Masters thesis

Distributed Determination of Connected Components

M. Sinnema

Supervisors:
W.H. Hesselink and A. Meijster

Rijksuniversiteit Groningen
Computer Science
Postbus 800
9700 AV Groningen

August 2001

Abstract

An important task in image processing is the labelling of connected components, which is a basic segmentation task. In this report we show how we parallelized Tarjan's disjoint set algorithm for determination of connected components on distributed memory systems, e.g. a set of desktop computers connected via a network.
We first give a sequential and a parallel solution for Tarjan's disjoint set algorithm. Secondly we show how to implement both algorithms. We also study the scalability of the algorithm.

Contents

Chapter 1

Introduction

This chapter introduces images and parallelism. In the literature many different ideas and notations have been used for these terms. In order to avoid misunderstanding, most definitions and notations are presented here.

1.1 Images

The central object in image processing is the *image*. We represent an image as a function from a certain domain D to a range E, i.e.

$$\texttt{image} :: D \to E.$$

In this report the range E is the set $\mathbb{N}$ of natural numbers. With minimal modifications, however, all algorithms and ideas also apply to other ordered ranges. An example of an image is a digital grey-scale photo, where E is the set of possible luminances of a pixel. In this example, the domain D is a square subset of $\mathbb{N} \times \mathbb{N}$, and $\texttt{image}[(x,y)]$ is the luminance of pixel (x,y).
In most cases, domain D is a subset of $\mathbb{N}^k$, where k is the dimension of the image. In figure 1.1 we show some examples of images of different dimensions. Image a. is a plot of a sound signal, image b. is one 2D slice of a ct scan, and image c. is a 3D volume of a ct scan.
In this report an image is a function from a domain D to the range $\mathbb{N}$, where $D \subset \mathbb{N}^k$.

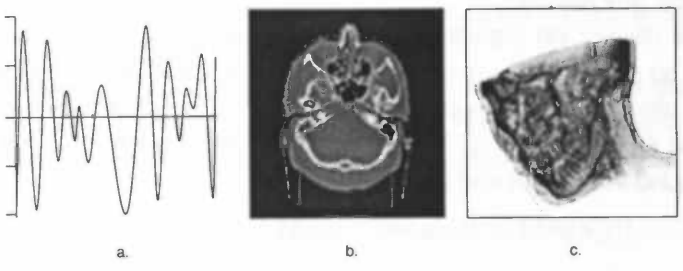

Figure 1.1: Some examples of images of different dimensions.

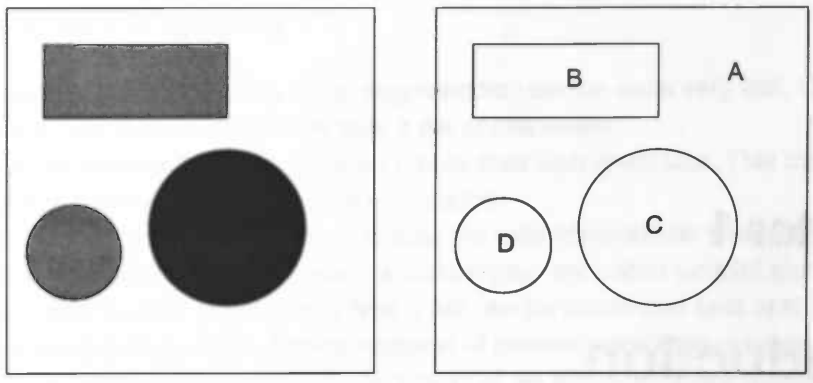

Figure 1.2: Connected component labelling

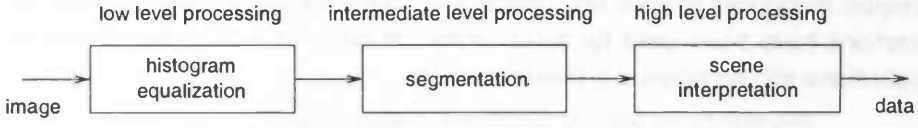

Figure 1.3: Levels in image processing

1.2 Segmentation

The main issue in this paper is the labelling of the *connected components* of an image, which is a primitive type of segmentation. In figure 1.2 an example of such a segmentation is shown. The left image is the original image, and the right image shows a labelling of the connected components. Each pixel is labelled with some value, equal to the other pixels in its connected component. In chapter 2 we give a more precise definition of a labelling.

Traditionally we distinguish three different levels of image processing, as shown in figure 1.3. In this figure an example of each level is shown in the box. Low level processing is performed at the pixel-level, like enhancing the image, which can be sharpening, histogram equalization, smoothing, etc. Intermediate level processing can be the detection of edges or connected components. With high level processing we mean recognition and interpretation of the objects in the scene of the image. This report focuses on the intermediate level processing.

We use Tarjan's disjoint set algorithm to label the connected components. This algorithm is based on the idea of region merging, and is presented in chapter 2. There are many other algorithms designed for segmentation, e.g. the detection of the borders between objects and background. More information about other methods for segmentation can be found in [Son99] and [Roe98].

Connectivity

For an image of dimension k, we define that two pixels $x, y \in D$ are *directly connected* iff $\texttt{image}[x]$ is equal to $\texttt{image}[y]$, and x and y are neighbours. Whether two pixels are neighbours depends on which connectivity is used. This connectivity is defined by a symmetrical set of vectors $S \subset \mathbb{Z}^k$. The set of neighbours $Nb(p)$ of pixel p is defined as

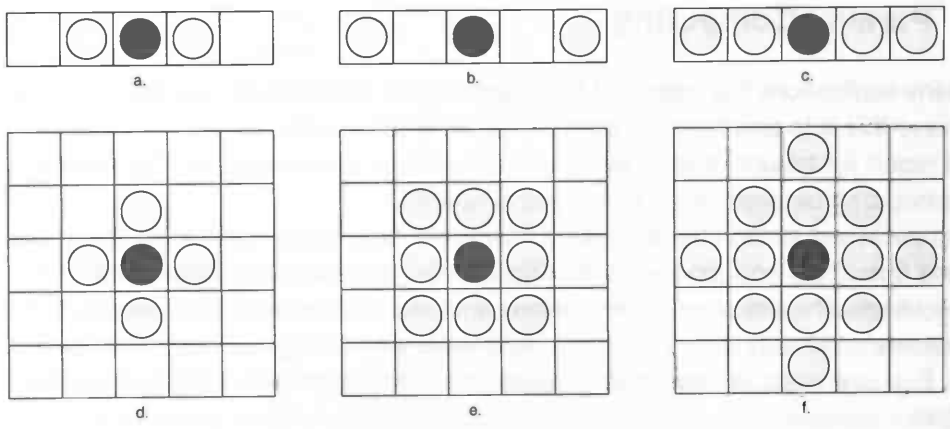

Figure 1.4: Some examples of connectivity. The light pixels are the neighbours of the dark pixel in the middle.

$$Nb(p) = D \cap (p + S),$$

where $p + S = \{p + s | s \in S\}$. Now x and y are neighbours iff $x \in Nb(y)$, which is equivalent to $y \in Nb(x)$, since S is symmetrical.
In figure 1.4 some examples are shown. The figures 1.4.a to 1.4.c are 1D examples, and the figures 1.4.d to 1.4.f are 2D examples.
If we denote S_i for the set vectors S in figure 1.4.i, we have

$$\begin{aligned}
S_a &= \{(-1), (1)\}, \\
S_b &= \{(-2), (2)\}, \\
S_c &= S_a \cup S_b, \\
S_d &= \{(u,v) \in \mathbb{Z}^2 \mid |u| + |v| = 1\}, \\
S_e &= \{(u,v) \in \mathbb{Z}^2 \mid 1 \leq |u| + |v| \leq 2 \wedge |u| \leq 1 \wedge |v| \leq 1\}, \text{ and} \\
S_f &= \{(u,v) \in \mathbb{Z}^2 \mid 1 \leq |u| + |v| \leq 2\}.
\end{aligned}$$

The connectivity S_d is also known as 4-connectivity, and S_e is also known as 8-connectivity.

Connected components

A formal description of a *connected component* can be found in chapter 2. Intuitively, two pixels x and y belong to the same connected component iff there exists a path from x to y on which all `image` values are equal.
In order to make reasoning about connected components easier, we define the undirected graph $G = (D, E)$, where D is the domain of the function `image`, and E is the set of pairs of pixels which are pairwise directly connected, i.e.

$$E = \{(x,y) \in D \times D \mid x \in Nb(y) \wedge \texttt{image}[x] = \texttt{image}[y]\}.$$

In chapter 2 is shown how the connected components are labelled by Tarjan's disjoint set algorithm.

1.3 Parallel Computing

For many applications it is important that segmentation can be done very fast. One way to achieve this is to distribute the work over a set of processes.
In this report we assume that all processes run on their own processor. This means no task scheduling between these processes is needed.
The processes work together in order to reduce the total computation time. Algorithms that use more than one process to reach a certain goal are called parallel algorithms. The problem with parallel algorithms is how a job can be distributed best and how the processes should communicate. The correctness of parallel algorithms requires special care. E.g. we need to show that parallelization of an algorithm does not introduce *deadlock*.
In this section we give definitions about the use and notation of parallel algorithms. In this report, the set of processes is denoted by *Processes*.

Communication

In this report, all communication between processes is done by messages.
A process can send a message to another process or to itself. When a message is sent, the sending process executes the next statement without waiting for receipt, which is known as non-blocking. The receiving operation is blocking. If the receiving process wants to receive a message, but no message has arrived yet, the process waits until a new message arrives. Then the next statement is executed.
We denote the sending and receiving of messages as follows

send *amsg* **to** y;
receive *amsg*,

where *amsg* is the kind of message. We denote the kind of message as the message type, or shortly, the *type* of the message.
Messages can have arguments with specific information. The notation in pseudo code for sending a message with arguments is

send $amsg(s,t)$ **to** y,

which means sending a message of type *amsg* with arguments s and t to process y. s and t can be of any type. After the arrival of an *amsg* message, s and t have the values of the arguments of the message.
The receiving of messages can be notated in two different ways:

1. If only one type of message *amsg* can arrive,

   receive $amsg(s,t)$

   is used.

2. If different types of messages are expected, the following notation is used

   in $mtx(a,b) \rightarrow$
   A ;
   [] $mty \rightarrow$

```
            B ;
[]  mtz(t)  →
            C ;
ni.
```

This means that three different types of messages can arrive, namely a *mtx*, a *mty*, or a *mtz* message. At the **in** instruction, the execution stops until a new message arrives. When a message is received, the execution continues depending on the type of message received. Upon arrival of an *mtx* message, which has two arguments, code fragment *A* is executed. The arrival of a *mty* message, with no arguments, results in executing fragment *B*. Finally, a *mtz* message, with just one argument, results in executing code fragment *C*.

An example

We give an example of a simple communication protocol. There are two processes P_a and P_b, and there are three types of messages, namely mX, $mY(j)$ and mZ.
First, P_a sends an mX message to P_b. Then it sends a mY message with argument 1 to P_b, and waits for two mZ messages to arrive. P_b waits for two messages to arrive. If an mX message arrives, P_b prints the character `A` to the output, and it sends a mZ message to P_a. If a $mY(j)$ message arrives, P_b prints j to the output, and sends a mZ message to P_a. The pseudo-code of the fragment for process P_a is

```
Pa:   send mX to Pb;
      send mY(1) to Pb;
      receive mZ;
      receive mZ,
```

and for process P_b

```
Pb:   for i := 1 to 2 do
        in  mX  →
              print 'A';
              send mZ to Pa;
        []  mY(j)  →
              print j;
              send mZ to Pa;
        ni;
      od.
```

Because the time between sending a message and receiving the message is unknown, and the fact that sending is non-blocking, messages can arrive in any order. Therefore, execution of this parallel algorithm can result in the outputs `A1` or `1A`, depending on the order in which the messages sent by P_a arrive at P_b.

Chapter 2

A sequential Union-Find algorithm for determination of components

In this chapter a variation of Tarjan's disjoint set algorithm for the determination of equivalence classes is presented. We give a formal description of the concepts that have been presented in chapter 1.

2.1 Problem description

Let f be an image. So, $f : D \to \mathbb{N}$ is a function from a finite domain $D \subset \mathbb{N}^k$. Let $S \subset \mathbb{Z}^k$ be a set of "neighbour vectors", which define a connectivity as follows.
The neighbours of a point $p \in D$ are the points $p+q$ with $q \in S$ and $p+q \in D$. We denote the set of neighbours of p with $Nb(p)$, i.e.

$$Nb(p) = D \cap (p+S).$$

A finite sequence of points $[x_1,..,x_n]$ is called an iso-level path from p to q iff $p = x_1$, $q = x_n$ and $x_{i+1} \in Nb(x_i)$ for $1 \leq i < n$ and $\forall(i : 1 < i \leq n : f(x_i) = f(x_1))$. The existence of such a path is denoted by $\pi(p,q)$.
A set $X \subseteq D$ is called a connected set iff $\forall(p,q \in X :: \pi(p,q))$, which is denoted by $Conn(X)$. A connected set X is called a *connected component* if the set is maximal, which is denoted by $CC(X)$, i.e.

$$CC(X) \quad \equiv \quad \forall(Y : Conn(Y) \wedge X \subseteq Y : X = Y).$$

Clearly, the connected components of an image partition the domain D. A function $lab : D \to \mathbb{N}$, which assigns a unique identification to each connected component, is called a labelling. This means that, for all connected components X and all $p \in X$ and $q \in D$, we require

$$lab(p) = lab(q) \quad \equiv \quad q \in X.$$

The problem we study in this report is to determine a labelling for a given input image f.

2.2 Tarjan's disjoint set algorithm

We first describe Tarjan's original algorithm. After that we show how to apply it to images of arbitrary dimensions.

Description of the algorithm

The algorithm maintains and modifies a family of disjoint sets. The members of this family are called sets.
For each set an arbitrary member is chosen as representative for that set. This element is called the *canonical element*. There are three basic operations.

- *MakeSet*(x), which creates a new singleton set $\{x\}$. This operation assumes that x is not a member of any other set.
- *Find*(x), which returns the canonical element of the set containing x.
- *Union*(x,y), which forms a new set that is the union of the two sets that contain x and y. This operation assumes that x and y are not in the same set.

Tarjan uses tree structures to represent sets. Each non-root node in a tree points to its parent, while the root of a tree points to itself. Two objects x and y are members of the same set if and only if x and y are nodes of the same tree, which is equivalent to saying that they share the same root of the tree they are stored in. Because canonical elements may be chosen arbitrarily, it is convenient to choose the root nodes. In this case, the *Find* operation reduces to finding the root node of a tree, and is therefore called *FindRoot*.
We assume that the elements that we want to store in the sets are integers from a bounded range (for a finite set of any type, we can always find such a mapping by enumeration, so this is not a restriction). The trees are implemented in a linear array, named `parent`, of which the indices are simply the elements of the trees themselves. The value $\texttt{parent}[x]$ is the parent of x in the tree x is contained in. When x is a canonical element, we have $\texttt{parent}[x] = x$.

Time complexity

Obviously, the operation *MakeSet*(x) can be performed in constant time, but the operations *FindRoot*(x), and *Union*(x,y) require a search for the canonical element of x and y. The canonical element of x is found by traversing the tree towards the root. Clearly, this operation requires time which is linear in the length of the path from x to its canonical element. Therefore, the operation *FindRoot* requires less time if we can reduce the length of these paths. Tarjan uses two important techniques to keep these paths reasonably short.
The first technique is called *path compression*. Every time the operation *FindRoot*(x) is applied, the parent pointers of the nodes on the *root-path* (the path from x to the root of the tree) are changed to point directly to the root of the tree. Thus, after performing the operation *FindRoot*(x), a second operation *FindRoot*(y), with y on the root-path of x, takes constant time.
The second technique is called *union by rank*. This technique is used in the operation *Union*(x,y). The idea is to make the root of the tree with fewer nodes point to the root of the tree with more nodes. For each node x, a value $\texttt{rank}[x]$ is maintained which is an approximation of the logarithm of the size of the subtree of which x is the root. This rank is also an upper bound on the height of the node in the tree. Note that path compression does not change the rank of the root of a tree, since the size of a subtree does not change.

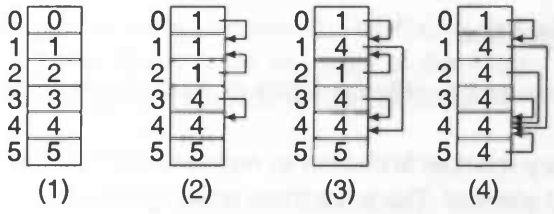

Figure 2.1: An example of how Tarjan's disjoint set algorithm works. The array `parent` is displayed for each situation in the text.

Tarjan shows that in an intermixed sequence of m operations, of which there are n *MakeSet* operations (and hence at most $n-1$ *Union* operations) and f *FindRoot* operations, the path-compression technique results in a worst-case running time of $\Theta(f \log_{1+f/n} n)$, if $f \geq n$, and $\Theta(n+f\log_2 n)$ otherwise. When both path-compression and union by rank is used, the worst case running time is $O(m\alpha(m,n))$, where $\alpha(m,n)$ is the very slowly growing inverse of Ackermann's function. For the exact derivation of these bounds we refer to [Tar75].

Basic operations

The basic operations for maintaining disjoint set can be implemented as follows.

MakeSet(x) :
```
parent[x] := x;
rank[x] := 0;
```

FindRoot(x) :
```
if x ≠ parent[x] then
    parent[x] := FindRoot(parent[x]);
fi;
return parent[x];
```

Link(x,y) :
```
parent[x] := y;
```

Union(x,y) :
```
p := FindRoot(x);
q := FindRoot(y);
if rank[p] > rank[q] then
    Link(q,p);
else if rank[p] < rank[q] then
    Link(p,q);
else
    Link(p,q);
    rank[q] := rank[q] + 1;
fi;
```

Example

To make clear how these basic operations work we give a sequence of basic operations together with its result. The operations below result in the four configurations of array `parent` in figure 2.1.

