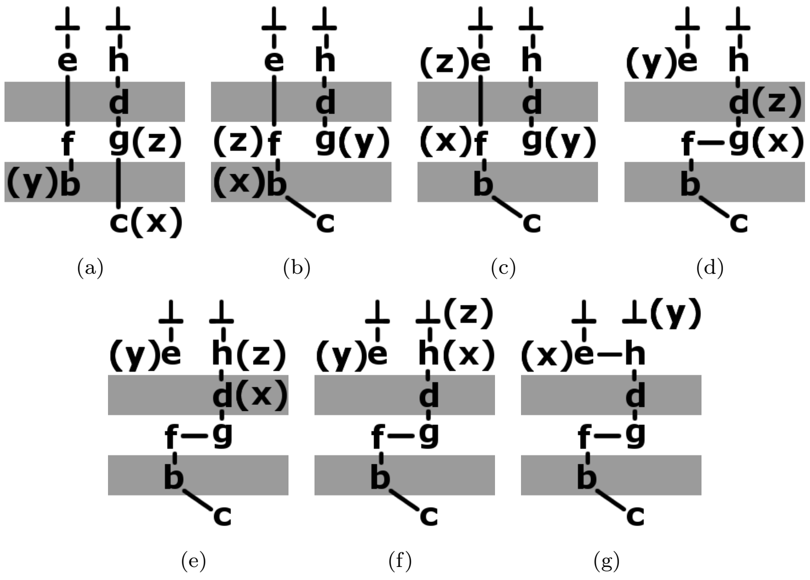


Figure 4.4: The `Connect` procedure for components b and c from different component trees. In (a), x is first set to the component that has the highest level of the two, in this case c. (Because the root is at the top of the image, the components with the highest levels are at the bottom of the image.) At the start of the first iteration, z is set to the level root of the parent of x (which is the parent itself, in this case). The level of z is not greater than or equal to the level of y, so y is set as the parent of x, x is set to y and y is set to z, as shown in (b). Also z is set again to the level root of the parent of x in the next loop iteration, also displayed in (b). Now the level of z is equal to the level of y, and $z \neq \perp$, so x is set to z and y remains unchanged. The result is shown in (c), again with z set to be the level root of the parent of x. This process is continued until finally $y = \perp$ in (g), and the loop is terminated.

We now want to use the `Connect` procedure to merge the component trees of two adjacent partial images. To achieve this, we take all pixel pairs (p, q) with p in one of the partial image, q in the other and p and q adjacent to each other. We then find the corresponding component pair (x, y) for each pixel pair by using the component map Ψ. Finally, we execute the `Connect` procedure for each pair of components, which should result in the component tree of the two partial images combined when the last pair is ‘connected’. This is illustrated in Figure 4.6.

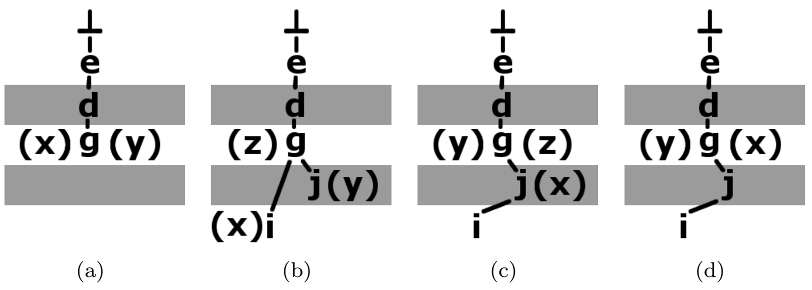


Figure 4.5: The `Connect` procedure for components in the same component tree. (a) shows the case where (the level roots of) the two input components are both g, in which case $x = y$ and the loop is terminated immediately. Figures (b) to (d) show the `Connect` procedure for components i and j. After two loop iterations $x = y$, as shown in (d), which terminates the loop.

Now we know how to merge the component trees of two adjacent partial images, we can implement the main algorithm: the `ParallelComponentTree` procedure. This procedure divides its input image into a number of slices that is equal to the number of threads. First each thread will then compute the component tree for its slice. After this, all component trees will be merged together into the component tree of the entire image, which also happens in parallel, in multiple iterations. In the first iteration, the slices are grouped into pairs, and the component trees of each pair are merged. The merging of a pair is performed by a single thread, so only half the amount of available threads can be used in the first iteration, since the number of pairs will be at most half the number of slices and therefore half the number of threads. In the second iteration, the merged slices are grouped into pairs again and then merged. Because the total number of pairs will be only half the number of pairs of the previous iteration, only a quarter of the available threads will be used in the second iteration. This merging process continues until the final iteration, in which the component trees of the final two merged slices will be merged into the component tree of the entire image. Since the merging of a single pair is performed by a single thread, the final iteration is performed by only one thread.
To coordinate which slices should merge in each iteration, and which thread should perform this merging, the following system is used. In the first iteration, each thread that has an identifier that is a multiple of 2 (i.e. an even number), merges its slice with the next adjacent slice, that has the identifier that is equal to its own identifier plus 1. This means that slices $\{0, 2, 4...\}$ merge with slices $\{1, 3, 5...\}$, respectively. In the next iteration, the slices that have identifiers that are multiples of 2*2 merge with the slices that have their identifier plus 1*2, meaning that slices $\{0, 4, 8...\}$ merge with slices $\{2, 6, 10...\}$, respectively. In the third iteration, the multiples of 2*2*2 merge with the slice that has their own identifier plus 1*2*2, and so on. This merging process is illustrated in Figure 4.7. If the total number of threads is not a power of two, the system still works, but some threads will be inactive during some iterations. For example, we could remove threads and therefore slices 6 and 7 from Figure 4.7. In that case, the six remaining threads would still react the same, except the thread responsible for

