Chapter 1

Introduction

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

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

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

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

Many algorithms for various morphological image operators have been published. The aim of this thesis is to speed up some of these algorithms by means of improvements in the original algorithms, or by the use of parallel computing techniques. Nowadays, many desktop workstations contain multiple CPUs, which could be exploited by parallelization of existing algorithms for morphological operators. Since these machines become more common, the focus is on par-
allelization for shared memory or SMP architectures (see section 1.2), which can be desktop workstations with multiple CPUs as well as massively parallel shared memory supercomputers.

1.1 Introduction to Mathematical Morphology

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

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

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

1.1.1 Operator Classification

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

- **Point operators**: the output value of a pixel only depends on the input value of that pixel. Typical operators of this type are taking the complement, union, or intersection. These operators have a time-complexity which is linear in the number of pixels. Parallelization of point operations is trivial, since the domain of the image can be partitioned in as many (almost) equally sized subdomains as there are processors, and each processor can apply the operator to one of these subdomains without having to interact with any other processor. Thresholding is an example of this type of operator. Point operators are not discussed in the rest of this thesis.
• **Local operators:** the output value of a pixel depends on the input values in the neighborhood of that pixel. A typical operator of this type is a binary edge detector, which sets the output value of each foreground pixel (or 1-pixel) which is completely surrounded by 1-pixels to background (or 0-pixel), while 1-pixels which have a 0-pixel as neighbor remain 1-pixels (0-pixels remain 0-pixels). Local operators generally have a time-complexity which is linear in the number of pixels multiplied by the size of the neighborhood under consideration. These operators are harder to parallelize than point operators, but usually the parallelization strategy is the same. However, at the boundaries of the subdomains (partitions) processors do have to communicate and synchronize.

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

### 1.1.2 Dilation and Erosion

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

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

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

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

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

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

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

$$\check{Y} = \{-y \mid y \in Y\}$$
• Erosion is in general not commutative: \( \varepsilon_B(A) \neq \varepsilon_A(B) \), or equivalently \( A \oplus B \neq B \oplus A \).

• Dilation is associative: \( \delta_{\delta_C(B)}(A) = \delta_C(\delta_B(A)) \), or equivalently \( A \oplus (B \oplus C) = (A \oplus B) \oplus C \), for any sets \( A, B, \text{ and } C \).

• Translation invariance: \( \delta_B(A_x) = (\delta_B(A))_x \), and \( \varepsilon_B(A_x) = (\varepsilon_B(A))_x \), or equivalently \( A_x \oplus B = (A \oplus B)_x \), and \( A_x \oplus B = (A \oplus B)_x \).

• Dilation and erosion are not inverses of each other: \( \delta_B(\varepsilon_B(A)) \neq A \neq \varepsilon_B(\delta_B(A)) \).

• Dilation distributes over union: \( \delta_B(\bigcup_{i=0}^n A_i) = \bigcup_{i=0}^n \delta_B(A_i) \).

• Erosion distributes over intersection: \( \varepsilon_B(\bigcap_{i=0}^n A_i) = \bigcap_{i=0}^n \varepsilon_B(A_i) \).

• The erosion of a set \( A \) by the union of two sets \( B \) and \( C \), is the same as the intersection of the erosion of \( A \) by \( B \), and the erosion of \( A \) by \( C \). This property can be generalized to \( \varepsilon_{C \cup B_i}(A) = \bigcap_{i=0}^n \varepsilon_B(A) \).

• Repeated erosion of a set \( A \) by sets \( B_0, \ldots, B_n \), is the same as the erosion of \( A \) by the dilation of the sets \( B_0, \ldots, B_n \): \( \varepsilon_{B_0}(\varepsilon_{B_1}(\ldots(\varepsilon_{B_n}(A)) \ldots)) = \varepsilon_{B_0 \oplus B_1 \oplus \ldots \oplus B_n}(A) \).