1. *MakeSet*(0); ...; *MakeSet*(5) : all ranks are set to 0.

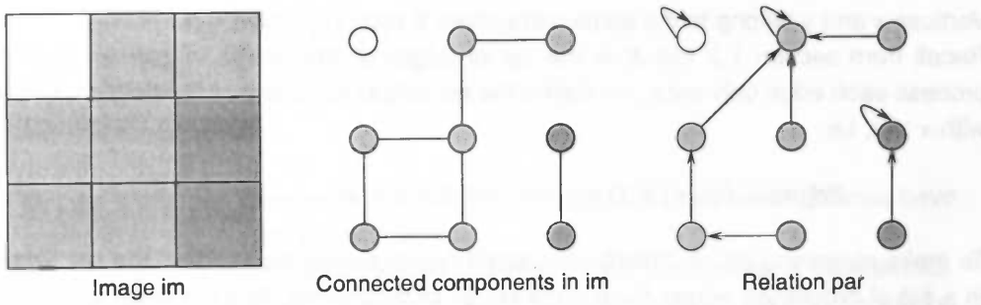

Figure 2.2: Connected component labelling with relation par.

2. *Union*$(0,1)$; *Union*$(1,2)$; *Union*$(3,4)$: First 0 is linked to 1, and rank$[1]$ becomes 1. Next 2 is linked to 1 since rank$[1] >$ rank$[2]$, leaving the ranks unchanged. Finally 3 is linked to 4, and rank$[4]$ is set to 1.

3. *Union*$(1,3)$: The root of 1 is 1, while the root of 3 is 4. Since rank$[1] =$ rank$[4]$, 1 can be linked to 4, and rank$[4]$ is incremented to 2.

4. *Union*$(2,5)$:

   First a *FindRoot* is started in 2, resulting in the root 4. As the result of path-compression 2 points directly to 4. Pixel 5 is its own root. Since rank$[5] <$ rank$[4]$, 5 is linked to 4.

2.3 Design of a sequential algorithm

The disjoint set forest parent is implemented as an array par, which is of type

par : **array** D **of** D.

Since we are interested in an algorithm for images, we want to process the images in scanline order, i.e. a lexicographical order on D, which we simply denote by $\leq$.
We will not use *Union by Rank* since it requires an auxiliary array of the same size of the input, which is generally quite large. Besides, from experiments we found that *Union by Rank* does not pay off in the case of images.
Moreover, it allows to introduce the following invariant, which makes sure that no cycles in the par-relation occur.

J1 $\quad \forall(p \in D :: 0 \leq \text{par}[p] \leq p)$

The goal of the algorithm is to build up the par trees. An example of a representation by par trees can be seen in figure 2.2. On the left, a grey-scale image of size 3×3 is displayed. In the middle we see its connected components, and on the right we see a representation as a par tree that satisfies invariant *J1*.
The *root* of a vertex x is found by successively applying par to x, i.e.

$$\mathit{root}(x) = \textbf{if } \text{par}[x] = x \textbf{ then } x \textbf{ else } \mathit{root}(\text{par}[x])$$

Vertices x and y belong to the same component if $root(x) = root(y)$.
Recall from section 1.2 that E is the set of edges of the image. Since we want to process each edge only once, we define the set *Edges* to consist of the pairs $(x,y) \in E$ with $x > y$, i.e.

$$Edges = \{(x,y) \in D \times D | (x,y) \in E \wedge x > y\}.$$

To make reasoning about already processed edges easier, we partition the set *Edges* in a set of processed edges E_p and the set E_n of edges that have not been processed yet. This is described by the following invariant

J2 $\quad E_p \cup E_n = Edges \quad \wedge \quad E_p \cap E_n = \emptyset.$

The idea is to withdraw an edge from E_n, process it by extending the disjoint set forest, and insert it in E_p. To express this formally we introduce the predicate $\pi_F(p,q)$, which denotes that there exists an iso-level path from p to q using only edges from the set F. Using this predicate we can now define the invariant

J3 $\quad \forall(x,y \in D :: \quad \pi_{E_p}(x,y) \quad \equiv \quad root(x) = root(y)).$

The invariants are initialized by the following fragment

Tarseq*init*: **for all** $x \in D$ **do**
$\quad \mathtt{par}[x] := x;$
od;
$E_n := Edges;$
$E_p := \emptyset;$

In the main fragment Tarseq*main*, repeatedly an edge in E_n is processed, and is moved from E_n to E_p. This preserves invariant *J2*. When the set E_n is empty, all edges have been processed.

Tarseq*main*: **while** $E_n \neq \emptyset$ **do**
$\quad$ **choose** $(x,y) \in E_n$;
$\quad$ Extend(x,y);
$\quad E_p := E_p \cup \{(x,y)\};$
$\quad E_n := E_n \setminus \{(x,y)\};$
od;

The function Extend should update `par` if necessary, in order to maintain invariant *J3*. Note that the edges can be processed in any order. Most implementations use a raster scan order. We found it more efficient, however, to use an anti-raster scan order. This means that at any time the edges (x,y) with the largest y in E_n are processed next. For every y, the edges (x,y) can be processed in arbitrary order of x. The order of the processing of the edges is described in the following fragment

Tarseq: **for all** $y \in D$ **in decreasing order do**
$\quad$ **for all** $x \in D$ **such that** $(x,y) \in E_n$
$\quad\quad$ Extend(x,y);
$\quad\quad E_p := E_p \cup \{(x,y)\};$

$E_n := E_n \setminus \{(x,y)\}$;
od;
od.

Note that the **while** and **choose** statements from the previous definition of Tarseq have been replaced by the two **for all** statements.
What remains now is the implementation of Extend(x,y), which has to maintain *J3*. This is done by joining the sets of x and y. Thus, after Extend(x,y), *root*(x) should be equal to *root*(y).
From the order, the definition of *Edges*, and invariant *J1*, we can conclude that Extend(x,y) must preserve $\mathtt{par}[y] = y$. Thus, we only have to compute *root*(x) instead of computing *root*(x) and *root*(y). Joining the sets that contain x and y can be accomplished by setting $\mathtt{par}[root(x)]$ to y. In order to find the root of vertex x we introduce the following fragment

FindRoot(x): **while** $\mathtt{par}[x] \neq x$ **do**
$x := \mathtt{par}[x]$;
od;
return x ;

Thus Extend(x,y) can be defined as follows:

Extend(x,y): $\mathtt{par}[\text{FindRoot}(x)] := y$;

The algorithm above suffices to maintain all invariants, however we use path compression to achieve better efficiency. Since we know that the root of x will be linked to y we can incorporate FindRoot, path compression, and linking in the following version of Extend

Extend(x,y): **do**
$p := \mathtt{par}[x]$;
$\mathtt{par}[x] := y$;
$x := p$;
while $\mathtt{par}[x] \neq y$;

Note that as a result of the ordering imposed by invariant *J1* and the fact that we use an anti-raster scan algorithm, memory references are likely to be very local. This is especially important on systems that utilize memory caches. Path compression increases the profit of this locality even more.

An example

In figure 2.3 an example of the construction of the disjoint sets is shown. The dotted line is the boundary between the vertices incident with processed edges, and those not incident with processed edges. The arrows represent the par values of the vertices.

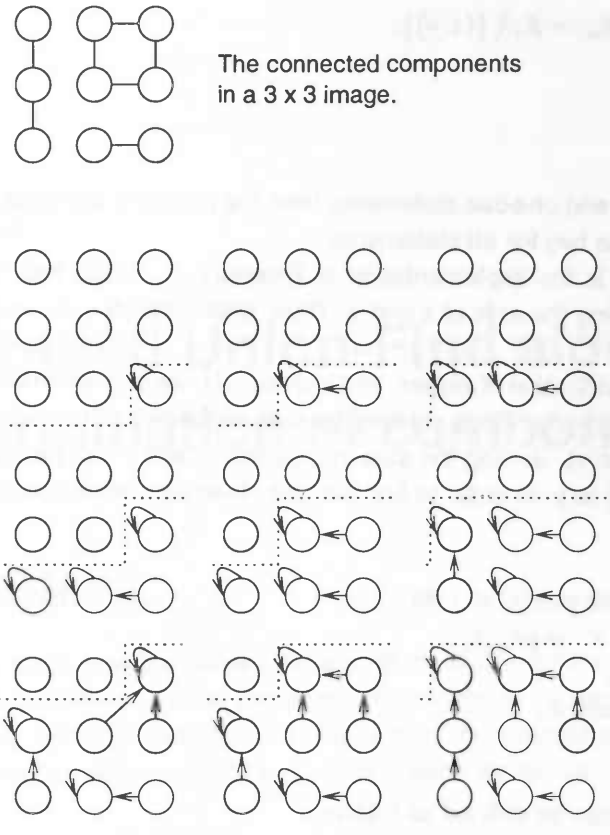

Figure 2.3: An example of how in Tarseq the disjoint sets in a 3×3 image are constructed.

2.4 Harvest

Recall from section 2.1 that each vertex has to be labelled with an identification label. We use the *root* of a vertex to be the label `lab` of the vertex, i.e. for all $x \in D$

$$\texttt{lab}[x] = \mathit{root}(x).$$

In fact this labelling is obtained by simply performing $\texttt{lab}[x] :=$ FindRoot(x) for all $x \in D$. Note that this is the final path compression of `par`.
However, we can do this more efficiently by using invariant *J1* and a raster scan algorithm. This leads to the following fragment

Harvseq: **for all** $x \in D$ **do in increasing order**
  if $\texttt{par}[x] = x$ **then**
    $\texttt{lab}[x] := x$;
  else
    $\texttt{lab}[x] := \texttt{lab}[\texttt{par}[x]]$;
  fi
od .

Clearly algorithm Harvseq is efficient, i.e. of order $O(\#D)$.

This harvest algorithm yields, together with the disjoint set algorithm, the following fragment which labels the connected components in an image

```
Labelseq :  Tarseqinit;
            Tarseq;
            Harvseq;.
```

Chapter 3

A distributed Union-Find algorithm for determination of components

3.1 Introduction

In this chapter we show how the disjoint set algorithm, of chapter 2 can be distributed over a number of processes. These processes communicate by means of message-passing. The idea is to distribute the set *Edges* over the processes. Therefore we define a function *owner*, which assigns a process to each vertex, i.e.

$$owner :: D \rightarrow Processes.$$

A process k can only inspect and update $\mathtt{par}[x]$ if $owner(x) = k$.
In order to distribute the edges over the processes we define a partition on the set *Edges* as follows

$$Edges(k) = \{(x,y) \in Edges \mid owner(x) = k\},$$

for each $k \in Processes$. Process k can only inspect the set $Edges(k)$.
For each $k \in Processes$ the set $Edges(k)$ is partitioned into the sets

$$InEdges(k) = \{(x,y) \in Edges(k) \mid owner(y) = k\}$$

which are the edges in $Edges(k)$ to a vertex that belongs to process k, and

$$OutEdges(k) = \{(x,y) \in Edges(k) \mid owner(y) \neq k\}$$

which are the edges in $Edges(k)$ to a vertex not belonging to process k.

3.2 Sequential processing

First, each process k applies Tars_{eq} to the set $InEdges(k)$. The invariant *J2* has to be redefined for this parallel situation

J2 $\qquad E_p(k) \cup E_n(k) = InEdges(k) \quad \wedge \quad E_p(k) \cap E_n(k) = \emptyset$

for each process k. $E_p(k)$ is the set processed edges of *InEdges*(k), and $E_n(k)$ is the set non-processed edges of *InEdges*(k). Now *J3* can be redefined as

J3 $\quad \forall(x,y \in D :: \exists(k :: \pi_{E_p(k)}(x,y)) \equiv root(x) = root(y))$.

From *J3* and the partition on *Edges* we can conclude that after Tar$_{seq}$

$$\forall(x,y \in D :: \exists(k :: \pi_{InEdges(k)}(x,y)) \equiv root(x) = root(y)).$$

3.3 Parallel processing

What remains now is the processing of the edges in *OutEdges*(k) for each k. The idea is to use the *Union*(x,y) fragment from section 2.2 to join all sets of x and y, where $(x,y) \in$ *OutEdges*(k). By the definition of *OutEdges*(k), process k can not inspect par$[y]$, which is needed to find the root of y. Process k might even not be the owner of the root of x.
We define the set F as the set of edges that have been processed, and we treat *OutEdges*(k) as a program variable of process k. We postulate the invariant

J4 $\quad F \quad \cup \quad \bigcup(k \in Processes :: OutEdges(k)) = Edges$

to hold while the disjoint sets in parallel are constructed. Invariant *J3* is now restated with the use of F.

J3 $\quad \forall(x,y \in D :: \pi_F(x,y) \equiv root(x) = root(y))$

After Tar$_{seq}$, *J4* is easily initialized by the statement

Tar$_{parinit}$: $\quad F := \bigcup(k \in Processes :: E_p(k))$.

The idea is that each process k withdraws edges from *OutEdges*(k), processes it in order to maintain *J3*, and moves the edge from *OutEdges*(k) to F, i.e.

Tar$_{parmain}$:
```
while OutEdges(k) ≠ ∅ do
    choose (x,y) ∈ OutEdges(k);
    Extend(x,y);
    F := F ∪ {(x,y)};
    OutEdges(k) := OutEdges(k) \ {(x,y)};
od;.
```

In the first two statements of *Union*(x,y) a *FindRoot* is done for both x and y. In Extend(x,y), the roots of x and y are searched for simultaneously by the fragment Search below.
The invariant

J5 $\quad y \leq x$

should remain valid while searching for the largest root. Of course, when $x = y$ the trees are already linked. Then *Search* has to do nothing.

```
Search(x,y):  if par[x] ≠ x ∧ x ≠ y then
                  x := par[x];
                  if x < y then x,y := y,x fi;
                  Search(x,y);
              fi;
```

The Extend(x,y) fragment first calls Search(x,y), which preserves invariant *J5*. It is easy to see that after Search, x and y can be linked by the statement $\mathtt{par}[x] = y$. Without considering the fact that the array par is distributed, we can define

```
Extend(x,y):  Search(x,y);
              if x ≠ y then
                  par[x] = y;
              fi,
```

which maintains all invariants. Note that in *Extend* the trees are only linked when $x \neq y$, and $\mathtt{par}[y]$ is not always equal to y.
In the following fragments, algorithms are indexed by process numbers, e.g. Search$_k$. This $k \in$ *Processes* is the process that executes the fragment, and can be used in the body.
Because in the fragment Search the value of the array par at x has to be available, Search can only be executed by the process which is the owner of x. We define the fragment Search$_k(x,y)$, which is the execution of Search(x,y) by process k, i.e.

```
Search_k(x,y):  if par[x] ≠ x ∧ x ≠ y then
                    x := par[x];
                    if x < y then x,y := y,x fi;
                    Search_owner(x)(x,y);
                fi.
```

In Extend(x,y), after Search(x,y), $\mathtt{par}[x]$ is set to y. This can only be done by the process k which is the owner of x. This is the process that executes Search$_k(x,y)$. Therefore we extend the fragment Extend(x,y) to the parallel fragment

```
Extend_k(x,y):  if par[x] ≠ x ∧ x ≠ y then
                    x := par[x];
                    if x < y then x,y := y,x fi;
                    Extend_owner(x)(x,y);
                else if x ≠ y then
                    par[x] = y;
                fi,
```

which maintains all invariants and process k only inspects and updates $\mathtt{par}[x]$, when $owner(x) = k$.

Implementation with messages

We introduce the message type *edge*(x,y), as the command to link the trees of x and y. This means the receiving process k has to execute Extend$_k(x,y)$, i.e. at the arrival of an *edge* message, process k executes

Extend$_k(x,y)$: **if** par$[x] \neq x \wedge x \neq y$ **then**
  $x :=$ par$[x]$;
  if $x < y$ **then** $x,y := y,x$ **fi**;
  send *edge*(x,y) **to** *owner*(x);
else if $x \neq y$ **then**
  par$[x] := y$;
fi.

All processes repeatedly receive *edge* messages and execute Extend for each edge.

Tarpar*main*$_k$: **while** TRUE **do**
  receive *edge*(x,y);
  Extend$_k(x,y)$;
od.

All processes k initialize the parallel processing with the following fragment

Tarpar*init*$_k$: $F := \bigcup(k \in$ *Processes* $:: E_p(k))$;
for all $(x,y) \in$ *OutEdges*(k) **do**
  send *edge*(x,y) **to** k;
od,

where all processes k send all edges in *OutEdges*(k) to itself. The complete parallel solution for Tarjan's disjoint set algorithm is

Tarpar: $||_k$ Tarpar$_k$,

which is the parallel composition of all processes k executing Tarpar$_k$. Here, Tarpar$_k$ is defined as follows

Tarpar$_k$: Tarseq(*InEdges*(k));
Tarpar*init*$_k$(*OutEdges*(k));
Tarpar*main*$_k$.

Termination

The fragment Tarpar*main*$_k$ never terminates, because of the **while** TRUE **do** statement. Indeed, a process never knows when to stop, for new *edge* messages might still arrive. We present the following solution, where each process administrates how many edges have been added to F. All processes may terminate when $F =$ *Edges*.
F is a program variable and is distributed over the processes. Therefore F cannot be used to detect termination. In order to show maintenance of the invariants we show where F changes in the following fragments.
We present the following solution, where the *edge* message gets an extra argument, the *origin* of the edge, i.e. the process k that sends this edge in Tarpar*init*$_k$. An *edge* message is denoted by *edge*$(x,y,$ *origin*$)$.

Each process k has a private variable *ctok* which is the number of edges (x,y) with $owner(x) = k$, that have not been linked yet. Termination can be concluded when all processes have $ctok = 0$. *ctok* is initialized by

Tarpar$init_k$: $F := \bigcup(k \in Processes :: E_p(k))$;
ctok $:= 0$;
for all $(x,y) \in OutEdges(k)$ **do**
send $edge(x,y,k)$ **to** k;
ctok $:=$ *ctok* $+ 1$;
od;
if *ctok* $= 0$ **then**
send *gcdown* **to** *adm*;
fi.