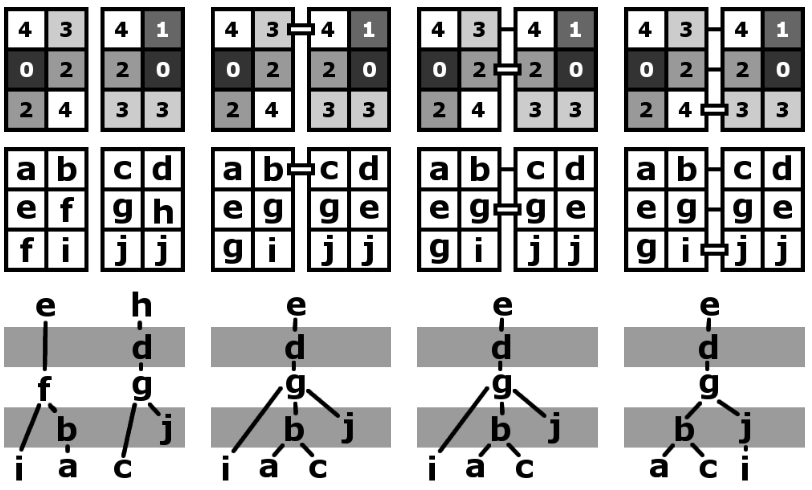


Figure 4.6: Merging the component trees of two partial images. The partial images are shown in the top row, the middle row shows *the level roots of* the components in the component maps of the images and the bottom row shows the component trees during the process. The leftmost column shows the initial situation, the other columns show the situation after the marked pairs have been connected. Because the `Connect` procedure only looks at the level roots of the components, the components that are not level roots (i.e. components f and h after the first pair is merged) are not shown in this figure for clarity. The steps of the `Connect` procedure for the first pair is shown in Figure 4.4. Figure 4.5(a) shows the `Connect` procedure for the second pair and Figures 4.5(b) to 4.5(d) show the `Connect` procedure for the third pair. Just like in this figure, Figure 4.5 does not show the components that are not level roots. The final component tree in this figure, where all pairs have been connected, is the component tree of the two partial images combined.

slice 4, which would now be inactive in the second iteration instead of merging slices 4 and 6.

The procedure `ParallelComponentTree` uses binary semaphores to synchronize the threads. Each thread has two own semaphores: `sa` and `sb`. The value of `sa` of a thread is set to 1 with the atomic action V when its tree is ready to be merged into the tree of another thread. When a thread wants to merge its tree with the tree of another thread, it first waits until the semaphore of that other thread is set to 1, using the action P. Similarly, if a thread is done with merging its tree with other trees, it calls P on its semaphore `sb`. This causes the thread to wait until its value of `sb` is set to 1 after thread 0 has called V on it.
The `ParallelComponentTree` procedure needs, just like the sequential component tree implementation, a vertex-weighted graph (V, E, F) as its input, where V contains all pixels, E contains all pairs of pixels from V that are neighbours in the image and F maps each pixel to its value. Additionally, each thread gets its own identifier `id`. When all threads have finished executing, the output of the algorithm will be the component tree $\mathcal{C}(\overline{F})$ and the component map Ψ.

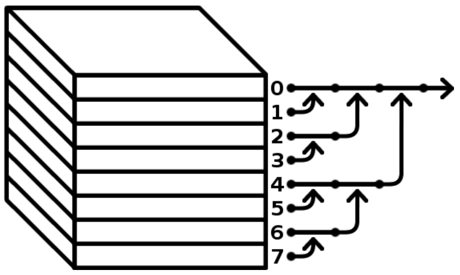


Figure 4.7: A 3D volume divided into eight slices, with the arrows indicating the merging process of the component trees of the slices. For 2D images the merging process stays the same, the image can simply be considered as a 3D volume with depth 1, represented by the front face of the block in this figure. In the first iteration, thread 0 merges its component tree with the component tree of thread 1, thread 2 merges its tree with the tree of thread 3, thread 4 its tree with the tree of 5 and thread 6 with the tree of 7. In the second iteration, thread 0 merges its (merged) tree with the tree of thread 2 and thread 4 merges its tree with the tree of thread 6. Finally, in the third iteration thread 0 merges its tree with the tree of thread 4, resulting in the final component tree.

```
Procedure ParallelComponentTree (Input $(V, E, F)$, $id$; Output $\mathcal{C}(\overline{F}), \Psi$)
01. $\mathcal{C}[id] \leftarrow$ component tree of slice $id$; update $\Psi$ for the pixels in slice $id$
02. Barrier()
03. step $\leftarrow$ 1; $\text{id}_2 \leftarrow$ id
04. While (id + step $< K$) $\wedge$ ($\text{id}_2$ MOD 2 = 0) Do
05.         P($sa[id$+step]) (* wait to connect with next neighbour *)
06.         For All edges $(p, q)$ between tree $id$ and tree ($id$+step) Do
07.               Connect($\Psi(p)$, $\Psi(q)$, $F$, $\mathcal{C}[id]$, $\mathcal{C}[id$+step])
08.         step $\leftarrow$ step * 2; $\text{id}_2 \leftarrow \text{id}_2$ / 2
09. If $id = 0$ Then
10.         For All threads $t$ Do V($sb[t]$) (* release the waiting threads *)
11. Else
12.         V($sa[id]$) (* signal previous neighbour *)
13.         P($sb[id]$) (* wait for thread 0 *)
14. CompressTree($F$, $\mathcal{C}[0]$, $\Psi$, $id$)
15. Barrier()
16. If $id = 0$ Then $\mathcal{C}(\overline{F}) \leftarrow \mathcal{C}[0]$
```

In line 01, each thread calculates the component tree of the slice that was assigned to that thread, using the pixels from V that lie within the slice, the pairs from E that connect pixels in the slice and the map F. All threads use the same component map Ψ to store the component maps of their component trees, but all use different parts of this map, so no conflicts will emerge there. The barrier in line 02 waits until all threads have finished computing their component tree, before continuing with the rest of the algorithm.
On line 03 the two local variables step and id_2 are initialized. The variable id_2 is used to determine if the thread should still do something in the current iteration. Its value is initially equal to the identifier of the thread, but is divided by 2 after each iteration. The variable step is used to calculate the index of

the tree that the thread should merge its own tree with.
The main loop of each thread starts on line 04. The first condition of the loop checks if the tree the thread wants to merge with its own tree exists, by checking if id + step $< K$, where K is the total number of threads or slices. The second condition checks whether $\texttt{id}_2$ is a multiple of 2.
On line 05, the thread calls P on the semaphore sa of the thread that is responsible for the tree it wants to merge with. This causes the thread to wait until V has been called for that same semaphore. When this has happened, the thread moves on to the loop that starts on line 06. In this loop the thread loops through all edges or pairs from E that have one pixel in both trees that are to be merged. On line 07, the components of both pixels are obtained using Ψ, and the resulting components are connected using the procedure Connect. On line 08, at the end of the loop, the step variable is multiplied by two. Also, the $\texttt{id}_2$ variable is divided by two, causing the second condition of the loop to check if id is a multiple of 4 in the second iteration, a multiple of 8 in the third iteration, and so on.
After the loop has terminated, the behaviour of the thread depends on the value of its identifier. If the thread is not thread 0, the thread calls V on its own semaphore sa signifying that its tree is ready to be merged into another tree. Then it calls P on its semaphore sb, causing the thread to wait for thread 0. If the thread has identifier 0, it calls V on the semaphore sb for all other threads, enabling all threads to continue to line 14.
On line 14, each thread calls the CompressTree procedure, which will be described next. When all threads have finished executing this procedure and have passed the barrier in line 15, finally thread 0 sets its component tree as the final component tree $\mathcal{C}(\overline{F})$.