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

Opening of X by Y

Closing of X by Y

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

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

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

1.1.3 Opening and Closing

We already know that erosion and dilation are not each other’s inverse operations. This is easy to see if we take an object like a disc with a very small hole (let us say, only one pixel) in its interior, and we dilate the disc with a smaller disc. The result will be that the disc grows in
size, but also the hole is filled. After erosion this disc will shrink again, but the small hole has disappeared. For this reason, this sequence of operations is called a structural closing. The dual operator, erosion followed by dilation, is called a structural opening. The structural opening of a set \( X \) with a structuring element \( Y \) is denoted by \( O_Y(X) \), while the corresponding closing is denoted by \( C_Y(X) \). In the case of infix notation we use the symbols ‘\( \circ \)’ and ‘\( \bullet \)’, respectively.

\[
O_Y(X) = X \circ Y = \delta_Y (\varepsilon_Y (X)) \\
C_Y(X) = X \bullet Y = \varepsilon_Y (\delta_Y (X))
\]

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

\[
O_Y(X) = \bigcup_{h, Y_h \subseteq X} Y_h \\
C_Y(X) = \bigcap_{h, X \subseteq (Y^c)_h} (Y^c)_h
\]

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

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

\[
O_Y(X) = (C^c_Y(X^c))^c, \quad X \circ Y = (X^c \bullet Y)^c
\]

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

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

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

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

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

1.1.4 Some Operators Based on Dilation and Erosion

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

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

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

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

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

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

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

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

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

1.1.5 Distance Transform

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

A grey-scale image is a function $f : D \rightarrow R$, where $D \subseteq \mathbb{R}^n$, and $R \subseteq \mathbb{R}$ is a set of grey values. A discrete (digital) grey-scale image is a grey-scale image for which $D \subseteq \mathbb{Z}^n$ and $R \subseteq \mathbb{N}$, for which there is a neighboring relation defined on the set $D$, as in the case of binary images. From now on, we will omit the prefix discrete (or digital) and simply refer to grey-scale images.
Figure 1.5. Noise removal using conditional dilation: let \( M \) be the noisy image, \( B \) a small disc, and \( A = \varepsilon_B(M) \). The conditional dilation \( \delta_{B,M}(A) \) is shown on the right.

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

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

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

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

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

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

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

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

\[
DT(p) = d(p, A^c)
\]

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

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

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

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

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

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

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

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

\[
S_{B,k}(A) = e_{\oplus_k B}(A) \setminus O_B(e_{\ominus_k B}(A)) = (A \ominus (\oplus_k B)) \setminus ((A \ominus (\oplus_k B)) \circ B), \quad k \in \mathbb{N}.
\]

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

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

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

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

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

\[
sk(p) = \begin{cases} 
  k + 1 & \text{if } p \in S_{B,k}(A) \\
  0 & \text{otherwise}
\end{cases}
\]
Figure 1.6. Top: a one-dimensional function $f$ (left), and its umbra $U(f)$ (right). Middle: a structuring element $g$ (left), and its umbra $U(g)$ (right). Bottom: binary dilation $U(f) \oplus U(g)$ (left), and the grey-value dilation $T(U(f) \oplus U(g))$ (right).

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

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

1.1.7 Grey Scale Morphology

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For example, the grey-scale opening (\(\ominus\)) and closing (\(\bullet\)) are defined as (see Fig. 1.7):
\[
    f \ominus g = (f \ominus g) \ominus g \\
    f \bullet g = (f \bullet g) \ominus g
\]