To notify the *origin* that two trees have been linked, a *down* message is sent to the origin of the edge. If a *down* message arrives at process k, it decrements its *ctok* by one.
One process $adm \in Processes$ is called the administrator. It counts the processes k that still have $ctok > 0$ in the variable *gc*. *gc* is initialized to the number of processes. Each process sends a *GcDown* message to *adm* when its *ctok* value is zero. At the arrival of a *GcDown* message *adm* decrements *gc* by one. When *gc* becomes zero, all processes are notified that they may terminate by a *stop* message.
When a *stop* message arrives at a process, it terminates by setting a boolean variable *continue* to *false*, i.e. for each process

$$continue \equiv F \neq Edges.$$

In Extend$_k(x,y,origin)$ a *down* message is sent to the *origin* when the trees of x and y are connected. Note that even when the trees were already connected ($x = y$) a *down* message is sent.

Extend$_k(x,y,origin)$:
if $\mathtt{par}[x] \neq x \wedge x \neq y$ **then**
$x := \mathtt{par}[x]$;
if $x < y$ **then** $x,y := y,x$ **fi**;
send $edge(x,y)$ **to** $owner(x)$;
else
if $x \neq y$ **then**
$\mathtt{par}[x] := y$;
fi;
send *down* **to** *origin*;
fi.

The complete Tarpar$main_k$ is given below.

Tarpar$main_k$ **while** *continue* **do**
in $edge(x,y,origin) \rightarrow$
Extend$_k(x,y,origin)$;
[] *down* $\rightarrow$
ctok $:=$ *ctok* $- 1$;

```
            F := F ∪ {(x,y)};
            OutEdges(k) := OutEdges(k) \ {(x,y)};
            if ctok = 0 then send gcdown to adm fi
        [] gcdown →
            gc := gc − 1;
            if gc = 0 then
                for all p ∈ Processes do send stop to p od;
            fi;
        [] stop →
            continue := false;
        ni
    od .
```

3.4 Parallel Harvest

Recall from section 2.1 that a final labelling is requested in array `lab`. The array `lab` is distributed just like the array `par`, i.e. process k can only modify or update `lab`$[x]$ if $owner(x) = k$.
We define the set $OutPar(k)$ as the set of vertices x that belong to k whose parent `par`$[x]$ does not belong to k, i.e.

$$OutPar(k) = \{x \in D \mid owner(x) = k \wedge owner(\texttt{par}[x]) \neq k\}.$$

Recall from section 2.4 that the `lab` value of the root of each `par` tree is propagated over the other vertices in the tree. The `lab` value of each vertex in $OutPar(k)$ has to be known before we can apply the sequential algorithm, since the arrays `par` and `lab` can only be inspected locally.
The idea is to let the owner of `par`$[x]$ find the root of each x in $OutPar(k)$. When the root is found, the root is sent back to the owner of x. It then can set `lab`$[x]$ to the root of x.
We introduce a message $query(p,n)$ which is the request for the root of p in order to set the `lab` value of vertex n. We introduce a message $answer(r,n)$ which is the answer to a request sent by the owner of n. When a process receives a message $answer(r,n)$, it sets `lab`$[n]$ to r.
The harvest fragment is initialized as follows

```
Harvparinit_k: for all x ∈ {x ∈ D | owner(x) = k} do lab[x] := ⊥ od;
               ctok := 0;
               for all x ∈ OutPar(k) do
                   send query(par[x],x) to owner(par[x]);
                   ctok := ctok + 1;
               od;
               if ctok = 0 then
                   send gcdown to adm;
               fi.
```

In the main fragment of the parallel harvest algorithm each process k receives *query* messages. If k can answer it directly, it sends the answer to the owner of n, otherwise

a new *query* is sent to the owner of the next ancestor. We have solved the termination problem the same way as it is solved in the parallel solution Tarpar in section 3.2.

Harvpar*main*$_k$: **while** *continue* **do**
in *query*(p,n) $\rightarrow$
if par$[p] = p$ **then**
send *answer*(p,n) **to** *owner*(n);
else
send *query*(par$[p],n$) **to** *owner*(par$[p]$);
fi;
[] *answer*(r,n) $\rightarrow$
lab$[n] := r$
ctok $:=$ *ctok* $-$ 1;
if *ctok* $=$ 0 **then send** *gcdown* **to** *adm* **fi**;
[] *gcdown* $\rightarrow$
gc $:=$ *gc* $-$ 1;
if *gc* $=$ 0 **then**
for all $p \in$ *Processes* **do send** *stop* **to** p **od**;
fi;
[] *stop* $\rightarrow$
continue $:=$ *false*;
ni
od .

After the lab$[x]$ values have been set for all $(x,y) \in$ *OutPar*(k), the following modified version of Harvseq sets all other lab values

Harvpar*local*$_k$: **for all** $x \in \{x \in D \mid$ *owner*$(x) = k\}$ **do in increasing order**
if lab$[n] = \perp$ **then**
if par$[n] = n$ **then** lab$[n] :=$ lab$[n]$
else lab$[n] :=$ lab[par$[n]$] **fi**
fi
od .

The complete parallel harvest algorithm is

Harvpar$_k$: Harvpar*init*$_k$;
Harvpar*main*$_k$;
Harvpar*local*$_k$.

Chapter 4

Implementation

4.1 Introduction

In order to test the practical efficiency of the distributed version of Tarjan's disjoint set algorithm, we implemented it on a distributed system. We have used the C programming language, and the LAM MPI implementation to enable communication between the processes. In this chapter we briefly introduce the MPI interface, and show how we have transformed the pseudo code fragments from chapter 3 in C code.

4.2 Images

In our C code, the image f is coded as a one-dimensional array `im` of size `npixels` and of type integer, i.e.

```
int *im = malloc(sizeof(int)*npixels),
```

where `npixels` is the total number of pixels in the image. In a 2D image of size $m \times n$, `npixels` $= mn$. The pixels of f are stored in scan-line order.
Figure 4.1 shows an example of how the pixels are stored in memory. In this example, `npixels` $= 3 \times 4 = 12$.

4.3 Communication with the MPI interface

We have used the the Message Passing Interface (MPI), which is a portable message-passing standard that facilitates the development of parallel applications and libraries. The scope of each MPI operation is defined by the *communicator* data object. By default this is the set of all processes, `MPI_COMM_WORLD`.
In this section we show the operations we have used to implement the algorithms in chapter 3.
More specific information about the MPI interface can be found in [MPIStd]. In MPI, the set of processes *Processes* is the set $\{0..N-1\}$, where N is the number of processes. Recall from section 1.3 that we assume that each process runs on a processor, and on one processor only one process is executed.

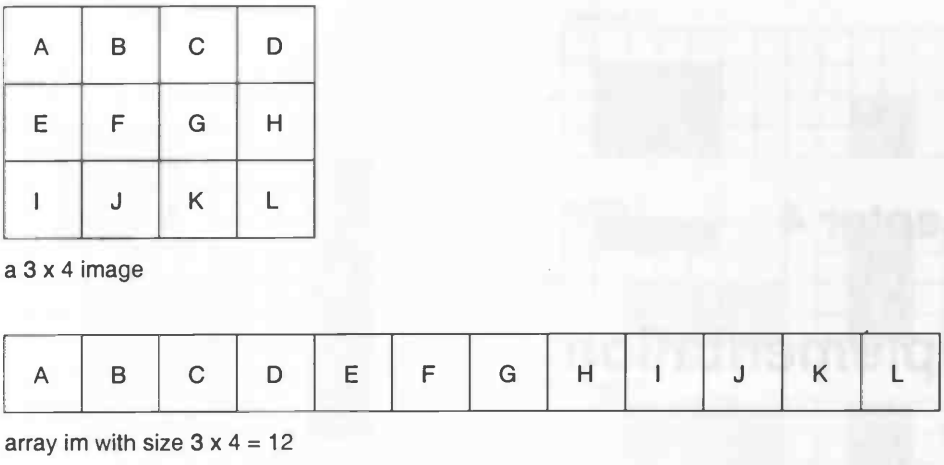

Figure 4.1: The order in which the pixels of a 3 x 4 image are stored in array `im`.

Distribution of memory

To distribute a buffer over processes, in our case array `im`, a call to

```
MPI_Scatter(sendbuf,sendcount,sendtype,
            recvbuf,recvcount,recvtype,
            root,communicator)
```

is made. In this operation, the process `root` sends an equal part of `sendbuf` to each process in `communicator`; `sendcount` is the number of `sendtype` elements in `sendbuf`. The parts of the buffer are stored in `recvbuf`, which assumes `recvcount` of `recvtype` elements to arrive.

Gathering of memory

The gathering of a buffer is the dual of the scatter operation. The syntax is

```
MPI_Gather(sendbuf,sendcount,sendtype,
           recvbuf,recvcount,recvtype,
           root,communicator).
```

In this operation each process sends its `sendbuf` to process `root`. There are `sendcount` elements of type `sendtype` in buffer `sendbuf`. Process `root` receives a part of its buffer from all processes in `communicator` (including itself), and stores all parts in `recvbuf`.

Sending a message

Recall from section 1.3 that the send operation we use is non-blocking. The operation `MPI_Send` for sending messages is a blocking MPI operation. We use a variation for immediately sending messages, `MPI_Isend`, which is non-blocking.
The procedure

```
MPI_Isend(outmessage,size,type,dest,tag,communicator,request)
```

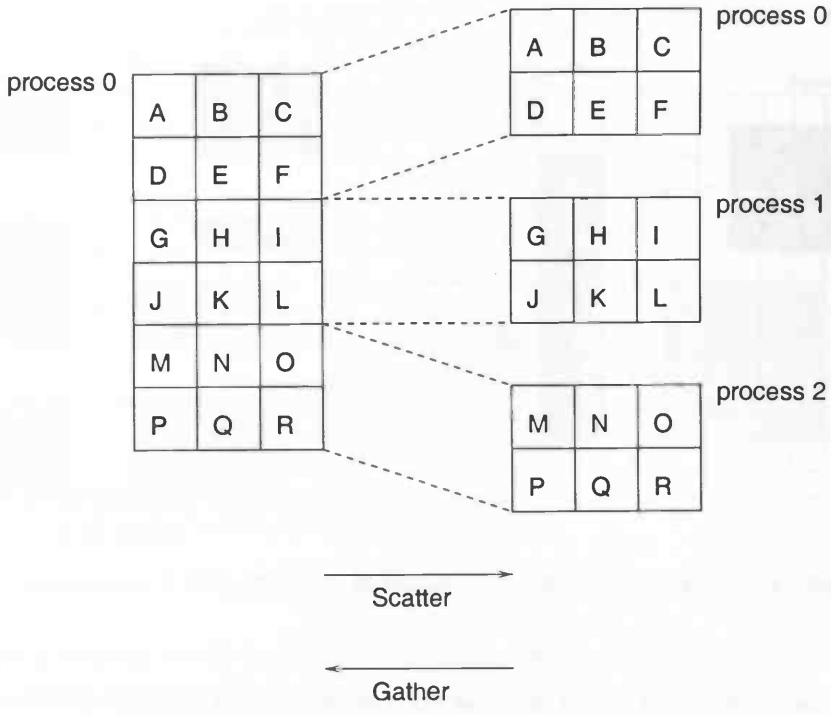

Figure 4.2: The scatter operation and its dual, the gather operation.

is used to send data to process `dest`; `outmessage` is the address of the databuffer to be sent, and contains `size` elements of type `type`. The `tag` is an integer value, which is sent with the message. The receiver can use the tag to select which message it wants to receive. The value `communicator` is a group of processes. `request` is the address of a `MPI_REQUEST` object, containing information about the status of the message, after `MPI_Isend` is called.
Note that the memory that has to be sent can be reused only after the message has actually been sent to the receiver. To check whether the memory can be reused, the `request` has to be checked with the `MPI_Test` procedure, which returns immediately. The `MPI_Wait` procedure can be used to perform a blocking wait for receipt of a message.

Receiving a message

We only use a blocking receive. This means the receiving process waits for a message to arrive when the procedure `MPI_Recv` is called.
The syntax of `MPI_Recv` is

```
MPI_Recv(inmessage,size,type,source,tag,communicator,&status),
```

where `inmessage` is the address of the buffer where the message should be stored. The integer value `size` is the maximum number of data items of type `type` that can arrive. `source` is the source of the message, and can be set to `MPI_ANY_SOURCE` if a message from multiple processes can arrive. `tag` is the tag sent with the message. If messages with different tags can arrive, the value `tag` should be set to `MPI_ANY_TAG`. `communicator` is the group of processes. In `status` some information about the

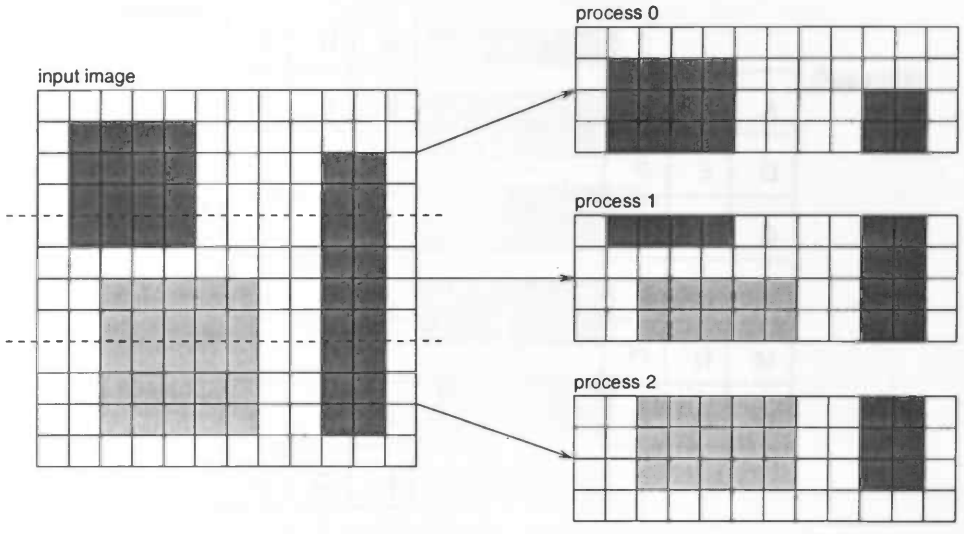

Figure 4.3: An example of the distribution of an image over 3 processes.

incoming message can be found, e.g. the number of received data-items, the source and the tag of the message.

4.4 Distribution of the image

Recall from chapter 3 that we defined a function *owner*. At the distribution of the image, each pixel x is sent to the process k, with $owner(x) = k$. The `MPI_Scatter` procedure is used for the distribution of the array `im`. The image is divided in N consecutive parts, where N is the number of processes. Each process k gets the k^{th} part of the scanlines. From this distribution we conclude that for each x and y in D

T1 $\quad x \leq y \quad \Rightarrow \quad owner(x) \leq owner(y).$

From *T1* and the invariant $\mathtt{par}[x] \leq x$ we conclude that for all $k \in Processes$

T2 $\quad \forall(x \in D : owner(x) = k : owner(\mathtt{par}[x]) \leq k).$

In figure 4.3 an example of this distribution is shown. Note that the concatenation of all parts of array `im` is equal to `im` itself.

4.5 Optimization

To enhance performance of the algorithms, we introduce a number of optimizations. In all cases this decreases the amount of communication. In this section we will show why and how we used it, and why it preserves the correctness of the parallel Tarjan's disjoint set algorithm. We introduce the following ideas.

Local FindRoot The search for a root on process k, within the part of `im` belonging to k, can be done without communication.

Suspended queries In the harvest fragment some queries are not answered directly.

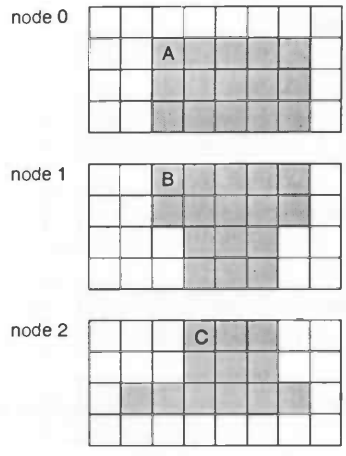

Figure 4.4: An example of a distributed 2D image. A, B and C are the roots of local connected components.

Message grouping Some types of messages are grouped together in one MPI message that is sent to another node.

Local FindRoot

In both Tar*parmain* and Harv*parmain* there are many messages sent from a process to itself. In both Tar*parmain* and Harv*parmain* the local root has to be found in order to continue. With the use of the function *LocalFindRoot* the processing of messages of type *edge* and *query* is rewritten to a more efficient implementation.
This optimization decreases the amount of communication dramatically, due to the distribution of `im` and the invariant $\mathtt{par}[x] \leq x$. The pseudo code fragment for FindLocalRoot is

FindLocalRoot$_k(x)$:

$r := x$;
while $\mathtt{par}[r] \neq r \wedge \mathit{owner}(\mathtt{par}[r]) = k$ **do**
  $r := \mathtt{par}[r]$;
od;
return r.

In paragraph **Optimized algorithms** we show how we used this optimization.

Suspended queries

In the harvest algorithm there are a number of pixels x whose owner is not equal to the owner of $\mathtt{par}[x]$. The roots of these pixels are needed in order to label the pixels correctly. Therefore a *query* message is sent to the owner of $\mathtt{par}[x]$.
When, in the original algorithm, the root of x does also not exist on the receiving process, this process forwards the *query* to the owner of the `par` of the local root. In our optimized algorithm it *suspends* this query, which means it stores the *query* until the real root of the local root is known. Note that also for this local root a *query* has already been sent. When the real root is known, an *answer* message with the real root is sent to the owner of x.