The procedure CompressTree that was mentioned before in the procedure ParallelComponentTree compresses the component tree by removing all components that are not level roots.
To be able to compress the tree in parallel, we will divide the component tree and component map into segments, and let the threads process one segment each. When a thread would call our current LevelRoot function on one of the components in its segment, it may also change the parent of one of the ancestors of the component. However, if that ancestor is part of a segment that is processed by another thread, this may cause conflicts between the threads. To prevent this, we will use a slightly modified function to obtain the level root of a component, called LevelRoot2. This function is implemented as follows:

```
Function LevelRoot2 (Input $c$, $F$, $\mathcal{C}(\overline{F})$)
01. If $c = \text{root}(\mathcal{C}(\overline{F})) \vee F(c) \neq F(\text{parent}(c))$ Then
02.         Return $c$
03. Else
04.         Return LevelRoot2(parent($c$), $\mathcal{C}(\overline{F})$, $F$)
```

This function is almost identical to the original LevelRoot function, apart from the fact that it only returns the level root of its parent in line 04, instead of storing it first and then returning it.

The procedure CompressTree needs a map F, a component tree $\mathcal{C}(\overline{F})$ and a corresponding component map Ψ as its input, as well as the identifier id of the thread that executes it. The output of the procedure consists of the compressed component tree and map.

```
Procedure CompressTree (Input $F$, $\mathcal{C}(\overline{F})$, $\Psi$, $id$; Output $\mathcal{C}(\overline{F})$, $\Psi$)
01. For All components $c \in$ segment $id$ of $\mathcal{C}(\overline{F})$ Do
02.         parent($c$) $\leftarrow$ LevelRoot2(parent($c$), $F$, $\mathcal{C}(\overline{F})$)
03. For All pixels $p \in$ segment $id$ of $F$ Do
04.         $\Psi(p) \leftarrow$ LevelRoot2($\Psi(p)$, $F$, $\mathcal{C}(\overline{F})$)
```

In the first two lines, the algorithm loops through the components in the segment of the component tree that is assigned to the current thread, and sets the parent of each component to be the level root of that parent.
The last two lines loop through the segment of the component map that is assigned to the thread, and sets each component in the component map to be the level root of that component.
When this process is completed, all components that are not level roots are no longer used and may be deleted from the memory.

4.3 Lowest Common Ancestor

The sequential version of the LCA implementation of Schieber and Vishkin [11], as described in Section 3.3, consists of a preprocessing stage and a part where the actual LCA of two tree nodes is computed. Fortunately, this computation part is already in constant time and therefore does not need to be parallelized. For the preprocessing stage, which their sequential algorithm performs in sequential time, Schieber and Vishkin introduced a parallel algorithm. The method described here is based on their algorithm.

The sequential LCA preprocessing algorithm made use of preorder and breadth-first traversal of the tree, to compute the values needed for the `inlabel`, `ascendant`, `level` and `head` arrays. However, we can not use these methods in the parallel algorithm, because letting one thread traverse the entire tree would already take linear time. Instead we will use the so-called *Euler tour technique*.
In the Euler tour technique we add for each edge $(u \to v)$ in the tree its antiparallel edge $(v \to u)$. In the graph that results, the indegree of each vertex is equal to its outdegree, the graph has an *Euler path* that starts and ends in the root, meaning that we can traverse the entire graph visiting each edge exactly once, starting and ending in the root vertex. Because we also want to visit each vertex exactly once, we will add new vertices at nodes that have an indegree higher than 1. For example, if a vertex b has indegree 3, we will split up vertex b into three vertices b_0, b_1 and b_2, each with indegree (and outdegree) 1. For a vertex c with indegree 1 however, we will only keep one vertex, called c_0.
To make sure that all vertices in the graph remain connected, we will distribute the incoming and outgoing edges for a vertex v as follows:

- the incoming edge from the parent of v will point to v_0
- if v has children, v_0 will have an outgoing edge to the first child, v_1 will have an outgoing edge to the second child, etc.
- again if v has children, the incoming edge back from the first child will point to v_1, the incoming edge from the second child will point to v_2, etc.

- v_i, where i is the number of children of v, points back to the parent of v

An example of a tree transformed into a graph like this is shown in Figure 4.8. The graph that emerges is path-shaped. If we would consider this path as a list and assign a weight to each edge, we can compute for each vertex the distance to the end of the list using *list ranking.* Using this technique it is not hard to compute the preorder number or the level of each vertex from the tree, as illustrated in Figure 4.8.

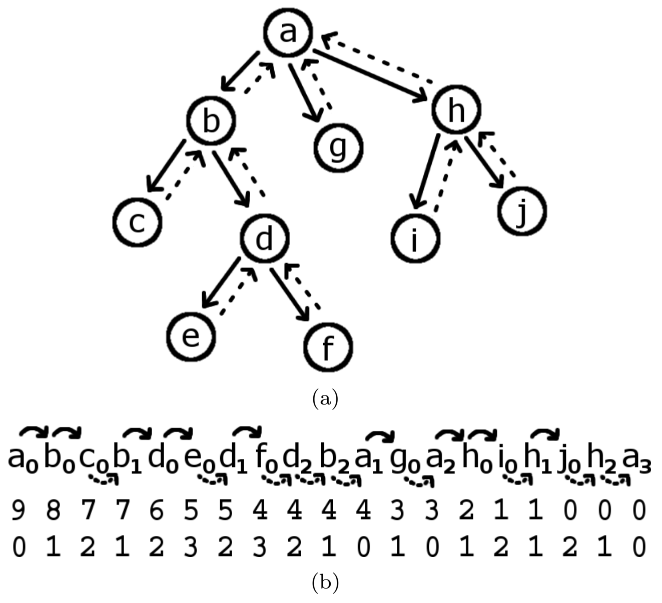


Figure 4.8: Illustration of the Euler tour technique. (a) shows a tree with the antiparallel edges added. (b) shows an Euler tour through this tree that visits all edges exactly once. The edges directed to the root are displayed as dashed arrows, the edges directed from the root are displayed as solid arrows. The first line of numbers shows the total distance to the end of the list for each vertex if the solid arrows represent a distance of 1 and the dashed arrows a distance of 0. The second line shows the distance to the end of the list if the solid arrows represent a 'distance' of -1 and the dashed arrows a distance of 1. The preorder number of a vertex v from the tree of (a) can be found by computing 10 (the number of tree nodes) minus the distance of v_0 from the first line. The level of each vertex v can be found by simply taking the distance of v_0 from the second line. For example, vertex d from (a) has preorder number $10 - 6 = 4$ and level 2.