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

1.2 Introduction to Parallel Computing

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

1.2.1 Distributed Memory Machines

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

1.2.2 Shared Memory Architectures

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

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

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

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

```plaintext
var x: integer; (* initially x = 1 *)

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

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

Let us assume that each assignment is an *atomic* (or *indivisible*) operation, i.e. it does not interfere with any other statement. This way, a concurrent execution of the program can be regarded as a sequential interleaving of the statements. If we label the three assignments of thread P with a, b, and c, and those of thread Q with d, e, and f, then each interleaving of these letters in which a precedes b, b precedes c, d precedes e, and e precedes f, is a possible execution of the
Such a possible execution is called a *trace*. The number of possible traces increases very rapidly with the number of statements of a program. This simple example consisting of six assignments gives rise to no less than 20 traces, yielding four possible outcomes. After execution of the program the value of the shared variable $x$ can be 2, 3, 4, or 6, depending on the order of execution of each thread, as can be seen in the following table:

<table>
<thead>
<tr>
<th>trace</th>
<th>$x$</th>
<th>trace</th>
<th>$x$</th>
<th>trace</th>
<th>$x$</th>
<th>trace</th>
<th>$x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcdedf</td>
<td>6</td>
<td>abdcf</td>
<td>3</td>
<td>dabefc</td>
<td>3</td>
<td>adefbc</td>
<td>2</td>
</tr>
<tr>
<td>defabc</td>
<td>4</td>
<td>adbcf</td>
<td>3</td>
<td>daefbc</td>
<td>2</td>
<td>adbefc</td>
<td>2</td>
</tr>
<tr>
<td>abdefc</td>
<td>3</td>
<td>debcf</td>
<td>3</td>
<td>dabefc</td>
<td>2</td>
<td>adebfcc</td>
<td>2</td>
</tr>
<tr>
<td>abdfce</td>
<td>3</td>
<td>deabfc</td>
<td>2</td>
<td>daefbc</td>
<td>2</td>
<td>deafbc</td>
<td>2</td>
</tr>
</tbody>
</table>

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

### 1.2.2.1 Mutexes

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

### 1.2.2.2 Condition Variables

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

Two operations are allowed on a condition variable: `wait` and `signal`. The `wait` operation suspends the calling thread, while a `signal` operation wakes a thread which is suspended on the condition variable. A condition variable is associated with a mutex, to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it. A thread that owns the mutex (i.e. it has locked the mutex), can decide upon inspection of shared data that it cannot continue. In this case, it suspends itself by calling `wait(c, m)`, where $c$ denotes the condition variable, and $m$ the associated mutex.
A thread is only allowed to call \texttt{wait} on a condition variable when it owns the associated mutex. Upon suspension of a thread, the mutex \textit{m} is unlocked. Unlocking the mutex and suspending on the condition variable is done atomically. This guarantees that the condition cannot be signaled between unlocking the mutex and starting to wait on the condition variable.

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

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

1.2.2.3 Barriers

Using mutexes and condition variables it is easy to build another very useful synchronization primitive, called a \textit{barrier}. A barrier is a so-called collective operation, i.e. all threads take part in it. A barrier is a routine which gets called by all threads. Each thread, except the last, gets suspended. When the last thread arrives, all other threads are released. A barrier is typically a
useful primitive in algorithms that consist of a sequence of stages. Each stage can be executed concurrently, however a next stage should not be started until all threads completed the current one. A typical example of a situation where a barrier can be useful is found in chapter 2, where an algorithm for distance transforms is discussed. The algorithm consists of two phases, that need to be separated by a barrier.

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

1.2.2.4 Semaphores

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

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

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

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

\[
S_p = \frac{T_1}{T_p}.
\]

Note that the times \( T_i \) are so called wall clock times in seconds. Wall clock time is simply the execution time of the program measured using e.g. a stopwatch. Note that this differs from what is generally called cpu-time, which is the accumulation of execution time over all processors. A concurrent algorithm that keeps 2 CPUs active during execution has a cpu-time time that is twice as large as its wall clock time.
Ideally, the speedup obtained by using \( p \) CPUs is \( p \). This is called linear speedup. In practice, linear speedup is hardly ever reached, since parallel programs spend time on synchronization, communication, or contain sequential sections.

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

\[
E_p = \left( \frac{S_p}{p} \right) \times 100\%.
\]

1.3 An Example: Concurrent dilation

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

\[
\text{var } \text{im}[0..M-1,0..N-1] \text{ of integer;}
\]

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

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

\[
X \oplus \{(i, j) | -1 \leq i, j \leq 1\} = X \oplus \{(i, 0) | -1 \leq i \leq 1\} \oplus \{(0, j) | -1 \leq j \leq 1\}.
\]

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

Just like in the naive algorithm a mutex is used. However this mutex is not used to guard accesses to the pixels of the image, but guards the variables \( \text{row} \) and \( \text{col} \). These variables are
1.3 An Example: Concurrent dilation

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

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

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

Figure 1.11. Code fragment of a concurrent algorithm implementing a binary dilation with structuring element $C_8 = \{(i,j)\mid -1 \leq i, j \leq 1\}$. 
much less frequently accessed, so the chance of several threads accessing the same mutex at the same time has reduced significantly. Besides, the critical section guarded by the mutex is very small. It consists of only two assignments, while processing of 8 neighboring pixels in a critical section takes more time. The smaller a critical section, the less time a thread spends in it. Hence, smaller critical sections lead to a smaller chance on contention.

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

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

1.4 Thesis organization

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

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

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

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

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

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