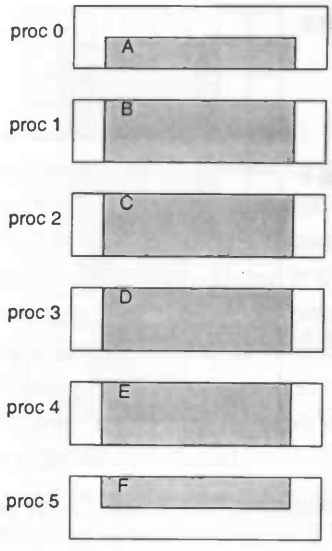

Figure 4.5: An image that contains a component that exists on all six processes.

From theorem *T2* we conclude this optimization works. The pseudo code can be found in the **Optimized Algorithms**.

Example: In figure 4.4 an example of a 2D image is shown, distributed over three processes. In Harvpar*main* the root of pixel C is asked for by process 2. Process 2 sends a *query* message for pixel C to process 1. Because process 1 is not the owner of $\mathtt{par}[B]$, it can not answer the query from process 2 directly.

Instead of forwarding the request to process 0, as was done in section 3.4, process 1 now suspends the query while the root of B is unknown, by storing it in the *WaitingList* of B. If process 1 receives the answer for B from process 0, process 1 checks whether there are suspended queries for B by checking the *WaitingList* of B. For each pixel in this *WaitingList* process 1 sends the answer A to the owner. In the example the *WaitingList* of B is the set $\{C\}$. Therefore process 1 sends the answer A to process 2.

Example: In figure 4.5, an image is shown, distributed over six processes. The shaded component exists on all processes. The characters are the roots of the local components. In the original situation the following messages have to be sent in order to set all roots.

1 process 5 sends *query* F to process 4
2 process 4 sends *query* E to process 3
3 process 3 sends *query* D to process 2
4 process 2 sends *query* C to process 1
5 process 1 sends *query* B to process 0
6 process 0 sends *answer* A to process 5
7 process 4 sends *query* E to process 3
8 process 3 sends *query* D to process 2
9 process 2 sends *query* C to process 1
10 process 1 sends *query* B to process 0
11 process 0 sends *answer* A to process 4
12 process 3 sends *query* D to process 2
13 process 2 sends *query* C to process 1

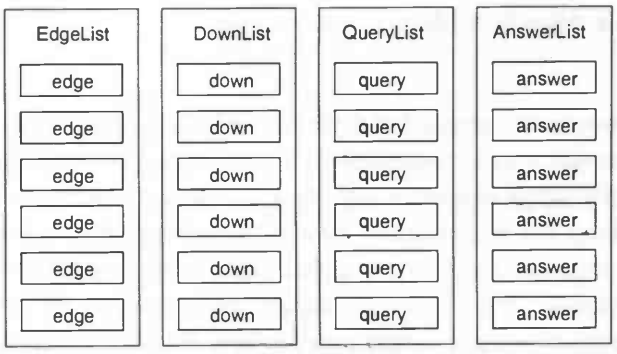

Figure 4.6: Four types of grouped messages.

14 process 1 sends *query* B to process 0
15 process 0 sends *answer* A to process 3
16 process 2 sends *query* C to process 1
17 process 1 sends *query* B to process 0
18 process 0 sends *answer* A to process 2
19 process 1 sends *query* B to process 0
20 process 0 sends *answer* A to process 1

With the optimization presented the following messages have to be sent.

1 process 5 sends *query* F to process 4
2 process 4 sends *query* E to process 3
3 process 3 sends *query* D to process 2
4 process 2 sends *query* C to process 1
5 process 1 sends *query* B to process 0
6 process 0 sends *answer* A to process 1
7 process 1 sends *answer* A to process 2
8 process 2 sends *answer* A to process 3
9 process 3 sends *answer* A to process 4
10 process 4 sends *answer* A to process 5

This optimization clearly decreases the number of messsages.

Message grouping

Some types of messages are grouped together in one MPI message. In figure 4.6 an example is shown for each type of message that can be grouped.
Each process keeps a messagelist of each type of message for each process $p \in$ *Processes*. If a single message of type *amsg* should be sent to process p, it adds *amsg* to the *AmsgList* of process p, which we denote as *AmsgList*(p). If the amount of single messages in a messagelist reaches a predefined maximum `MAXSIZE` the messagelist is sent to process p. We introduce the new keyword **add**, which adds a single message to a messagelist, i.e.

add *msg* **to** *MsgList*(p):
MsgList(p) := *MsgList*$(p) \cup$ *msg*;
if #*MsgList*(p) = `MAXSIZE` **then**

```
            send MsgList(p) to p;
        fi.
```

This evidently decreases the amount of communication. In order to avoid deadlock a process sends all current messagelists before it waits for a new message to arrive.
The correctness of the single message algorithm is shown in chapter 3, which does not assume anything about the time between sending a message and actually receiving it. Deadlock is not introduced by grouping the messages in the way we described.
This is shown as follows: when deadlock occurs, all processes are waiting for message reception and therefore have empty send buffers. Because all grouped messages are sent before a process waits for a new message to arrive, this is equal to the original algorithm. We have shown that in the original algorithm deadlock can not occur. Therefore it can not occur in the optimized version.
The time between the sending and receiving of a single message may be larger when they are grouped than when all single messages are sent directly. Still, in practice, this optimization has a positive influence on the performance.

Optimized algorithms

Below the optimized pseudo code fragments of Tarpar and Harvpar are given.

Tarpar$init_k$:

```
ctok := 0;
for all (x,y) ∈ OutEdges(k) do
    ctok := ctok + 1;
    add edge(x,y,k) to EdgeList(k);
od;
if ctok = 0 then
    send gcdown to adm
else
    send EdgeList(k) to k
fi .
```

Tarpar$main_k$:

```
while continue do
    in EdgeList →
            forall edge(x,y,origin) in EdgeList do
                x := FindLocalRoot(x);
                y := FindLocalRoot(y);
                if x < y then x,y := y,x fi;
                if owner(x) ≠ k then
                    add edge(x,y,origin) to EdgeList(owner(x));
                else
                    par[x] := y;
                    add down to DownList(origin) ;
                fi ;
            od;
            for all p ∈ Processes do
                if not empty(EdgeList(p)) then
                    send EdgeList(p) to p;
                fi;
                if not empty(DownList(p)) then
```

```
                    send DownList(p) to p;
                fi;
            od;
        [] DownList  →
            ctok := ctok − #DownList ;
            if ctok = 0 then send gcdown to adm fi;
        [] gcdown  →
            gc := gc − 1;
            if gc = 0 then
                for all p ∈ Processes do send stop to p od;
            fi;
        [] stop  →
            continue := false;
        ni
    od .
```

Harv$par init_k$:

```
for all x ∈ {x ∈ D | owner(x) = k} do root[x] := ⊥ od;
ctok := 0;
continue := TRUE;
for all x ∈ OutPar(k) do
    add query(par[x], x) to QueryList(owner(par[x]));
    ctok := ctok + 1;
od;
if ctok = 0 then
    send gcdown to adm;
else
    forall p ∈ Processes do
        if not empty(QueryList(p)) then
            send QueryList(p) to p;
        fi;
    od;
fi.
```

Harv$par main_k$:

```
while continue do
    in AnswerList  →
            forall answer(r, n) in AnswerList do
                root[n] := r
                for all x in WaitingList(r) do
                    add answer(r, x) to AnswerList(owner(x));
                od;
                ctok := ctok − 1;
                if ctok = 0 then send gcdown to adm fi;
            od;
            for all p ∈ Processes do
                if not empty(AnswerList(p)) then
                    send AnswerList(p) to p;
                fi;
            od;
    [] QueryList  →
```

```
                forall query(r,n) in QueryList do
                    r := FindLocalRoot(r);
                    if owner(r) = k then
                        add answer(r,n) to AnswerList(owner(n));
                    else if root[r] ≠ ⊥ then
                        add answer(root[r],n) to AnswerList(owner(n));
                    else
                        add n to WaitingList(r);
                    fi;
                od;
                for all p ∈ Processes do
                    if not empty(QueryList(p)) then
                        send QueryList(p) to p;
                    fi;
                    if not empty(AnswerList(p)) then
                        send AnswerList(p) to p;
                    fi;
                od;
            [] gcdown →
                gc := gc − 1;
                if gc = 0 then
                    for all p ∈ Processes do send stop to p od;
                fi;
            [] stop →
                continue := false;
            ni
        od .
```

4.6 Translation to C

In this section we show how we translated some pseudo code fragments to actual C code fragments, using the MPI interface. The type of a message is defined by the tag of a message. The tag values are predefined integers, e.g. the message-type *amsg* is the integer `AMSG` in a C fragment.

Sending a single message

The pseudo code call to

send *amsg*(a,b,c) **to** y,

which means sending a message of type *amsg* with integer arguments a, b, and c to process y, is transformed to the C code fragment

```
int *outmessage = malloc(3*sizeof(int));
MPI_REQUEST request;

outmessage[0] = a;
outmessage[1] = b;
outmessage[2] = c;
```

```
MPI_Isend(outmessage,3,MPI_INT,y,
          AMSG,MPI_COMM_WORLD,&request).
```

Receiving a single message

The corresponding pseudo code call to

receive *amsg*(a,b,c) **from** x

is translated to

```
int *inmessage = malloc(3*sizeof(int));
MPI_Status status;

MPI_Recv(inmessage,3,MPI_INT,x,
         AMSG,MPI_COMM_WORLD,&status);
a = inmessage[0];
b = inmessage[1];
c = inmessage[2].
```

The pseudo code fragment

in *xmsg* $\rightarrow$
    fragA;
[] *ymsg*(a) $\rightarrow$
    fragB(a)
[] *zmsg*(b,c) $\rightarrow$
    fragC(b,c);
ni,

which is used when messages of multiple message types can arrive is translated to

```
int *inmessage = malloc(2*sizeof(int));
int a,b,c;
MPI_Status status;

MPI_Recv(inmessage,2,MPI_INT,MPI_ANY_SOURCE,
         MPI_ANY_TAG,MPI_COMM_WORLD,&status);
switch (status.MPI_TAG) {
  case XMSG:
    fragA();
    break;
  case YMSG:
    a = inmessage[0];
    fragB(a);
    break;
  case ZMSG:
    b = inmessage[0];
    c = inmessage[1];
    fragC(b,c);
    break;
}
```

Sending of grouped messages

We define a new C structure for grouped messages

```
struct MessageList {
  int *data;
  int nmsg;
  MPI_Request request;
}.
```

One messagelist is initialized as follows.

```
#define MSIZE 2
struct MessageList AMList;
AMList.data = malloc(MAXSIZE*MSIZE*sizeof(int));
AMList.nmsg = 0;
```

The **add** procedure

add *msg*(x,y) **to** *MsgList* :
  MsgList := *MsgList* $\cup$ *msg*(x,y);
  if #*MsgList* = MAXSIZE **then**
    send *MsgList* **to** *procR*;
  fi.

is translated to

```
AMList.data[MSIZE*AMList.nmsg]=x;
AMList.data[MSIZE*AMList.nmsg+1]=y;
AMList.nmsg++;
if (AMList.top==MAXSIZE) {
  MPI_Isend(AMList.data,MSIZE*MAXSIZE,MPI_INT,procR,
            AMLIST,MPI_COMM_WORLD,&(AMList.request)));
}.
```

If the messagelist `AMList` has to be sent to process y before the maximum size is obtained, we use the following fragment.

```
MPI_Isend(AMList.data,MSIZE*AMList.nmsg,MPI_INT,procR,
          AMLIST,MPI_COMM_WORLD,&(AMList.request)).
```

Receiving of grouped messages

The corresponding pseudo code fragment for receiving a grouped message

receive *AMList* **from** *procS*;
for all *Atype*(x,y) **in** *AMList* **do**
  process(x,y);
od

is translated to the C code fragment

```
int *inmessage = malloc(MSIZE*MAXSIZE*sizeof(int));
int i,nelements,x,y;
```

```
MPI_Status status;

MPI_Recv(inmessage,MSIZE*MAXSIZE,MPI_INT,procS,
         AMLIST,MPI_COMM_WORLD,&status);
nelements = status.MPI_SIZE / MSIZE;
for (i=0;i<nelements;i++) {
  x = inmessage[MSIZE*i];
  y = inmessage[MSIZE*i+1];
  process(x,y);
}.
```

Chapter 5

Performance

This chapter discusses the performance of the algorithms presented in the previous chapters. We check the practical efficiency of the parallel implementation of Tarjan's disjoint set algorithm. Recall from section 1.3 that we assume that each process runs on a processor, and on each processor only one process is executed.
The performances of application to 2D images is shown in this chapter. The results are a good indication for the performance of images of arbitrary dimensions.
It is interesting to measure the time a process needs to analyse a 256×256 2D image and compare it to the time it takes to analyse a 512×512 image. We varied five parameters, which are

- The number of processes.
- The contents of the input image.
- The size of the input image.
- The implementation of Tarjan's disjoint set algorithm.
- The maximum size of grouped messages.

In this chapter we show why we expect an increase or decrease of the performance, when one of the parameters is changed. We also show how the performance depends on the parameters in practice.
We measure the wall clock time in milliseconds t_1. It is interesting to see what happens if we change one of the five parameters given above. The new time measured, t_2, is compared to t_1. An important value is the speedup, the ratio between t_1 and t_2, i.e.

$$speedup = \frac{t_1}{t_2}$$

5.1 Contents of the image

The contents of an image has influence on the performance of Tarjan's disjoint set algorithm. When there are many connected components in an image that reside at more than one process, a lot of communication is needed. The number of connected components that are present at more than one process also depends on the distribution of the image. In our distribution, as described in section 4.4, the rows are distributed over the processes.

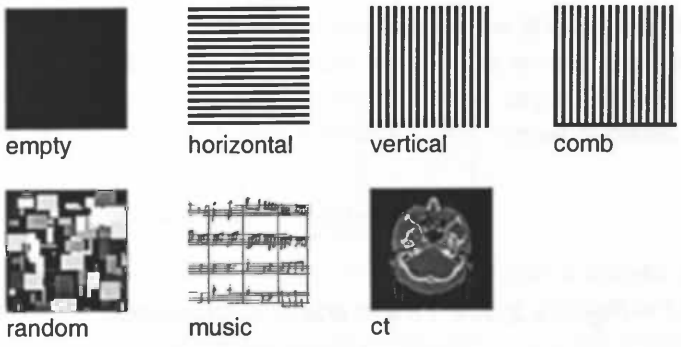

Figure 5.1: Images used for testing.

We used seven different images for the testing of the performance. In figure 5.1 the images are shown. The images `vertical`, `horizontal` and `comb` are shown at size 32×32 for visibility reasons. The other images have the original size 256×256. Below, we give a description of all images.

`empty` All pixels are black. This image therefore consists of just one connected component. The amount of communication is small.

`horizontal` In this image there are only horizontal lines. All odd lines are black and all even lines are white. In a $n \times m$ image there are n connected components. There are no connected components present at more than one process. Therefore, the amount of communication is very small.

`vertical` This image is equal to `horizontal`, turned 90 degrees. All odd columns are black and all even columns are white. In a $n \times m$ image there are m connected components. All connected components are present on all processes. Therefore, a lot of communication is needed.

`comb` This image is equal to `vertical`, except for the last line, which is black. This results in a large connected component, consisting of all vertical black lines. All white lines are separate components. In a $n \times m$ image there are $(m+1)/2+1$ connected components. All connected components are present on all processes. Therefore, a lot of communication is needed.

`random` This is an image of 50 randomly placed squares of different sizes and grey values. The background is black. This resembles more natural images than the previous ones. Some components have to be linked on more than one process, therefore an average amount of communication is needed.

`music` This is a two colour scan of a paper with handwritten music. This image consists of a few large and many small components. Some components have to be linked on more than one process, therefore an average amount of communication is needed.

`CT` This is a realistic photo which is one slice of a CT scan of a human head. There are many connected components of different sizes. Some components have to be linked on more than one process, therefore an average amount of communication is needed.

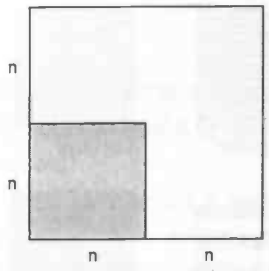

Figure 5.2: A $n \times n$ and a $2n \times 2n$ 2D image.

5.2 Expected performance

Number of processes

We denote the number of processes with *nprocs*. An increase of the number of processes should result in a performance increase, i.e. $speedup > 1$. In the ideal case the speedup is equal to the ratio between the number of processes. E.g. if t_1 is the time a certain job takes when only one process is used, the time to do the same job on two processes t_2 is ideally $t_1/2$. If there is no communication nor computation overhead we have

$$speedup = \frac{t_1}{t_2} = \frac{nprocs_2}{nprocs_1},$$

where $nprocs_1$ and $nprocs_2$ are the numbers of processes used to measure time t_1 and t_2.
Because there is always some communication needed to do a certain job, the *speedup* is usually smaller than $nprocs_2/nprocs_1$.

Contents of the input image

The performance of Tarjan's algorithm strongly depends on the contents of the image. More communication results in lower performance.
E.g., processing of the image `vertical` results in much more communication overhead than the image `horizontal` and therefore a lower performance.
How the contents has effect on the performance is discussed in section 5.6.

Size of the input image

We only consider 2D square images of size $n \times n$. The parameter n of an image is called the size of the image.
The area, i.e. the number of pixels, of the image is n^2. Of course, the speedup is less than one when larger images are analysed. More exactly, the expected speedup is equal to the factor between the areas of the images. Therefore the theoretical *speedup* is

$$speedup = \frac{t_1}{t_2} = (\frac{n_1}{n_2})^2,$$

where n_1 and n_2 are sizes of the images used to measure time t_1 and t_2.

An example is given in figure 5.2, where the area of the shaded box, with size n, is n^2 and the area of the white box, of size $2n$, is $4n^2$. The area is four times larger and the time needed to analyse will therefore be four times larger. Hence, the expected *speedup* is $1/4$, for an $O(N)$ algorithm, where N is the number of pixels.