To construct this 'Euler tour'-list in parallel, we first take the complete set Q of all vertices in our input tree T. We then divide this set Q into as many intervals as there are threads. Each thread is assigned its own interval I. Because vertices in the tree that have children will produce multiple vertices in our final list, we have an additional set R to store these vertices. Note that each child always causes exactly one additional vertex to appear for its parent. Since all vertices except for the root are children of another vertex, the size of

set R will therefore be equal to the size of set Q minus one. If we store set Q in an array, we can create an array for R with the same size, where the position of each vertex in Q corresponds with the position of the vertex it causes to appear in R. Using this approach, we can use the same interval I for both Q and R.
For each list vertex, both in the sets Q and R, a pointer to the next vertex in the list is stored. Also, for each vertex in Q, a pointer to the last vertex that belongs to the same tree node is stored in the variable `lastPart`. For a vertex v with i children, the `lastPart` variable of v_0 should eventually point to v_i.
The algorithm that constructs the Euler tour, `ParallelEulerTour`, makes use of barriers like before, and of mutual exclusion (mutex) objects. A mutex works similar to a binary semaphore. A mutex can be used to make sure certain variables will never be written to by two threads at the same time. If a thread wants to modify such a variable, it first tries to lock the responsible mutex. If it is already locked, the thread will wait until the mutex becomes unlocked and then obtain the lock itself. The thread can then be sure no other threads have access to the variable and safely modify it. Afterwards, the thread needs to unlock the mutex again so other threads can modify the variable.
The algorithm needs the tree T, the set of vertices in the tree Q and the interval of the current thread I as its input, and as its output it updates the set Q and initializes its part of the set R. Additionally, a mutex object for each element of Q is needed.

```
Procedure ParallelEulerTour (Input $T$, $Q$, $I$; Output $Q$, $R$)
01. For All vertices $v$ in $Q(I)$ Do
02.     $v$.lastPart $\leftarrow v$; $v$.next $\leftarrow -1$
03. Barrier()
04. For All vertices $v$ in $Q(I)$ Do
05.     If $v \neq$ root($T$) Then
06.         $u \leftarrow$ parent($v$)
07.         mutex_lock($u$); mutex_lock($v$)
08.         Add new vertex $w$ to $R(I)$
09.         $w$.next $\leftarrow u$.lastPart.next
10.         $u$.lastPart.next $\leftarrow v$
11.         $v$.lastPart.next $\leftarrow w$
12.         $u$.lastPart $\leftarrow w$
13.         mutex_unlock($v$); mutex_unlock($u$)
14. Barrier()
```

In the first two lines, the vertices from Q that lie within the interval I are initialized. The `lastPart` variable is set to the vertex itself, and the `next` variable is set to -1, meaning that there is no next vertex (yet). All threads are then synchronized in line `03`. Line `04` loops through all vertices in the interval I of Q again, and unless the vertex is the root, a number of actions are performed for each vertex. As an example, we will look at node d being added to node b, as shown in Figures 4.9(c) (before adding node d) and 4.9(d) (after adding).
First, the node and its parent are locked using the corresponding mutex objects, in this case the objects for d_0 and b_0. Vertex b_1 does not need to be locked, as it is part of R and not of Q. Then a new vertex, b_2 is created. The `next` variable of b_2 is set to b_0`.lastPart.next`, which is b_1`.next`, which is a_1. The `next` variable of b_1 is then set to d_0, and the `next` variable of the `lastPart` of

d_0, d_0 itself, is set to b_2. Finally, the `lastPart` variable of b_0 is set to b_2.

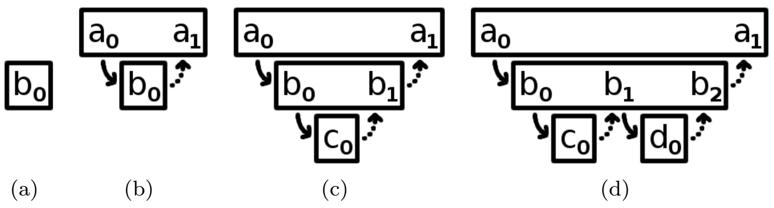


Figure 4.9: The evolution of the vertices that belong to tree node b during the parallel construction of the Euler tour. (a) shows the initial vertex, with just b_0. b_0 has a `lastPart`-pointer that points to b_0 itself. (b) shows tree node b after it has added itself as a child to tree node a. This caused a new vertex, a_1 to appear. The `next` variables of vertices a_0 and b_0 are set as indicated by the arrows, and the `lastPart` variable of a_0 is set to the new vertex a_1. (c) shows the situation after vertex c has added itself as a child to tree node b, causing b_1 to appear. The `next` variables of b_0, c_0 and b_1 are updated, and the `lastPart` variable of b_0 is set to b_1. Finally, (d) shows the situation after vertex d has added itself as a child to tree node b, causing b_2 to appear. The `next` values for b_1, d_0 and b_2 are updated, and the `lastPart` value of b_0 is set to b_2.

With the Euler tour constructed, we now need to perform the *list ranking*. A list ranking algorithm takes a linked list as its input, with a weight assigned to each link. The algorithm then computes the sum of all weights of the links it has to pass through to reach the end of the list. An example of such a list is shown in 4.10(a), where each link has a weight of 1. After the list ranking algorithm, each item knows its 'distance' to the end of the list, or its 'rank', as shown in 4.10(f).
A simple parallel list ranking algorithm, loosely based on an algorithm described in [15], is described below. In our case, the list consists of two parts: the sets Q and R. Before the procedure is called, each item in the sets Q and R needs to have its `distToNext` variable initialized to the weight of the edge between the item itself and the item it points to. Each thread is assigned its own interval I in the sets Q and R, just like in the `ParallelEulerTour` procedure. Running the procedure results in all items in the sets Q and R having their ranks computed, based on the weights that were stored in the `distToNext` variables. The rank of the list items can be obtained from the same `distToNext` variables after running the procedure. Again barriers are used to synchronize the threads. A mutex object is used for each object in Q and R, to make sure that the pointer and distance variable of each list item are updated atomically.
The algorithm is implemented as follows:

```
Procedure ParallelListRanking (Input $Q$, $R$, $I$; Output $Q$, $R$)
01. For All vertices $v \in Q(I) \cup R(I)$ Do $v.\text{next}_2 \leftarrow v.\text{next}$
02. Barrier()
03. For All vertices $v \in Q(I) \cup R(I)$ Do
04.     While $v.\text{next}_2 \neq -1$ Do
05.         mutex_lock($v.\text{next}_2$)
06.         nextnext $\leftarrow v.\text{next}_2.\text{next}_2$
07.         nextdist $\leftarrow v.\text{next}_2.\text{distToNext}$
08.         mutex_unlock($v.\text{next}_2$)
09.         mutex_lock($v$)
10.         $v.\text{next} \leftarrow$ nextnext
11.         $v.\text{distToNext} \leftarrow v.\text{distToNext} +$ nextdist
12.         mutex_unlock($v$)
13. Barrier()
```