Implementation of Tarjan's disjoint set algorithm

There are many variations of the implementations of Tarjan's disjoint set algorithm. In [WHH01], a sequential solution is given, which we have parallellized for distributed memory computers. Because the pixels are processed in positive scan-line order, we call this algorithm the *forward* variation, and we call the algorithm given in section 2 the *backward* variation.
The forward and the backward version of Tarjan's algorithm have both advantages and disadvantages. These are sketched below. The actual effect of these factors is discussed in the results of the experiment in section 5.6.
The forward algorithm has an advantage to the backward algorithm. In the parallel harvest a *Findroot* is done for all vertices in *OutPar*. Because in the forward algorithm almost full path-compression is used, the roots can be found very quickly.
The backward algorithm has the following advantage to the forward algorithm: because in Tarjan's disjoint set algorithm only pixels in two following scan-lines are inspected and the scan-lines are placed next to each other in memory, the caches of the processors are used better. In the forward algorithm, the root of the `par` trees is inspected each time trees are linked. This root is often a few scan-lines away, and will probably not be in the cache anymore.

The maximum size of grouped messages

The value `MAXSIZE`, introduced in section 4.5, also has effect on the performance of the parallel execution. `MAXSIZE` is the maximum number of messages in a grouped message. A small `MAXSIZE` should result in faster delivery of messages. However, there is more traffic on the network. In order to create comparable tests, we define the `MAXSIZE` as the factor between the width of the input image, and the parameter *mpart*. This parameter *mpart* is the part of a scanline that can be grouped in one message. $mpart = \infty$ means that no messages are grouped together at all.
A large *messagepart* should result in many messages, and therefore result in a lower performance.

5.3 Architecture

In order to give a realistic view on the measured timings, we describe the system we have used to measure the performance.
We have used a cluster of four Compaq XP1000 workstations linked by a 100 Mbit, full duplex network. Each workstation is equipped with the 667 MHz 21264A Alpha processor and 256 MB memory. The RedHat Linux operating system is used together with LAM MPI.

5.4 Timing

To be able to time the implemented algorithms we made a small timing module, which enables us to determine the execution time of a program fragment. It also allows us to start multiple timers in one execution.
The interface is very simple. A user can declare a variable of type `Timer`. The time is initialized by the command `newtimer` which returns a new `Timer`. The timer can be started with the command `starttimer`, and stopped with the command `stoptimer`. The value of a timer can be read with the command `timervalue`. This returns the value of the timer in milliseconds. The next C fragment illustrates the use of this timer.

```
Timer t;
t = newtimer();
starttimer(t);
Tarjan(image, par);
stoptimer(t);
printf("%ld",timervalue(t));
```

In this C fragment the duration of the function `Tarjan` is measured and written to standard output.

5.5 Method

We have timed the segmentation several times, while changing the parameters discussed above:

for all *image* $\in$ {`empty`, `horiontal`, `vertical`, `comb`, `random`, `music`, `ct`} **do**
  for all *size* $\in \{128, 256, 512, 1024, 2048, 4096\}$ **do**
    for all *nprocs* $\in \{1, 2, 3, 4\}$ **do**
      for all *mpart* $\in \{1, 4, 16, 64, 256, \infty\}$ **do**
        for all *method* $\in$ {*forward*, *backward*} **do**
          time *segmentation*(*image*, *size*, *nprocs*, *mpart*, *method*);
        od;
      od;
    od;
  od;
od;

The results can be found in appendix A.

5.6 Results

In this section we show the graphs, generated from the test results in appendix A. The efficiency graphs can be found in figure 5.3 to 5.8. The value of the x axis is the *method* used to analyse the image. The meaning of each method can be found in table 5.6.
In the graphs the efficiency of one processor is shown, when four are used to solve the problem. Note that the lines between points in method 5 and 6 have no meaning. The efficiency is calculated as follows.

$$Efficiency = \frac{t_1}{nt_n} \times 100\%,$$

where t_n is the time spent using n processes.

method	processing method	mpart
0	forward	1
1	forward	4
2	forward	16
3	forward	64
4	forward	256
5	forward	∞
6	backward	1
7	backward	4
8	backward	16
9	backward	64
10	backward	256
11	backward	∞

Table 5.1: The meaning of each method in the efficiency graphs.

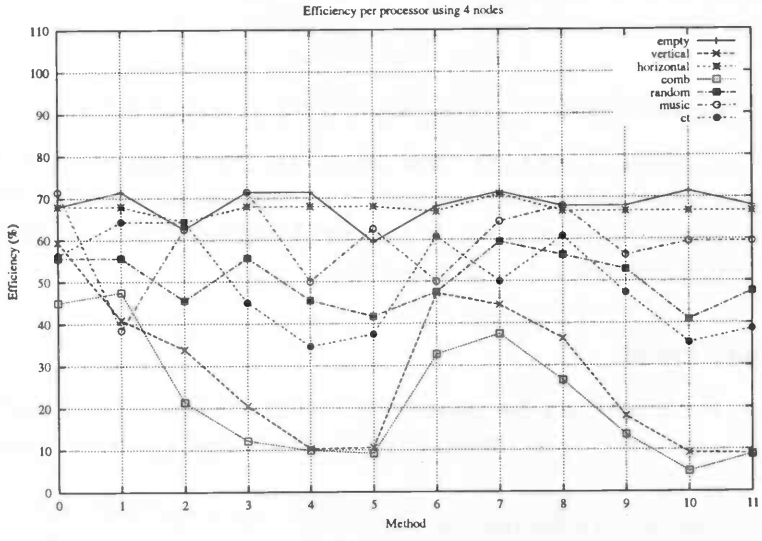

Figure 5.3: Efficiency graph of the test with 128×128 images.

5.7 Conclusions

In this section we show our conclusions from the test results and the efficiency graphs. In some cases, the efficiency of four processes is larger than 100%. Probably this is caused by a more efficient use of the cache of the processors when the image is split up. In the following paragraphs we compare the practical speedups to the speedups expected in section 5.2.

Number of processes

From the efficiency graphs we conclude that for large enough images the segmentation algorithm can almost perfectly be distributed over more than one process.

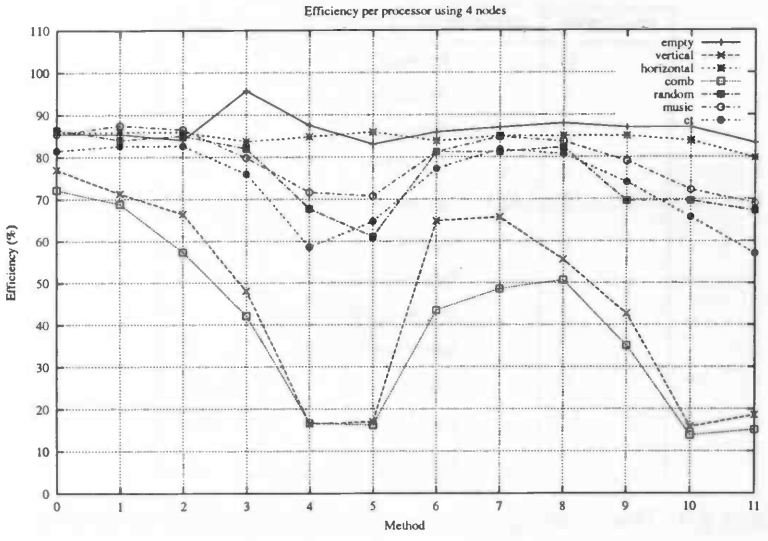

Figure 5.4: Efficiency graph of the test with 256×256 images.

Contents of the input image

As predicted, the images `vertical` and `comb` are the images that are the most difficult to process distributedly. From appendix A we also can conclude that the images where almost no linking is needed, e.g. `horizontal` and `vertical`, can be analysed more quickly than the others, e.g. `music` and CT.

Size of the input image

From appendix A we conclude the *speedup* is exactly the way we expected it to be. When the size is multiplied by two, the calculation takes about four times more time.

Implementation of Tarjan's disjoint set algorithm

The backward algorithm indeed is a little faster than the forward algorithm, however, when more than one process is used the drawback of the minimal path compression becomes visible. Especially in the case of images with many distributed connected components, e.g. in `vertical`, and large images, e.g. of size 4096, the speedup is lower.

Number of grouped messages

The effect of the grouping of messages becomes very clear from the efficiencygraphs. Of course, if the width of an image is larger than the *messagepart*, no extra efficiency can be expected. But even when a little more grouping is done, the efficiency increases. When the image is very large, e.g. 4096, the effect of grouping is minimal, due to the enormous amount of calculations needed to execute the sequential part. Though, also in this case, a *mpart* of 256 is better than no grouping at all, i.e. $mpart = \infty$.

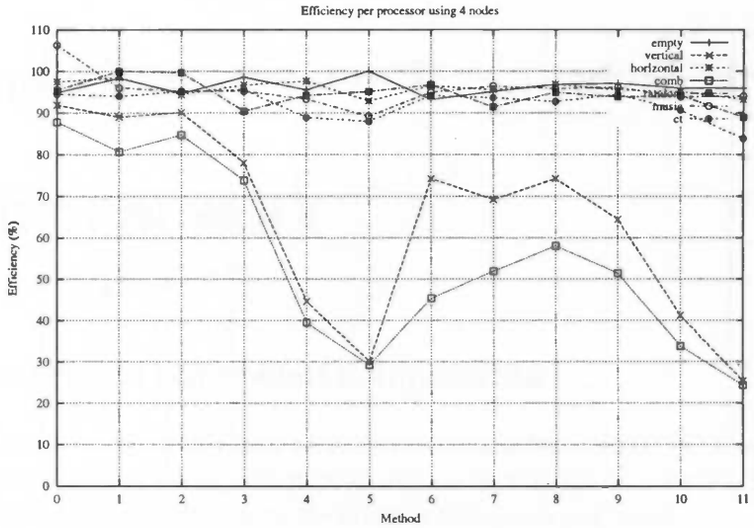

Figure 5.5: Efficiency graph of the test with 512×512 images.

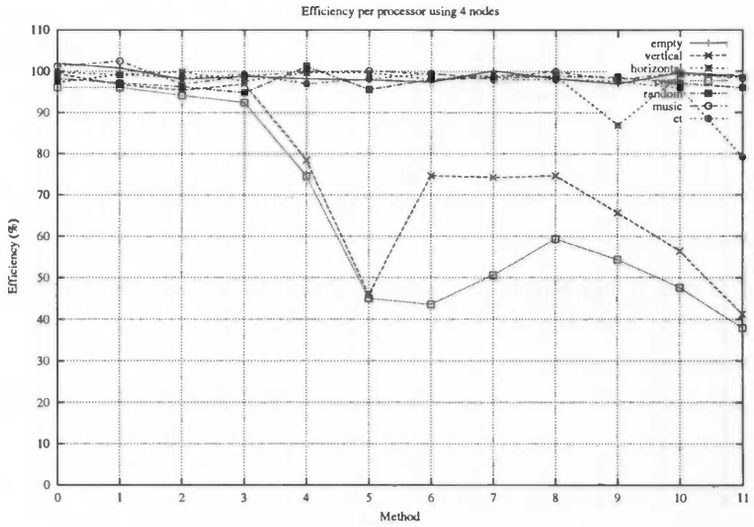

Figure 5.6: Efficiency graph of the test with 1024×1024 images.

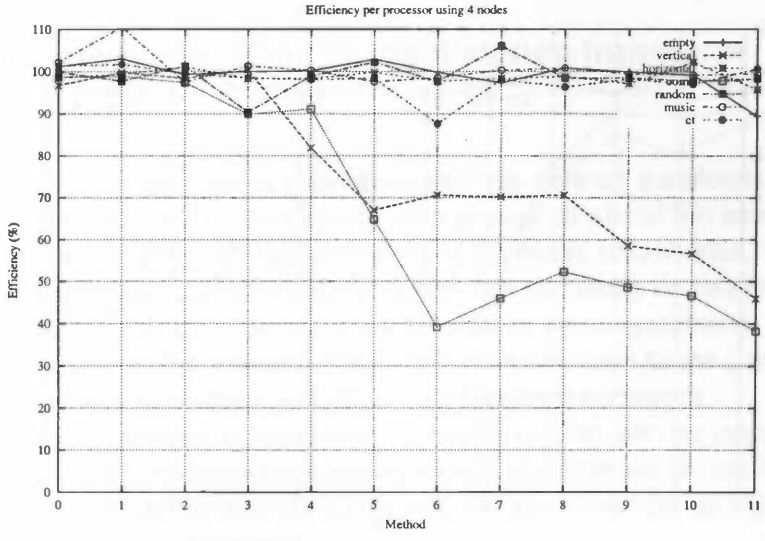

Figure 5.7: Efficiency graph of the test with 2048 × 2048 images.

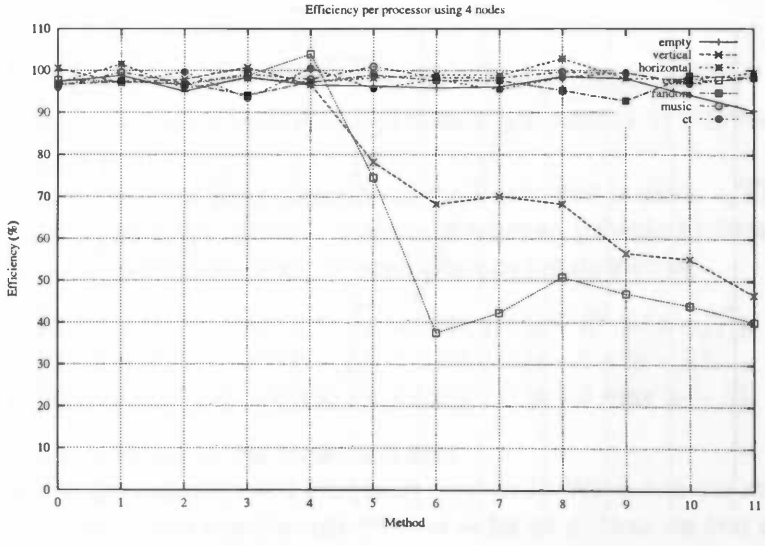

Figure 5.8: Efficiency graph of the test with 4096 × 4096 images.

Chapter 6

Additional work

6.1 The area of connected components

It is often useful to know the area of the connected components. In this section we show how we changed the algorithm, discussed in section 2.3, to calculate the area of each connected component, during the construction of the par trees.
In our solution, the area of each connected component is stored at the root of the par tree. The idea is to add the area of the set of x to the area of the set of y, when x is linked to y. Recall from section 2.3 that y is always a new root of a par tree in the sequential algorithm. Therefore the area of the root of x must be added to the area of y. Note that the original algorithm of constructing the par trees does not change at all. The area is stored in the array area. The area of the connected component of x can therefore be found in $\mathtt{area}[root(x)]$. Besides the invariants defined in section 2.3 we introduce the following new invariant

J_a: $\quad \forall(x \in D :: \mathtt{area}[x] = \#\{y \mid y \in D \wedge root(y) = x\})$.

This invariant is initialized by

Tar$_{\mathtt{area}init}$: **for all** $x \in D$ **do**
$\quad \mathtt{area}[x] = 1;$
$\quad \mathtt{par}[x] = x;$
od.

The only change to the original fragment Extend(x, y) is that $\mathtt{area}[root(x)]$ is added to $\mathtt{area}[y]$.

Extend(x, y): **while** $\mathtt{par}[x] \neq x$ **do**
$\quad p := \mathtt{par}[x]$;
$\quad \mathtt{par}[x] := y$;
$\quad x := p$;
od;
if $x \neq y$ **then**
$\quad \mathtt{par}[x] = y;$
$\quad \mathtt{area}[y] := \mathtt{area}[y] + \mathtt{area}[x];$
$\quad \mathtt{area}[x] := 0;$
fi.

The same transformation can be made for the parallel algorithm. When in *Extend*$(x, y, origin)$ the `par` of a vertex x is set to y, the area of x must be added to the area of the *root* of y.

6.2 Distributed calculation of the distance transform

Introduction

In [AM2000] a general algorithm is given for computing distance transforms in linear time. The algorithm consists of two phases. Both phases consist of two scans, a forward and a backward scan. The first phase scans the image column-wise, while the second phase scans the image row-wise. The algorithm can easily be parallelized because the computation for each row row is independent of the computation of the other rows. This is also true for the columns. The algorithm can be used for the computation of the exact Euclidean, Manhattan, and Chessboard distance transforms.
In [AM2000] a parallel solution for the distance transform is given, with the use of shared memory computers. In this section we show how this algorithm works, and we show how we ported it to distributed memory computers. We also show how we transformed this algorithm into a feature transform.

Problem description

The image-domain is a rectangular grid of size $m \times n$, i.e.

$$D = \{0,..,m-1\} \times \{0,..,n-1\},$$

and there exists a binary image

$$b :: D \rightarrow \{\mathtt{TRUE}, \mathtt{FALSE}\}.$$

We define the set B as

$$B = \{(x,y) \in D \mid b(x,y) = \mathtt{TRUE}\}.$$

The goal of the distance transform is to assign to each grid point $(x,y) \in D$ the distance to the nearest point in B.
The definition of the nearest point depends on the metric that is used. $\sqrt{EDT(x,y)}$ is the exact Euclidean distance, $MDT(x,y)$ is the Manhattan (city-block) distance, and $CDT(x,y)$ is the Chessboard distance. These distances are defined by

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

Note that for each point $p \in B$, the distance is zero.
The idea is to find the minimal i and j for each (x,y) in D. We define the minimum of the empty set to be ∞, and use the rule $z + \infty = \infty$ for all z. Now we find with some calculation

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

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

A sequential solution

The algorithm can be summarized as follows. In a first phase each column C_x (defined by points (x,y) with x fixed) is scanned separately. For each point (x,y) on C_x, the distance $G(x,y)$ of (x,y) to nearest points of $C_x \cap B$ is determined. In a second phase each row R_y (defined by points (x,y) with y fixed) is scanned separately, and for each point (x,y) on R_y the minimum of $(x-x')^2 + G(x',y)^2$ for EDT, $|x-x'| + G(x',y)$ for MDT, and $|x-x'|$ **max** $G(x',y)$ for CDT is determined, where (x',y) ranges over row R_y.
We give the pseudo code fragment from [AM2000] which calculates the distance transform, with the use of the definitions above. Information about the construction and correctness of this fragment can be found in [AM2000].

Phase 1:

```
for all x ∈ [0..m−1] do
  if b(x,0) then
    G[x,0] := 0;
  else
    G[x,0] := ∞;
  fi;
  for y := 1 to n−1 do
    if b(x,y) then
      G[x,y] := 0;
    else
      G[x,y] := 1 + G[x,y−1];
    fi;
  od;
  for y := n−2 downto 0 do
    if G[x,y+1] < G[x,y] then
      G[x,y] := 1 + G[x,y+1];
    fi;
  od;
od;
```

Phase 2:

```
for all y ∈ [0..n−1] do
  q := 0;
  s[0] := 0;
  t[0] := 0;
  for u := 1 to m−1 do
    while q ≥ 0 ∧ f(t[q],y,s[q]) > f(t[q],y,u) do
      q := q−1;
    od;
    if q < 0 then
      q := 0;
      s[0] := u;
    else
      w := 1 + Sep(s[q],y,u);
      if w < m then
        q := q+1;
```

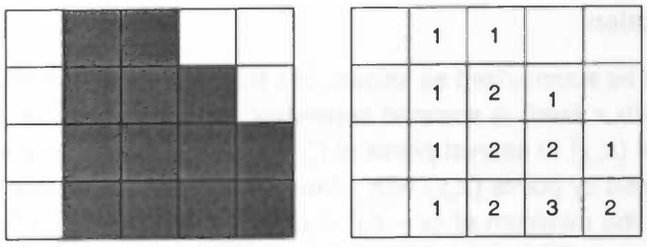

Figure 6.1: An example of the Manhattan distance transform.

```
            s[q] := u;
            t[q] := w;
          fi;
        fi;
      od;
      for u := m − 1 downto 0 do
        dt[u,y] := f(u,y,s[q]);
        if u = t[q] then
          q := q − 1;
        fi;
      od;
    od;
```

The array `dt` is the distance transform of the binary image b defined on the rectangular grid $[0..m-1] \times [0..n-1]$.
The choice of function f and *Sep* depends on the metric that is used, i.e.

EDT:
$f(x,y,i) = (x-i)^2 + G[i,y]^2$
$Sep(i,u) = (u^2 - i^2 + G[u,y])^2 - G[i,y]^2)$ **div** $(2(u-i))$

MDT:
$f(x,y,i) = |x-i| + G[i,y]$
$Sep(i,y,u) =$ **if** $G[u,y] \geq G[i,y] + u - i$ **then** ∞
else if $G[i,y] > G[u,y] + u - i$ **then** $-\infty$
else $(G[u,y] - G[i,y] + h + i)$ **div** 2

CDT:
$f(x,y,i) = |x-i|$ **max** $G[i,y]$
$Sep(i,y,u) =$ **if** $G[i,y] \leq G[u,y]$ **then**
$(i + G[u,y])$ **max** $((i+u)$ **div** $2)$
else
$(u - G[i,y])$ **min** $((i+u)$ **div** $2)$

An example of a distance transform is given in figure 6.1. In the image on the left the white pixels form the set B. In the right the Manhattan distance transform (MDT) is given. Note that for readability the zeros have been replaced by spaces. This conventions is used in all figures in this chapter.

A distributed solution

Since the computation per row and per column is independent of the computation of the other rows and columns, the algorithm is well suited for parallelization. In [AM2000] a parallel solution is given for shared memory computers.
In this paragraph we show how the distance transform fragment can be distributed over a set of processes with distributed memory. The rows and columns are distributed equally over the processes. We define $Columns_k$ as the set of columns and $Rows_k$ as the set of rows that belong to process k.
The distributed algorithm is

DTpar_k:
1: **apply** *Phase 1* **to** $Columns_k$;
2: **forall** $p \in Processes$ **do**
send $Columns_k \cap Rows_p$ **of** G **to** p;
od;
3: **forall** $p \in Processes$ **do**
receive $Rows_k \cap Columns_p$ **of** G **from** p;
od;
4: **apply** *Phase 2* **to** $Rows_k$.

In figure 6.2 an example of the distributed Manhattan distance transform is given. Figure a. is the original image. The white pixels form the set B. In b. a distribution of the columns is shown. $Columns_0 = \{A,B\}$, $Columns_1 = \{C,D\}$, and $Columns_2 = \{E,F\}$. In c., G is shown after the first (forward) scan in *Phase 1*. In d., G is shown after the second (backward) scan in *Phase 1*. In e. G is shown after the statements 2 and 3 of DTpar_k. In this example, $\{C5, C6, D5, D6\}$ of G are sent from process 1 to process 2. In e., G is shown after *Phase 2*, statement 4 of DTpar_k.

The feature transform

In the feature transform `ft` of an image, for each pixel p, the nearest pixel $q \in B$ is calculated. This is done in the following way. In the distance transform the nearest pixel is updated for each pixel whose G or dt value is updated. In the fragment below, q is the array containing the nearest pixels. The value *null* is assigned to the array q for those pixels whose nearest pixel is still unknown.

Phase 1:
for all $x \in [0..m-1]$ **do**
if $b(x,0)$ **then**
$q[x,0] := (x,0)$;
$G[x,0] := 0$;
else
$q[x,0] :=$ *null*;
$G[x,0] := \infty$;
fi;
for $y := 1$ **to** $n-1$ **do**
if $b(x,y)$ **then**
$q[x,y] := (x,y)$;
$G[x,y] := 0$;

```
      else
        q[x,y] := q[x,y-1];
        G[x,y] := 1 + G[x,y-1];
      fi;
    od;
    for y := n-2 downto 0 do
      if G[x,y+1] < G[x,y] then
        q[x,y] := q[x,y+1];
        G[x,y] := 1 + G[x,y+1];
      fi;
    od;
  od;
```

Phase 2:

```
  for all y ∈ [0..n-1] do
    q := 0;
    s[0] := 0;
    t[0] := 0;
    for u := 1 to m-1 do
      while q ≥ 0 ∧ f(t[q],y,s[q]) > f(t[q],y,u) do
        q := q-1;
      od;
      if q < 0 then
        q := 0;
        s[0] := u;
      else
        w := 1 + Sep(s[q],y,u);
        if w < m then
          q := q+1;
          s[q] := u;
          t[q] := w;
        fi;
      fi;
    od;
    for u := m-1 downto 0 do
      ft[u,y] := q[u,s[q]];
      dt[u,y] := f(u,y,s[q]);
      if u = t[q] then
        q := q-1;
      fi;
    od;
  od;
```

The algorithm above can be parallelized just like the distance transform, i.e.

FTpar$_k$:

```
1:    apply Phase 1 to Columns_k;
2:    forall p ∈ Processes do
        send Columns_k ∩ Rows_p of G to p;
        send Columns_k ∩ Rows_p of q to p;
```

```
        od;
3:      forall p ∈ Processes do
            receive Rows_k ∩ Columns_p of G from p;
            receive Rows_k ∩ Columns_p of q from p;
        od;
4:      apply Phase 2 to Rows_k;
```

In figure 6.3 an example of the parallel feature transform is given. This example corresponds with the example in figure 6.2. The value in the bottom of each pixel is the nearest pixel in B, i.e. array q.

6.3 Merging of connected components

In practice, labeling of connected components on real-life images yields an oversegmentation. Therefore, in this section we present a filter that merges some connected components with other connected components. In this case we use the size of the connected components as a merging criterion. However, other criteria can be used as well.

We assume that from an image f, the harvested arrays `lab` and `area` are available, and define that there exists a function

$$keepsize : \mathbb{Z} \rightarrow \{\texttt{TRUE}, \texttt{FALSE}\}.$$

We define the function b from section 6.2 as follows.

$$b(x,y) = keepsize(\texttt{area}[(x,y)])$$

The feature transform is rewritten as follows. $\texttt{lab}[x]$ is only updated if its nearest pixel is changed. Other structures stored in an array can also be updated at these moments, e.g. array `par` and `area`.

```
Phase 1:
  for all x ∈ [0..m−1] do
    if b(x,0) then
      G[x,0] := 0;
    else
      lab[x,0] := ⊥;
      G[x,0] := ∞;
    fi;
    for y := 1 to n−1 do
      if b(x,y) then
        G[x,y] := 0;
      else
        lab[x,y] := lab[x,y−1];
        G[x,y] := 1+G[x,y−1];
      fi;
    od;
    for y := n−2 downto 0 do
      if G[x,y+1] < G[x,y] then
```

```
            lab[x,y] := lab[x,y+1];
            G[x,y] := 1+G[x,y+1];
        fi;
    od;
od;
```

Phase 2:

```
for all y ∈ [0..n−1] do
    q := 0;
    s[0] := 0;
    t[0] := 0;
    for u := 1 to m−1 do
        while q ≥ 0 ∧ f(t[q],y,s[q]) > f(t[q],y,u) do
            q := q−1;
        od;
        if q < 0 then
            q := 0;
            s[0] := u;
        else
            w := 1 + Sep(s[q],y,u);
            if w < m then
                q := q+1;
                s[q] := u;
                t[q] := w;
            fi;
        fi;
    od;
    for u := m−1 downto 0 do
        lab[u,y] := lab[u,s[q]];
        dt[u,y] := f(u,y,s[q]);
        if u = t[q] then
            q := q−1;
        fi;
    od;
od.
```

After Phase 2 has been executed, the array `lab` is the labelling of the new image.
Note: When one prefers to merge components in the original image, the array `lab` must be replaced by the input image in the algorithm above.

Examples

In figure 6.4 an example is given of the connected component merging algorithm. In this example $keepsize(a) = a > 3$, which means all connected components of size three and smaller have to be merged with the other connected components.
In a. the array `lab` is shown. There are twelve connected components, named from A to L. In b. the harvested array `area` is shown. The shaded pixels belong to a connected component that is too small. In c. the array `lab` is shown for the pixels whose `lab` has been changed because of their size. Figure d. is the new array `lab`. Note that some small components have been broken up into two or more pieces, e.g. component J.

In figure 6.5 an application is shown. On the left a picture is shown of a number of circles of different sizes. In this example we are only interested in the circles of a bounded range of sizes. The right image is the new image, when all other circles have been merged with the background.

Distributed Connected Component Merging

We have parallelized the connected component merging algorithm the same way as the other variations of the distance transform, i.e.

CCR_{par_k}:

1: **apply** *Phase 1* **to** $Columns_k$;
2: **forall** $p \in Processes$ **do**
  send $Columns_k \cap Rows_p$ **of** G **to** p;
  send $Columns_k \cap Rows_p$ **of** `lab` **to** p;
od;
3: **forall** $p \in Processes$ **do**
  receive $Rows_k \cap Columns_p$ **of** G **from** p;
  receive $Rows_k \cap Columns_p$ **of** `lab` **from** p;
od;
4: **apply** *Phase 2* **to** $Rows_k$;

We have implemented and tested this algorithm for distributed memory computers. It appeared to be highly scalable in terms of the input image and the number of processes.

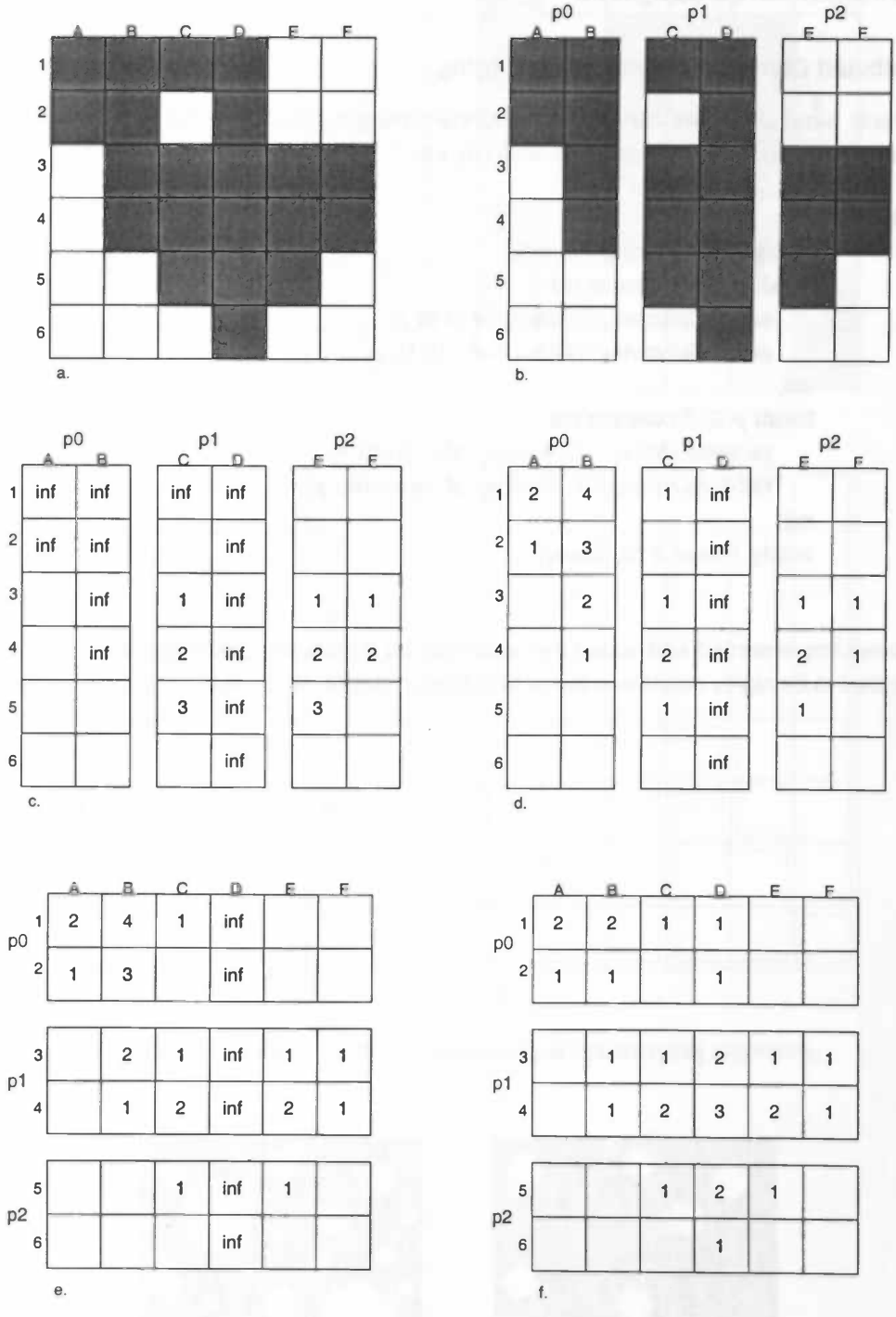

Figure 6.2: An example of the parallel distance transform.

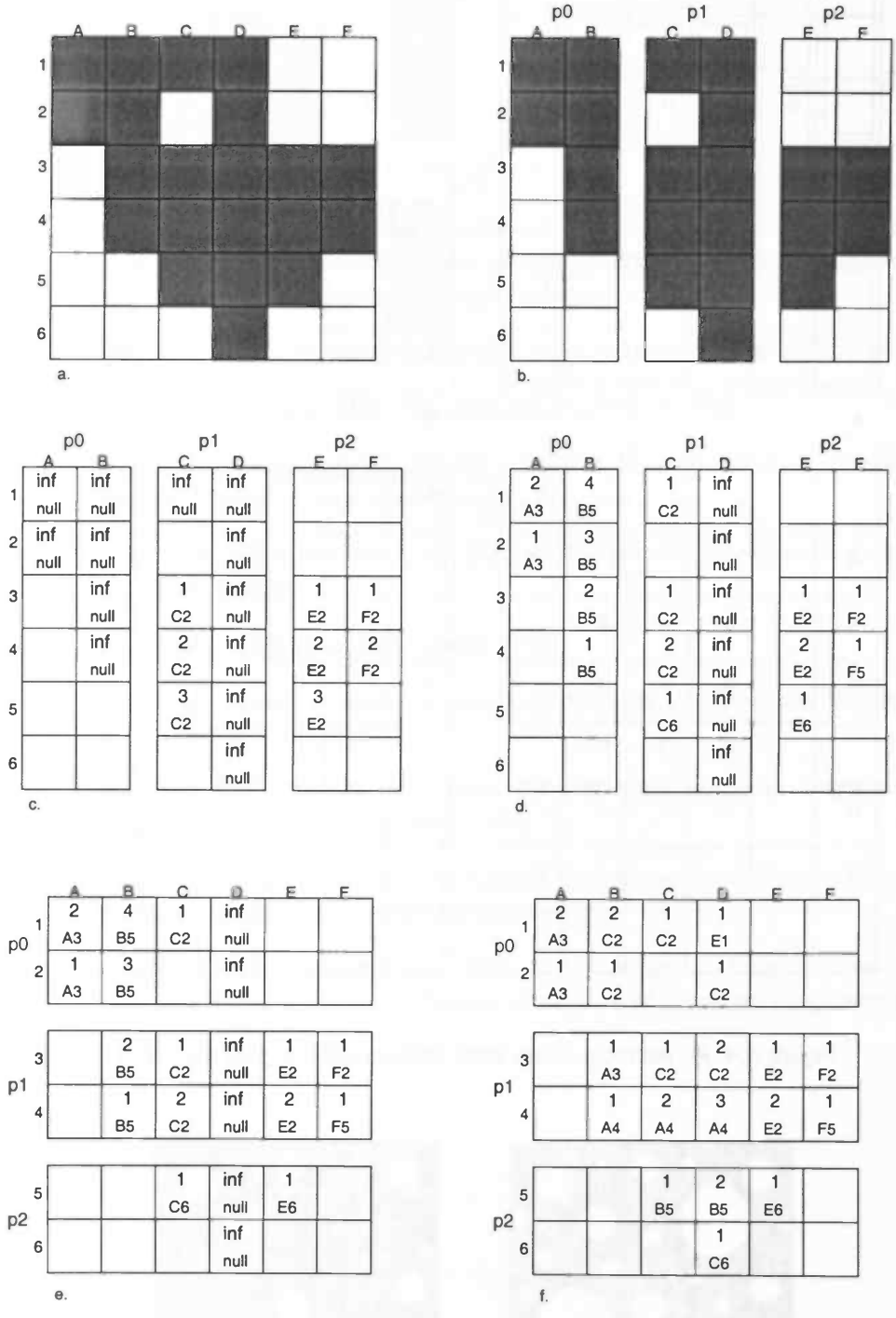

Figure 6.3: An example of the parallel feature transform.