Each list item has a variable `next`$_2$ that stores the item it currently points to, and a variable `distToNext` that stores the distance to that item. Each thread initializes the `next`$_2$ variables of the items in its interval I to the `next` variables that were computed in the `ParallelEulerTour` procedure. The threads are then synchronized in line `02` to make sure all list items are initialized before starting the main loop.
Each thread then loops through all list items in its assigned interval in line `03`. For each item it starts another loop in line `04`, which loops until the `next`$_2$ variable does not point to a list item anymore, which means the end of the list has been reached. In this loop, first the item that is pointed at by `next`$_2$ is locked, to make sure no values change while they are read. The `next`$_2$ and `distToNext` values of that list item are then read and stored in temporary variables. After the list item has been unlocked, the current list item is locked to make sure no other threads read its values while they are changed. Then, the `next`$_2$ value of the current item is set to the item that the other item pointed to, and the distance that was just found is added to the distance that was already stored in the `distToNext` variable of the current item. The current item is then unlocked and the next iteration starts, until the rank of the list item is found.
While this algorithm does work, a lot of performance may be lost or gained depending on the order the vertices in $Q(I)$ and $R(I)$ are processed by the thread that is responsible for them. If at some point the current item from $Q(I) \cup R(I)$ points to another item in $Q(I) \cup R(I)$, it may be wise to process that other item first, and resume processing the current item later, to prevent the thread from computing the same steps multiple times.
The parallel list ranking algorithm is illustrated in Figure 4.10.

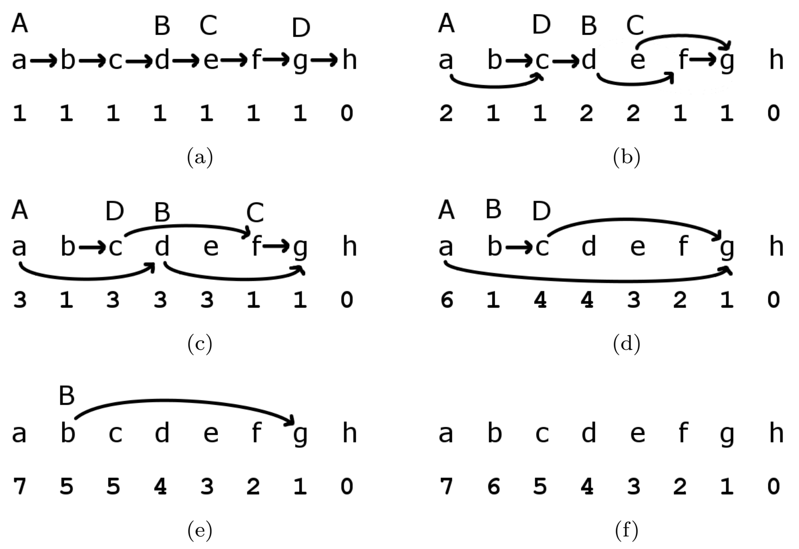


Figure 4.10: Parallel list ranking on a list with eight items with four threads. The lower case letters represent the list items, the upper case letters represent the threads. The list items are assigned to the threads randomly. In each successive image, each thread will perform one loop iteration for the item it is currently processing. In reality however, some threads may perform more loop iterations than others in the same time interval, as the threads are not synchronized during the process. The number below each list item shows the distance to the item it is currently pointing to. As an example of what happens in each loop iteration, we will look at thread A in Figures (c) and (d). Thread A processes item a which points at item d in (c), with distance 3. Item d in its turn points at item g which is at a distance of 3 from d. Thread A then lets item a point at g and adds the distances: $3 + 3 = 6$. The result is shown in (d). Thread A only modified list item a, and did not change d. However, thread B did modify item d, the result of which is also visible in (d). Gradually more and more items point to nothing, meaning that they have found their final rank.

Now we have the procedures `ParallelEulerTour` and `ParallelListRanking` available, we can implement the parallel LCA preprocessing algorithm. As its input it takes a tree T, as well as the identifier of the current thread `id`. The output of the algorithm consists of the arrays `inlabel`, `ascendant`, `level` and `head`, just like the sequential implementation. The algorithm is pretty long, but the basic idea is quite similar to the sequential implementation. An explanation is provided below it.

```
Procedure ParallelLCAPreprocessing (Input $T$, $id$;
   Output $inlabel$, $ascendant$, $level$, $head$)
01. $Q \leftarrow$ the vertices of $T$
02. $I \leftarrow$ the interval of $Q$ that is assigned to thread $id$
03. ParallelEulerTour($T$, $Q$, $I$)

04. For All vertices $v \in Q(I) \cup R(I)$ Do
05.     If $v$.next $\in Q$ Then $v$.distToNext $\leftarrow 1$
06.     Else $v$.distToNext $\leftarrow 0$
07. ParallelListRanking($Q$, $R$, $I$)
08. For All vertices $v \in Q(I)$ Do
09.     $preorder[v] \leftarrow N - v$.distToNext
10.     $size[v] \leftarrow v$.distToNext $- v$.lastPart.distToNext $+ 1$
11.     $i \leftarrow$ LeftmostDiff($preorder[v] - 1$, $preorder[v] + size[v] - 1$)
12.     $inlabel[v] \leftarrow$ ZeroRightBits($preorder[v] + size[v] - 1$, $i$)

13. For All vertices $v \in Q(I) \cup R(I)$ Do
14.     If $v$.next $\in Q$ Then $v$.distToNext $\leftarrow -1$
15.     Else $v$.distToNext $\leftarrow 1$
16. ParallelListRanking($Q$, $R$, $I$)
17. For All vertices $v \in Q(I)$ Do
18.     $level[v] \leftarrow v$.distToNext

19. For All vertices $v \in Q(I) \cup R(I)$ Do $v$.distToNext $\leftarrow 0$
20. Barrier()
21. For All vertices $v \in Q(I) \cup R(I)$ Do
22.     If $v$.next $\in Q$ Then
23.         If $inlabel[v$.next$] \neq inlabel[$parent$(v$.next$)]$ Then
24.             $i \leftarrow$ RightmostOne($inlabel[v$.next$]$)
25.             $v$.distToNext $\leftarrow -2^i$
26.             $v$.next.lastPart.distToNext $\leftarrow 2^i$
27.             $head[inlabel[v$.next$]] \leftarrow v$.next
28. ParallelListRanking($Q$, $R$, $I$)
29. For All vertices $v \in Q(I)$ Do
30.     $ascendant[v] \leftarrow v$.distToNext $+ 2^{l-1}$
31. If ($id = 0$) Then
32.     $head[inlabel[$root$(T)]] \leftarrow$ root$(T)$
```