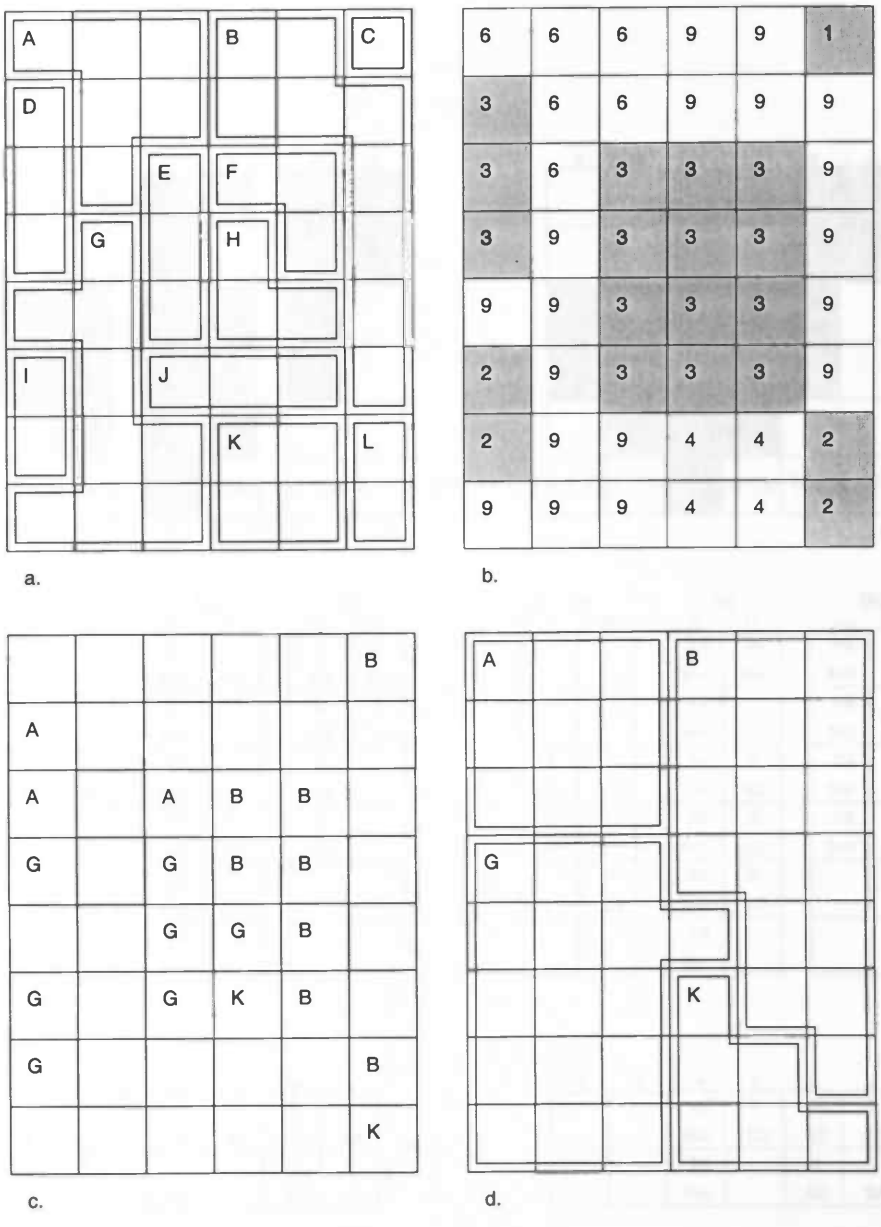

Figure 6.4: An example of the small component merging algorithm.

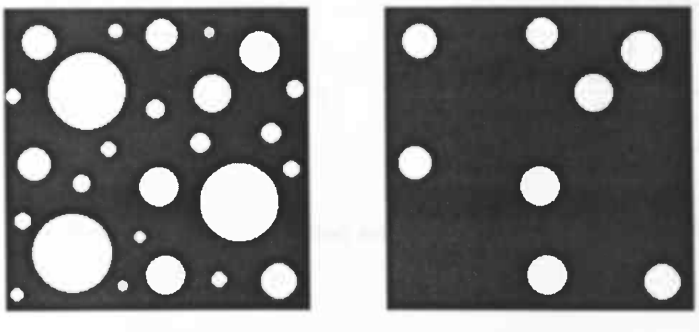

Figure 6.5: An example of the component merging algorithm.

Bibliography

[AM2000] Meijster A., Roerdink J.B.T.M., Hesselink W.H.: A general algorithm for computing distance transforms in linear time. In: Mathematical Morphology and its Applications to Image and Signal Processing, 2000, pp. 331-340.

[Cor90] Cormen T.H., Leisserson C.E., Rivest E.L.: Introduction to Algorithms (1990)

[Gro] Gropp W.: Tutorial on MPI: The Message-Passing Interface

[MPIStd] Message Passing Interface Forum : MPI: A Message-Passing Interface Standard (`http://www.mpi-forum.org`)

[MW98] Meijster A., Wilkinson M.H.F.: An Efficient Algorithm for Morphological Area Operators (1998)

[Roe98] Roerdink J.B.T.M.: Computer Vision (1998)

[Son99] Sonka M., Hlavac V., Boyle R.: Image Processing, Analysis, and Machine Vision (2nd edition) (1999)

[Tar75] Tarjan R.E.: Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215-225 (1975)

[Tar83] Tarjan R.E.: Data structures and network algorithms. SIAM 1983 (Regional Conference Series in Applied Mathematics)

[WHH01] Hesselink W.H., Meijster A., Bron C.: Concurrent Determination of Connected Components. *To appear in Science of Computer Programming* (2001)

Appendix A

Test results

On the next pages the timing results and efficiency of the performance tests on the DEC/Alpha cluster can be found.

		Timings (ms)					
image	#P	128	256	512	1024	2048	4096
empty	1	19	82	338	1483	6208	24281
	2	12	45	174	739	3067	12351
	3	9	31	121	484	2149	8353
	4	7	24	89	364	1533	6237
vertical	1	19	77	316	1419	5668	23010
	2	12	43	162	710	2882	11637
	3	10	31	110	480	1982	7761
	4	8	25	86	357	1466	5724
horizontal	1	19	79	316	1397	5776	23536
	2	11	42	166	712	2985	12112
	3	8	28	110	477	1963	8203
	4	7	23	81	359	1464	6068
comb	1	18	78	316	1396	5715	22861
	2	12	44	164	710	2967	11584
	3	11	33	114	484	1939	7842
	4	10	27	90	363	1448	5850
random	1	20	83	336	1473	5974	24208
	2	11	45	173	742	3037	12400
	3	10	32	116	492	2061	8120
	4	9	24	88	375	1495	6249
music	1	20	82	370	1474	6060	24490
	2	12	44	172	752	3019	12896
	3	11	32	117	494	2049	8263
	4	7	24	87	364	1482	6377
ct	1	18	75	325	1467	6025	24172
	2	11	42	168	732	3089	12391
	3	9	30	115	490	2180	8434
	4	8	23	86	368	1487	6255

Table A.1: Performance test of the implementation of Tarjan's disjoint set algorithm. The forward version is used with a *mpart* of 1.

		Efficiency per processor					
image	#P	128	256	512	1024	2048	4096
empty	2	79.2%	91.1%	97.1%	100.3%	101.2%	98.3%
	3	70.4%	88.2%	93.1%	102.1%	96.3%	96.9%
	4	67.9%	85.4%	94.9%	101.9%	101.2%	97.3%
vertical	2	79.2%	89.5%	97.5%	99.9%	98.3%	98.9%
	3	63.3%	82.8%	95.8%	98.5%	95.3%	98.8%
	4	59.4%	77.0%	91.9%	99.4%	96.7%	100.5%
horizontal	2	86.4%	94.0%	95.2%	98.1%	96.8%	97.2%
	3	79.2%	94.0%	95.8%	97.6%	98.1%	95.6%
	4	67.9%	85.9%	97.5%	97.3%	98.6%	97.0%
comb	2	75.0%	88.6%	96.3%	98.3%	96.3%	98.7%
	3	54.5%	78.8%	92.4%	96.1%	98.2%	97.2%
	4	45.0%	72.2%	87.8%	96.1%	98.7%	97.7%
random	2	90.9%	92.2%	97.1%	99.3%	98.4%	97.6%
	3	66.7%	86.5%	96.6%	99.8%	96.6%	99.4%
	4	55.6%	86.5%	95.5%	98.2%	99.9%	96.8%
music	2	83.3%	93.2%	107.6%	98.0%	100.4%	95.0%
	3	60.6%	85.4%	105.4%	99.5%	98.6%	98.8%
	4	71.4%	85.4%	106.3%	101.2%	102.2%	96.0%
ct	2	81.8%	89.3%	96.7%	100.2%	97.5%	97.5%
	3	66.7%	83.3%	94.2%	99.8%	92.1%	95.5%
	4	56.2%	81.5%	94.5%	99.7%	101.3%	96.6%

Table A.2: The efficiency calculated from table A.1

		Timings (ms)					
image	#P	128	256	512	1024	2048	4096
empty	1	20	82	346	1500	6194	25086
	2	11	44	171	757	3015	12394
	3	9	31	117	496	2074	8537
	4	7	24	88	372	1503	6346
vertical	1	18	77	317	1382	5710	23062
	2	12	41	160	710	2909	11571
	3	11	33	114	469	1913	7625
	4	11	27	89	357	1430	5945
horizontal	1	19	79	323	1416	5808	24286
	2	11	42	162	712	2992	12077
	3	9	30	109	480	1984	7903
	4	7	23	82	357	1462	5981
comb	1	19	77	313	1388	5675	22460
	2	12	43	163	712	2902	11620
	3	11	33	114	483	1981	7960
	4	10	28	97	361	1436	5762
random	1	20	84	352	1435	5934	24926
	2	12	45	171	739	3138	12515
	3	10	32	122	486	1996	8187
	4	9	25	88	369	1519	6403
music	1	20	84	338	1500	6521	24588
	2	12	44	172	738	3072	12317
	3	10	32	115	494	2029	8169
	4	13	24	88	366	1475	6174
ct	1	18	76	327	1472	6197	24261
	2	11	41	166	732	3101	12454
	3	9	31	116	490	2030	8520
	4	7	23	87	371	1523	6240

Table A.3: Performance test of the implementation of Tarjan's disjoint set algorithm. The forward version is used with a *mpart* of 4.

		Efficiency per processor					
image	#P	128	256	512	1024	2048	4096
empty	2	90.9%	93.2%	101.2%	99.1%	102.7%	101.2%
	3	74.1%	88.2%	98.6%	100.8%	99.5%	98.0%
	4	71.4%	85.4%	98.3%	100.8%	103.0%	98.8%
vertical	2	75.0%	93.9%	99.1%	97.3%	98.1%	99.7%
	3	54.5%	77.8%	92.7%	98.2%	99.5%	100.8%
	4	40.9%	71.3%	89.0%	96.8%	99.8%	97.0%
horizontal	2	86.4%	94.0%	99.7%	99.4%	97.1%	100.5%
	3	70.4%	87.8%	98.8%	98.3%	97.6%	102.4%
	4	67.9%	85.9%	98.5%	99.2%	99.3%	101.5%
comb	2	79.2%	89.5%	96.0%	97.5%	97.8%	96.6%
	3	57.6%	77.8%	91.5%	95.8%	95.5%	94.1%
	4	47.5%	68.8%	80.7%	96.1%	98.8%	97.4%
random	2	83.3%	93.3%	102.9%	97.1%	94.6%	99.6%
	3	66.7%	87.5%	96.2%	98.4%	99.1%	101.5%
	4	55.6%	84.0%	100.0%	97.2%	97.7%	97.3%
music	2	83.3%	95.5%	98.3%	101.6%	106.1%	99.8%
	3	66.7%	87.5%	98.0%	101.2%	107.1%	100.3%
	4	38.5%	87.5%	96.0%	102.5%	110.5%	99.6%
ct	2	81.8%	92.7%	98.5%	100.5%	99.9%	97.4%
	3	66.7%	81.7%	94.0%	100.1%	101.8%	94.9%
	4	64.3%	82.6%	94.0%	99.2%	101.7%	97.2%

Table A.4: The efficiency calculated from table A.3

		Timings (ms)					
image	#P	128	256	512	1024	2048	4096
empty	1	20	84	337	1464	6024	24087
	2	12	44	176	750	3009	12914
	3	9	31	116	486	2136	8256
	4	8	25	89	373	1517	6342
vertical	1	19	77	317	1378	5676	22982
	2	13	43	164	716	2907	11762
	3	12	32	116	473	1956	8243
	4	14	29	88	361	1442	5866
horizontal	1	18	79	321	1427	5938	23282
	2	11	42	165	710	3008	11866
	3	9	29	106	475	1972	7849
	4	7	23	85	357	1469	6079
comb	1	18	78	315	1376	5608	22742
	2	13	42	164	705	2913	12177
	3	18	37	116	484	1962	7807
	4	21	34	93	365	1440	5887
random	1	20	85	351	1445	6197	24404
	2	12	45	172	755	3053	12387
	3	11	32	116	492	2050	8389
	4	11	25	88	375	1530	6301
music	1	20	83	339	1449	5923	24481
	2	13	44	171	727	3124	12531
	3	11	31	115	492	2032	8274
	4	8	24	89	374	1519	6314
ct	1	18	76	327	1458	6059	25463
	2	11	43	170	753	3082	12373
	3	14	32	122	500	2021	8183
	4	7	23	86	370	1522	6392

Table A.5: Performance test of the implementation of Tarjan's disjoint set algorithm. The forward version is used with a *mpart* of 16.

		Efficiency per processor					
image	#P	128	256	512	1024	2048	4096
empty	2	83.3%	95.5%	95.7%	97.6%	100.1%	93.3%
	3	74.1%	90.3%	96.8%	100.4%	94.0%	97.3%
	4	62.5%	84.0%	94.7%	98.1%	99.3%	95.0%
vertical	2	73.1%	89.5%	96.6%	96.2%	97.6%	97.7%
	3	52.8%	80.2%	91.1%	97.1%	96.7%	92.9%
	4	33.9%	66.4%	90.1%	95.4%	98.4%	97.9%
horizontal	2	81.8%	94.0%	97.3%	100.5%	98.7%	98.1%
	3	66.7%	90.8%	100.9%	100.1%	100.4%	98.9%
	4	64.3%	85.9%	94.4%	99.9%	101.1%	95.7%
comb	2	69.2%	92.9%	96.0%	97.6%	96.3%	93.4%
	3	33.3%	70.3%	90.5%	94.8%	95.3%	97.1%
	4	21.4%	57.4%	84.7%	94.2%	97.4%	96.6%
random	2	83.3%	94.4%	102.0%	95.7%	101.5%	98.5%
	3	60.6%	88.5%	100.9%	97.9%	100.8%	97.0%
	4	45.5%	85.0%	99.7%	96.3%	101.3%	96.8%
music	2	76.9%	94.3%	99.1%	99.7%	94.8%	97.7%
	3	60.6%	89.2%	98.3%	98.2%	97.2%	98.6%
	4	62.5%	86.5%	95.2%	96.9%	97.5%	96.9%
ct	2	81.8%	88.4%	96.2%	96.8%	98.3%	102.9%
	3	42.9%	79.2%	89.3%	97.2%	99.9%	103.7%
	4	64.3%	82.6%	95.1%	98.5%	99.5%	99.6%

Table A.6: The efficiency calculated from table A.5

		Timings (ms)					
image	#P	128	256	512	1024	2048	4096
empty	1	20	88	351	1510	6019	24735
	2	12	44	172	752	3107	12684
	3	9	31	117	507	2096	8274
	4	7	23	89	382	1504	6284
vertical	1	18	77	315	1405	5757	23150
	2	21	49	171	716	2920	11601
	3	23	43	125	481	1918	7685
	4	22	40	101	362	1434	5747
horizontal	1	19	77	321	1406	5719	23413
	2	11	42	166	718	3012	11924
	3	8	29	107	470	1975	7896
	4	7	23	83	360	1455	5966
comb	1	19	76	313	1378	5610	23210
	2	21	48	169	721	2904	11437
	3	23	47	126	482	1976	7784
	4	39	45	106	373	1560	5871
random	1	20	82	340	1445	6101	23660
	2	12	45	174	750	3039	12586
	3	10	34	121	505	2004	8136
	4	9	25	94	381	1687	6288
music	1	20	83	335	1498	6047	24490
	2	12	44	175	749	3082	12195
	3	10	35	119	492	2075	8254
	4	7	26	88	379	1492	6181
ct	1	18	76	329	1478	6011	24153
	2	11	42	171	743	3090	12358
	3	12	33	117	491	2076	8255
	4	10	25	86	372	1526	6462

Table A.7: Performance test of the implementation of Tarjan's disjoint set algorithm. The forward version is used with a *mpart* of 64.