In the first three lines, Q and I are initialized and the Euler tour is computed. Lines `04` to `06` initialize the weights of the links. Links that lead to items from Q (the vertices with subscript 0, the solid arrows in Figure 4.8(b)) are initialized with weight 1, the others with weight 0. After the parallel list ranking has completed, the preorder number of each vertex is found by subtracting the rank of the vertex in Q from the total number of tree nodes N. The size of a vertex can be found by subtracting the rank of the last part of the vertex from the rank of the first part of the vertex and adding one. For example, the rank of b_2 in Figure 4.8(b) is 4 and the rank of b_0 is 8. $8 - 4 + 1 = 5$, which is the size of tree node b in Figure 4.8(a). With the preorder number and size of each vertex known, the `inlabel` of the nodes can be computed for each node in the same way as in the sequential algorithm.
The level of the vertices is computed in lines `13` to `18`. First the weights are initialized, now -1 and 1 instead of 0 and 1, and the parallel list ranking is

performed again. The level of a tree vertex can then easily be obtained by just taking the rank of the corresponding vertex from Q, as illustrated in Figure 4.8(b). Note however, that the rank of vertex a_3 will not get the value 0 in our implementation, as it is displayed in the figure, but will get the value 1 instead. This will not cause problems because the rank of a_3 will never be used.
When computing the level of a vertex, each edge directed towards the root has weight 1, because each vertex is one level further away from the root than its parent. Similarly, each edge directed from the root has weight -1, because the level of each vertex is one level closer to the root than its child. The `ascendant` value of a vertex was defined in the sequential procedure `PreprocessLCA` from Section 3.3 as the `ascendant` value of its parent, unless the vertex had a different `inlabel` than its parent. If the `inlabel`s were different, the `ascendant` value of the vertex was defined as the `ascendant` value of the parent plus 2^i, where i was the position of the rightmost `1` in the binary representation of the `inlabel` of the vertex. In our parallel algorithm, we only have to store the difference in the values from the parent to its child and the other way around, just like with the computation of the `level` values. So if the `inlabel` of a vertex and its parent are the same, both differences will be 0, as defined in line `19`. If they are different however, the differences are -2^i and 2^i. Therefore the weight of the edge from parent to child is set to -2^i in line `25`, and the weight of the edge back from the child to its parent is set to 2^i in line `26`, where i is the rightmost one of the `inlabel` of the child. As an illustration to this, we will look back at Figure 3.10. In this figure, node 6 has `inlabel 00110`, while its parent, node 4, has `inlabel 00100`. The `inlabel`s are different, so we set the difference between the `ascendant` values of the two to 2^i (or -2^i for the other way around). Here i is the rightmost 1 in the `inlabel` of node 6, which is at position 1. The difference between the `ascendant` values should therefore be $2^1 = 2$ or `00010` in binary notation. In Figure 3.10 we can see that the `ascendant` values of the nodes are `11110` and `11100`, which indeed have a difference of `00010`.
After the list ranking in line `28`, we can find the `ascendant` value of each vertex by taking its list rank and adding the `ascendant` value of the root, which is 2^{l-1}, as defined before in the sequential algorithm.
The only thing left to do is filling the `head` array. A vertex is the head of its `inlabel` path if its `inlabel` differs from the `inlabel` of its parent. So if this is the case, we set the vertex to the head of its `inlabel` path in line `27`. Finally, thread 0 sets the root of the tree as the head of its `inlabel` path on line `32`, to complete the `head` array.

Chapter 5

Results

The implementation was tested on the following four different input images:

- a satellite image of an airfield, with a size of 4000 $\times$ 4000 pixels
- a random image of 4000 $\times$ 4000 pixels, in which each pixel has a random gray value
- an image with a spiral shaped plateau of 1000 $\times$ 1000 pixels
- an angiogram with dimensions 256 $\times$ 256 $\times$ 128

The images are displayed in Figure 5.1.

The topological watersheds of these images were computed on a machine with four 6-core Opteron processors in the following ways:

- of the satellite image with 4-connectivity using 1 to 24 threads
- of the satellite image with 8-connectivity using 1, 8, 16 and 24 threads
- of the angiogram with 6-connectivity using 1, 8, 16 and 24 threads
- of the angiogram with 18-connectivity using 1, 8, 16 and 24 threads
- of the angiogram with 26-connectivity using 1, 8, 16 and 24 threads
- of the random image with 4-connectivity using 1, 8, 16 and 24 threads
- of the spiral image with 4-connectivity using 1, 8, 16 and 24 threads

Also, the previously existing sequential algorithm by Couprie et al. [5] and the newly implemented parallel algorithm were run on the same machine with the same input (the satellite image, with 4-connectivity using 1 thread) to compare the two versions.

The total wall-clock time of the algorithm consists of four stages: the construction of the component tree, the compression of this component tree, the preprocessing for the LCA and finally the computation of the topological watershed of the image. The contribution of each of these stages while processing the satellite image with 4-connectivity on 1 to 24 threads is displayed in Figure 5.2. Their individual speedups are shown in Figure 5.3. The construction of

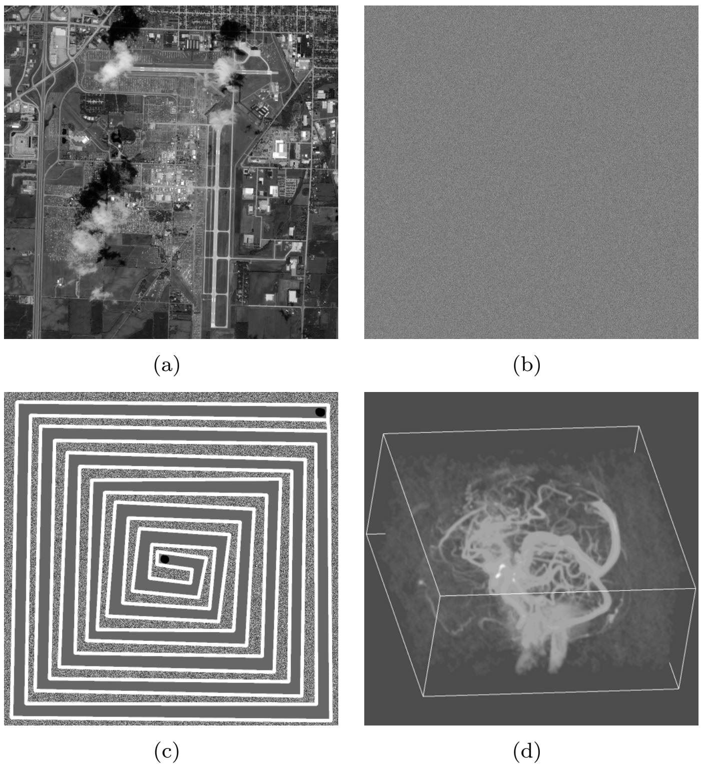

(a) (b) (c) (d)

Figure 5.1: The four test input images. (a) shows a satellite image, (b) an image with random pixel values, (c) an image with a spiral-shaped plateau and (d) an angiogram, which is a 3D volume.

the component tree and the final stage that computes the topological watershed parallelize quite well, while the tree compression and the LCA preprocessing parallelize rather poorly. Fortunately, the tree compression takes up only a small percentage of the total wall-clock time, even when 24 threads are used. However, this percentage will probably increase when more threads are used. The LCA has a larger impact on the total wall-clock time of the algorithm, and will cause the speedup of the total algorithm to decrease more severely when using a larger number of threads.

The speedups for the total wall-clock times for the previously listed tests are displayed in Figure 5.4. In this figure we can see that the speedup when using 24 threads is around 11 on average, not considering the special case with the spiral-shaped plateau. This means that the wall-clock time with 24 threads is about 11 times shorter than when only a single thread is used. The shape of the graphs suggest that an even better speedup may be obtained when more than 24 threads are used.
However, these speedups are relative to the wall-clock times of the parallel algorithm using one thread. In practice, the previously existing sequential implementation performs about 1.5 times faster than parallel algorithm when only one thread is used. Further optimizations in the parallel algorithm, for example by using a better tree compression method, may reduce this difference.

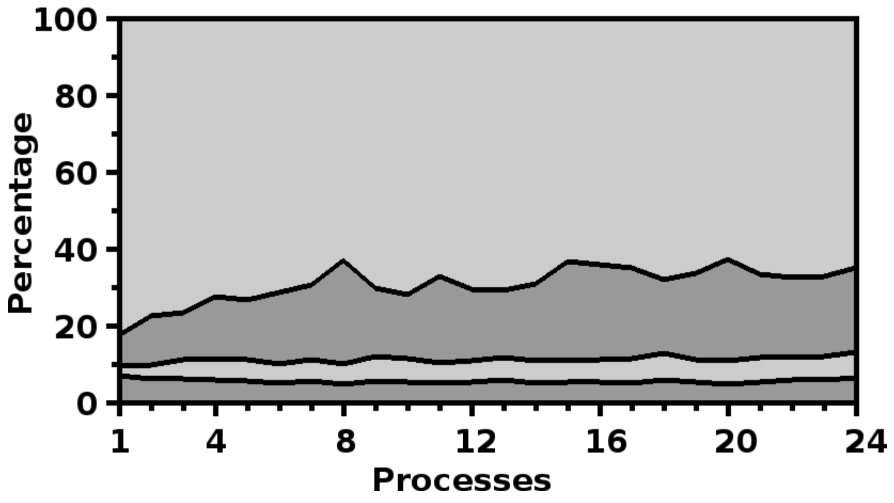
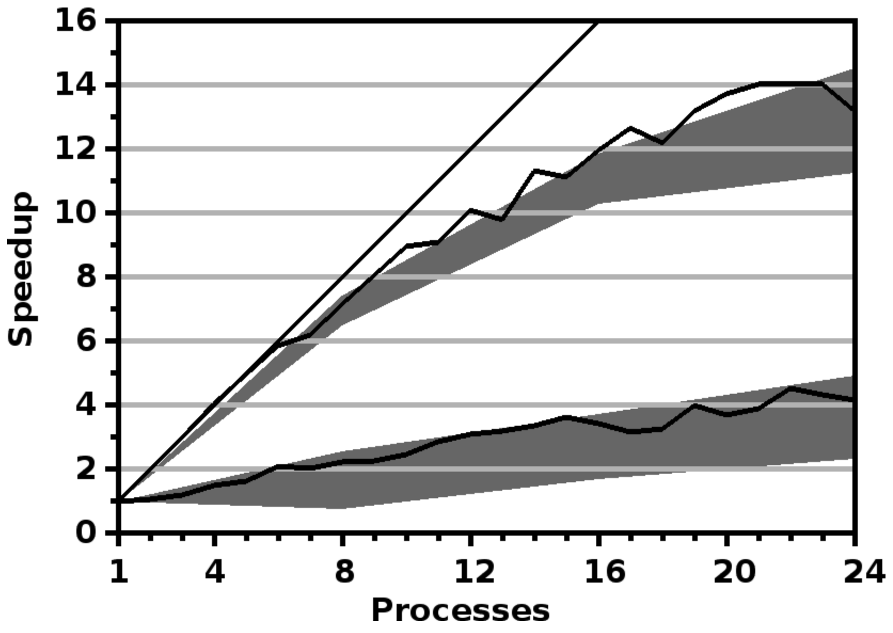
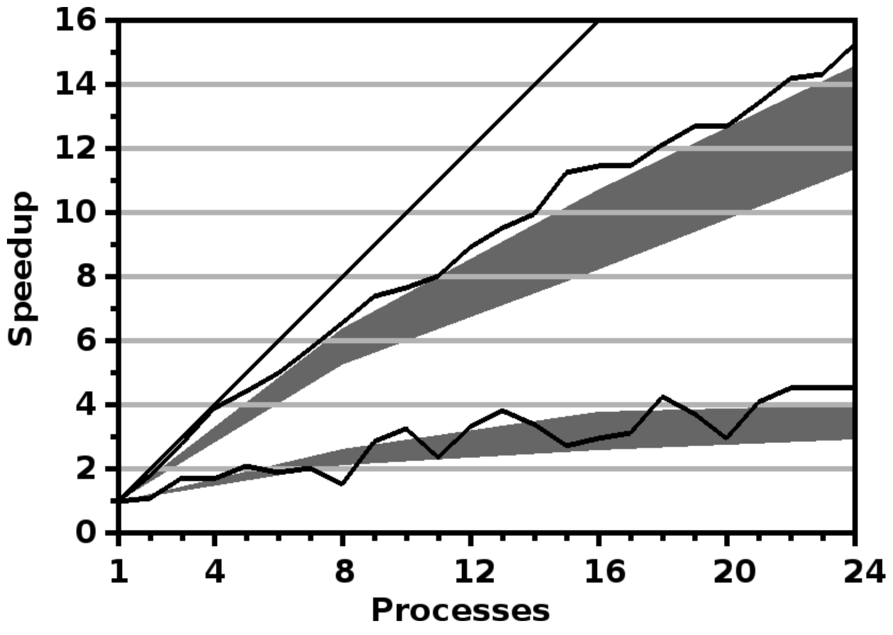


Figure 5.2: The contribution of the four stages to the total wall-clock time of the algorithm, when computing the topological watershed for the satellite image with 4-connectivity. The bottom layer represents the time consumed by the construction of the component tree, the second layer from the bottom represents the time taken to compress the component tree, the third layer shows the time it takes to perform the preprocessing for the LCA, and finally the remaining layer on top represents the time consumed by the final stage which produces the topological watershed.

The results for the spiral image show that in that case the parallel algorithm does not perform well at all. This is due to the large spiral shaped plateau in the image, that has a minimum on both ends. The plateau needs to be shrunk into a single watershed line. This is done by lowering the pixels of the plateau, starting at the minima and traversing the entire spiral until only a line is left. However, when many threads are used, the plateau lies in many different tiles. Each time the current border of the plateau reaches a new tile, the thread responsible for that tile needs to take over the lowering of the plateau pixels, but not before the next iteration has started. Therefore a lot of communication between threads and a lot of iterations are needed to obtain the final result, which causes the bad performance. In practice however, the unlikely requirements for this situation will probably not occur very often.



(a) Top: component tree construction, bottom: tree compression



(b) Top: topological watershed computation, bottom: LCA preprocessing

Figure 5.3: The speedups of the four stages of the algorithm. The diagonal lines show the ideal speedup, where the speedup is equal to the number of threads. The speedup of each stage is represented by a black line and a gray area, where the line shows the speedup of that stage when computing the satellite image with 4-connectivity. The corresponding gray area represents the results of the remaining tests, except for tests on the spiral image. The top and bottom of the gray areas are defined as the average speedup of their stage plus and minus the standard deviation, respectively.

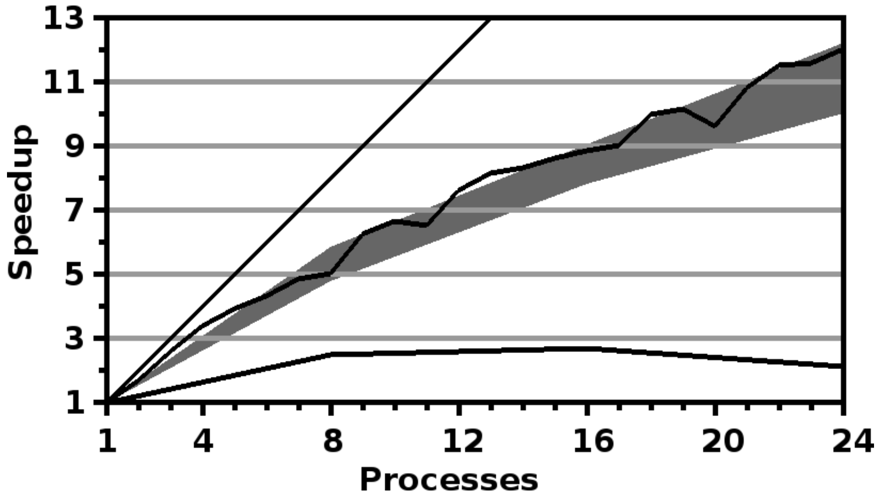


Figure 5.4: The total speedups in the performed tests, relative to the wall-clock time for a single thread with the same input. The diagonal line shows the ideal speedup, the line at the bottom shows the speedup for the spiral image. The line through the gray area shows the speedup for the satellite image with 4-connectivity, which is computed for each number of threads individually. The gray area again represents the results of the remaining tests, with its boundaries defined as in Figure 5.3, but now for all four stages combined.

Chapter 6

Conclusions and Discussion

This report described a way to parallelize the computation of the topological watershed. An implementation that was created according to this description showed that a reasonably good speedup could be achieved while using up to 24 threads, and the trend in the results suggests that even better speedups may be achieved when more than 24 threads are used.

The parallel implementation may be improved further by creating a more compact component tree in the component tree construction or compressing it more in the tree compression stage. For example, each node in the component tree that has only one child can be merged with that child, setting all pixels belonging to its component to the gray level of its child, and setting the parent of its child to be the parent of the node itself. This simple improvement could significantly reduce the LCA preprocessing time, and may also have positive effects on the wall-clock time of the last stage where the topological watershed is computed.
Another improvement could be to use a more sophisticated method of parallel list ranking. Better performing algorithms already exist, the algorithm in this report was used for simplicity reasons only. A list ranking algorithm that parallelizes better and has higher speedups, may significantly reduce the total wall-clock time of the algorithm, especially when using larger numbers of threads.
Furthermore, the image is now divided into tiles which are each processed by their own thread. This division is only based on the image dimensions and the number of threads, not on the contents of the image. Taking into account the contents of the image, maybe in combination with the component tree and component map, while dividing the image among the threads may reduce the communications needed between the different threads, which could lead to a faster algorithm.

In short, there is still room for improvement in the parallel algorithm proposed in this report, but in its current form it can already be used to greatly speed up the computation of the topological watershed, compared to the previously existing sequential algorithm.

Bibliography

[1] J. Angulo and D. Jeulin. Stochastic watershed segmentation. In *Intern. Symp. on Mathematical Morphology*, volume 8, pages 265–276, 2007.

[2] G. Bertrand. On topological watersheds. *Journal of Mathematical Imaging and Vision*, 22(2):217–230, 2005.

[3] S. Beucher and C. Lantuéjoul. Use of watersheds in contour detection. In *International Workshop on image processing, real-time edge and motion detection/estimation*, pages 17–21, 1979.

[4] M. Couprie and G. Bertrand. Topological grayscale watershed transformation. In *SPIE Vision Geometry V Proceedings*, volume 3168, pages 136–146. Citeseer, 1997.

[5] M. Couprie, L. Najman, and G. Bertrand. Quasi-linear algorithms for the topological watershed. *J. Math. Imag. Vis.*, 22:231–249, 2005.

[6] R. Jones. Connected filtering and segmentation using component trees. *Computer Vision and Image Understanding*, 75(3):215–228, 1999.

[7] L. Najman and M. Couprie. Watershed algorithms and contrast preservation. In *Discrete geometry for computer imagery*, pages 62–71. Springer, 2003.

[8] L. Najman and M. Couprie. Building the component tree in quasi-linear time. *IEEE Trans. Image Proc.*, 15:3531–3539, 2006.

[9] J.B.T.M. Roerdink and A. Meijster. The Watershed Transform: Definitions, Algorithms and Parallelization Strategies. *Fundamenta Informaticae*, 41:187–228, 2001.

[10] P. Salembier, A. Oliveras, and L. Garrido. Antiextensive connected operators for image and sequence processing. *IEEE Transactions on Image Processing*, 7(4):555–570, 1998.

[11] B. Schieber and U. Vishkin. On finding lowest common ancestors: simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.

[12] R.E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[13] L. Vincent and P. Soille. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *IEEE transactions on pattern analysis and machine intelligence*, 13(6):583–598, 1991.

[14] M. H. F. Wilkinson, H. Gao, W. H. Hesselink, J. E. Jonker, and A. Meijster. Concurrent computation of attribute filters using shared memory parallel machines. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(10):1800–1813, 2008.

[15] James C. Wyllie. The complexity of parallel computations. Technical report, Ithaca, NY, USA, 1979.