		Efficiency per processor					
image	#P	128	256	512	1024	2048	4096
empty	2	83.3%	100.0%	102.0%	100.4%	96.9%	97.5%
	3	74.1%	94.6%	100.0%	99.3%	95.7%	99.6%
	4	71.4%	95.7%	98.6%	98.8%	100.0%	98.4%
vertical	2	42.9%	78.6%	92.1%	98.1%	98.6%	99.8%
	3	26.1%	59.7%	84.0%	97.4%	100.1%	100.4%
	4	20.5%	48.1%	78.0%	97.0%	100.4%	100.7%
horizontal	2	86.4%	91.7%	96.7%	97.9%	94.9%	98.2%
	3	79.2%	88.5%	100.0%	99.7%	96.5%	98.8%
	4	67.9%	83.7%	96.7%	97.6%	98.3%	98.1%
comb	2	45.2%	79.2%	92.6%	95.6%	96.6%	101.5%
	3	27.5%	53.9%	82.8%	95.3%	94.6%	99.4%
	4	12.2%	42.2%	73.8%	92.4%	89.9%	98.8%
random	2	83.3%	91.1%	97.7%	96.3%	100.4%	94.0%
	3	66.7%	80.4%	93.7%	95.4%	101.5%	96.9%
	4	55.6%	82.0%	90.4%	94.8%	90.4%	94.1%
music	2	83.3%	94.3%	95.7%	100.0%	98.1%	100.4%
	3	66.7%	79.0%	93.8%	101.5%	97.1%	98.9%
	4	71.4%	79.8%	95.2%	98.8%	101.3%	99.1%
ct	2	81.8%	90.5%	96.2%	99.5%	97.3%	97.7%
	3	50.0%	76.8%	93.7%	100.3%	96.5%	97.5%
	4	45.0%	76.0%	95.6%	99.3%	98.5%	93.4%

Table A.8: The efficiency calculated from table A.7

		Timings (ms)					
image	#P	128	256	512	1024	2048	4096
empty	1	20	84	340	1501	6094	23955
	2	12	47	173	757	3005	12723
	3	9	32	117	483	2073	8664
	4	7	24	89	382	1520	6205
vertical	1	19	77	316	1386	5654	22847
	2	26	72	195	740	2934	11601
	3	39	87	165	537	1979	7782
	4	46	117	177	442	1725	5913
horizontal	1	19	78	324	1426	5808	23628
	2	11	42	188	714	3013	11897
	3	8	28	110	484	2009	7909
	4	7	23	83	358	1476	6076
comb	1	19	77	313	1379	5762	24518
	2	26	69	194	731	2915	11721
	3	40	98	179	532	2038	7726
	4	48	115	198	463	1581	5902
random	1	20	84	339	1462	5938	24632
	2	14	45	175	722	2986	12487
	3	12	35	118	505	2012	8239
	4	11	31	90	361	1501	6338
music	1	20	83	336	1470	5967	24292
	2	13	45	172	752	3102	12235
	3	16	38	122	515	2069	8697
	4	10	29	90	368	1488	6203
ct	1	18	75	327	1467	5972	24598
	2	14	45	168	729	3107	12755
	3	14	49	206	499	2108	8303
	4	13	32	92	378	1522	6120

Table A.9: Performance test of the implementation of Tarjan's disjoint set algorithm. The forward version is used with a *mpart* of 256.

		Efficiency per processor					
image	#P	128	256	512	1024	2048	4096
empty	2	83.3%	89.4%	98.3%	99.1%	101.4%	94.1%
	3	74.1%	87.5%	96.9%	103.6%	98.0%	92.2%
	4	71.4%	87.5%	95.5%	98.2%	100.2%	96.5%
vertical	2	36.5%	53.5%	81.0%	93.6%	96.4%	98.5%
	3	16.2%	29.5%	63.8%	86.0%	95.2%	97.9%
	4	10.3%	16.5%	44.6%	78.4%	81.9%	96.6%
horizontal	2	86.4%	92.9%	86.2%	99.9%	96.4%	99.3%
	3	79.2%	92.9%	98.2%	98.2%	96.4%	99.6%
	4	67.9%	84.8%	97.6%	99.6%	98.4%	97.2%
comb	2	36.5%	55.8%	80.7%	94.3%	98.8%	104.6%
	3	15.8%	26.2%	58.3%	86.4%	94.2%	105.8%
	4	9.9%	16.7%	39.5%	74.5%	91.1%	103.9%
random	2	71.4%	93.3%	96.9%	101.2%	99.4%	98.6%
	3	55.6%	80.0%	95.8%	96.5%	98.4%	99.7%
	4	45.5%	67.7%	94.2%	101.2%	98.9%	97.2%
music	2	76.9%	92.2%	97.7%	97.7%	96.2%	99.3%
	3	41.7%	72.8%	91.8%	95.1%	96.1%	93.1%
	4	50.0%	71.6%	93.3%	99.9%	100.3%	97.9%
ct	2	64.3%	83.3%	97.3%	100.6%	96.1%	96.4%
	3	42.9%	51.0%	52.9%	98.0%	94.4%	98.8%
	4	34.6%	58.6%	88.9%	97.0%	98.1%	100.5%

Table A.10: The efficiency calculated from table A.9

		Timings (ms)					
image	#P	128	256	512	1024	2048	4096
empty	1	19	83	352	1472	6260	24401
	2	12	46	170	746	3108	12631
	3	9	33	118	499	2266	8189
	4	8	25	88	376	1521	6337
vertical	1	19	76	317	1383	5759	23071
	2	30	73	228	835	3197	12565
	3	42	83	241	722	2478	9025
	4	45	112	263	753	2150	7379
horizontal	1	19	79	312	1426	5806	23300
	2	11	41	167	716	3009	11853
	3	9	30	113	479	1998	8377
	4	7	23	84	358	1454	5934
comb	1	18	76	316	1377	5703	22622
	2	25	71	228	828	3144	12296
	3	39	100	242	727	2493	8867
	4	49	117	271	765	2200	7595
random	1	20	83	350	1427	6129	24842
	2	13	46	173	737	3049	12938
	3	11	36	120	484	2066	8384
	4	12	34	92	373	1500	6282
music	1	20	82	332	1497	5915	24692
	2	12	45	173	729	3045	12469
	3	11	36	121	493	2065	8478
	4	8	29	93	374	1515	6120
ct	1	18	75	327	1471	5972	24547
	2	13	44	169	739	3063	12524
	3	14	43	141	530	2064	8236
	4	12	29	93	375	1518	6411

Table A.11: Performance test of the implementation of Tarjan's disjoint set algorithm. The forward version is used with a *mpart* of ∞.

		Efficiency per processor					
image	#P	128	256	512	1024	2048	4096
empty	2	79.2%	90.2%	103.5%	98.7%	100.7%	96.6%
	3	70.4%	83.8%	99.4%	98.3%	92.1%	99.3%
	4	59.4%	83.0%	100.0%	97.9%	102.9%	96.3%
vertical	2	31.7%	52.1%	69.5%	82.8%	90.1%	91.8%
	3	15.1%	30.5%	43.8%	63.9%	77.5%	85.2%
	4	10.6%	17.0%	30.1%	45.9%	67.0%	78.2%
horizontal	2	86.4%	96.3%	93.4%	99.6%	96.5%	98.3%
	3	70.4%	87.8%	92.0%	99.2%	96.9%	92.7%
	4	67.9%	85.9%	92.9%	99.6%	99.8%	98.2%
comb	2	36.0%	53.5%	69.3%	83.2%	90.7%	92.0%
	3	15.4%	25.3%	43.5%	63.1%	76.3%	85.0%
	4	9.2%	16.2%	29.2%	45.0%	64.8%	74.5%
random	2	76.9%	90.2%	101.2%	96.8%	100.5%	96.0%
	3	60.6%	76.9%	97.2%	98.3%	98.9%	98.8%
	4	41.7%	61.0%	95.1%	95.6%	102.2%	98.9%
music	2	83.3%	91.1%	96.0%	102.7%	97.1%	99.0%
	3	60.6%	75.9%	91.5%	101.2%	95.5%	97.1%
	4	62.5%	70.7%	89.2%	100.1%	97.6%	100.9%
ct	2	69.2%	85.2%	96.7%	99.5%	97.5%	98.0%
	3	42.9%	58.1%	77.3%	92.5%	96.4%	99.3%
	4	37.5%	64.7%	87.9%	98.1%	98.4%	95.7%

Table A.12: The efficiency calculated from table A.11

		Timings (ms)					
image	#P	128	256	512	1024	2048	4096
empty	1	19	79	328	1445	5967	24073
	2	11	67	168	737	3143	12670
	3	9	30	113	489	2041	8090
	4	7	23	88	371	1494	6281
vertical	1	17	70	285	1257	5191	20783
	2	13	48	180	820	3710	14966
	3	11	35	126	564	2505	10249
	4	9	27	96	421	1839	7619
horizontal	1	16	67	274	1255	5200	21071
	2	9	36	141	648	2655	10617
	3	7	26	95	431	1812	7052
	4	6	20	71	320	1330	5319
comb	1	17	73	296	1285	5309	22099
	2	15	55	214	949	4309	17706
	3	14	48	180	833	3860	16344
	4	13	42	163	737	3388	14730
random	1	19	78	321	1423	5846	23787
	2	11	43	165	722	2988	11954
	3	9	30	112	488	2006	8032
	4	10	24	83	363	1497	6091
music	1	18	78	319	1422	5833	23757
	2	12	42	164	722	3039	11990
	3	9	30	111	488	2011	8066
	4	9	24	84	358	1481	6040
ct	1	17	71	305	1421	5744	23502
	2	10	39	159	704	2943	11839
	3	14	29	108	475	1978	8172
	4	7	23	81	357	1639	6008

Table A.13: Performance test of the implementation of Tarjan's disjoint set algorithm. The backward version is used with a *mpart* of 1.

		Efficiency per processor					
image	#P	128	256	512	1024	2048	4096
empty	2	86.4%	59.0%	97.6%	98.0%	94.9%	95.0%
	3	70.4%	87.8%	96.8%	98.5%	97.5%	99.2%
	4	67.9%	85.9%	93.2%	97.4%	99.8%	95.8%
vertical	2	65.4%	72.9%	79.2%	76.6%	70.0%	69.4%
	3	51.5%	66.7%	75.4%	74.3%	69.1%	67.6%
	4	47.2%	64.8%	74.2%	74.6%	70.6%	68.2%
horizontal	2	88.9%	93.1%	97.2%	96.8%	97.9%	99.2%
	3	76.2%	85.9%	96.1%	97.1%	95.7%	99.6%
	4	66.7%	83.8%	96.5%	98.0%	97.7%	99.0%
comb	2	56.7%	66.4%	69.2%	67.7%	61.6%	62.4%
	3	40.5%	50.7%	54.8%	51.4%	45.8%	45.1%
	4	32.7%	43.5%	45.4%	43.6%	39.2%	37.5%
random	2	86.4%	90.7%	97.3%	98.5%	97.8%	99.5%
	3	70.4%	86.7%	95.5%	97.2%	97.1%	98.7%
	4	47.5%	81.2%	96.7%	98.0%	97.6%	97.6%
music	2	75.0%	92.9%	97.3%	98.5%	96.0%	99.1%
	3	66.7%	86.7%	95.8%	97.1%	96.7%	98.2%
	4	50.0%	81.2%	94.9%	99.3%	98.5%	98.3%
ct	2	85.0%	91.0%	95.9%	100.9%	97.6%	99.3%
	3	40.5%	81.6%	94.1%	99.7%	96.8%	95.9%
	4	60.7%	77.2%	94.1%	99.5%	87.6%	97.8%

Table A.14: The efficiency calculated from table A.13

		Timings (ms)					
image	#P	128	256	512	1024	2048	4096
empty	1	20	80	327	1446	5887	24077
	2	11	43	167	843	3050	12087
	3	9	30	111	485	2031	8290
	4	7	23	86	361	1513	6263
vertical	1	16	71	288	1243	5163	21481
	2	12	46	183	820	3689	15195
	3	11	34	125	557	2493	10128
	4	9	27	104	419	1838	7661
horizontal	1	17	68	275	1280	5207	21001
	2	9	36	140	645	2678	10565
	3	7	25	95	429	1813	7083
	4	6	20	72	323	1324	5316
comb	1	18	72	295	1299	5340	21419
	2	14	54	211	943	4277	17595
	3	14	45	168	777	3522	14771
	4	12	37	142	642	2901	12669
random	1	19	78	318	1421	6250	23589
	2	11	42	166	718	3050	12255
	3	9	30	111	483	1999	8060
	4	8	24	87	361	1474	6043
music	1	18	78	320	1432	5854	23502
	2	11	42	166	723	3014	11964
	3	9	30	113	484	2034	8235
	4	7	23	83	363	1459	5985
ct	1	16	72	307	1396	5704	23653
	2	10	39	158	701	2959	12010
	3	8	29	108	472	1978	8004
	4	8	22	82	356	1454	6196

Table A.15: Performance test of the implementation of Tarjan's disjoint set algorithm. The backward version is used with a *mpart* of 4.

		Efficiency per processor					
image	#P	128	256	512	1024	2048	4096
empty	2	90.9%	93.0%	97.9%	85.8%	96.5%	99.6%
	3	74.1%	88.9%	98.2%	99.4%	96.6%	96.8%
	4	71.4%	87.0%	95.1%	100.1%	97.3%	96.1%
vertical	2	66.7%	77.2%	78.7%	75.8%	70.0%	70.7%
	3	48.5%	69.6%	76.8%	74.4%	69.0%	70.7%
	4	44.4%	65.7%	69.2%	74.2%	70.2%	70.1%
horizontal	2	94.4%	94.4%	98.2%	99.2%	97.2%	99.4%
	3	81.0%	90.7%	96.5%	99.5%	95.7%	98.8%
	4	70.8%	85.0%	95.5%	99.1%	98.3%	98.8%
comb	2	64.3%	66.7%	69.9%	68.9%	62.4%	60.9%
	3	42.9%	53.3%	58.5%	55.7%	50.5%	48.3%
	4	37.5%	48.6%	51.9%	50.6%	46.0%	42.3%
random	2	86.4%	92.9%	95.8%	99.0%	102.5%	96.2%
	3	70.4%	86.7%	95.5%	98.1%	104.2%	97.6%
	4	59.4%	81.2%	91.4%	98.4%	106.0%	97.6%
music	2	81.8%	92.9%	96.4%	99.0%	97.1%	98.2%
	3	66.7%	86.7%	94.4%	98.6%	95.9%	95.1%
	4	64.3%	84.8%	96.4%	98.6%	100.3%	98.2%
ct	2	80.0%	92.3%	97.2%	99.6%	96.4%	98.5%
	3	66.7%	82.8%	94.8%	98.6%	96.1%	98.5%
	4	50.0%	81.8%	93.6%	98.0%	98.1%	95.4%

Table A.16: The efficiency calculated from table A.15

		Timings (ms)					
image	#P	128	256	512	1024	2048	4096
empty	1	19	81	329	1442	5971	23892
	2	11	43	167	741	3071	12094
	3	9	30	112	490	2040	8249
	4	7	23	85	367	1481	6064
vertical	1	16	69	285	1254	5152	20728
	2	13	48	181	817	3721	14948
	3	13	36	128	568	2532	10094
	4	11	31	96	420	1824	7602
horizontal	1	16	68	275	1272	5220	21715
	2	9	36	140	647	2674	10716
	3	7	25	95	432	1791	7150
	4	6	20	71	322	1321	5281
comb	1	18	73	297	1304	5296	21612
	2	15	54	211	939	4269	17670
	3	17	43	158	707	3267	13679
	4	17	36	128	550	2531	10630
random	1	18	79	319	1434	5837	23785
	2	12	44	164	729	3043	12031
	3	11	31	112	480	1955	7969
	4	8	24	84	363	1483	6243
music	1	19	77	322	1438	5901	23870
	2	13	42	165	716	2995	11926
	3	10	30	112	482	2016	8073
	4	7	23	84	360	1464	5991
ct	1	17	71	304	1391	5728	23773
	2	10	39	158	699	2942	11904
	3	12	32	109	475	2010	8220
	4	7	22	82	355	1487	6049

Table A.17: Performance test of the implementation of Tarjan's disjoint set algorithm. The backward version is used with a *mpart* of 16.

		Efficiency per processor					
image	#P	128	256	512	1024	2048	4096
empty	2	86.4%	94.2%	98.5%	97.3%	97.2%	98.8%
	3	70.4%	90.0%	97.9%	98.1%	97.6%	96.5%
	4	67.9%	88.0%	96.8%	98.2%	100.8%	98.5%
vertical	2	61.5%	71.9%	78.7%	76.7%	69.2%	69.3%
	3	41.0%	63.9%	74.2%	73.6%	67.8%	68.4%
	4	36.4%	55.6%	74.2%	74.6%	70.6%	68.2%
horizontal	2	88.9%	94.4%	98.2%	98.3%	97.6%	101.3%
	3	76.2%	90.7%	96.5%	98.1%	97.2%	101.2%
	4	66.7%	85.0%	96.8%	98.8%	98.8%	102.8%
comb	2	60.0%	67.6%	70.4%	69.4%	62.0%	61.2%
	3	35.3%	56.6%	62.7%	61.5%	54.0%	52.7%
	4	26.5%	50.7%	58.0%	59.3%	52.3%	50.8%
random	2	75.0%	89.8%	97.3%	98.4%	95.9%	98.8%
	3	54.5%	84.9%	94.9%	99.6%	99.5%	99.5%
	4	56.2%	82.3%	94.9%	98.8%	98.4%	95.2%
music	2	73.1%	91.7%	97.6%	100.4%	98.5%	100.1%
	3	63.3%	85.6%	95.8%	99.4%	97.6%	98.6%
	4	67.9%	83.7%	95.8%	99.9%	100.8%	99.6%
ct	2	85.0%	91.0%	96.2%	99.5%	97.3%	99.9%
	3	47.2%	74.0%	93.0%	97.6%	95.0%	96.4%
	4	60.7%	80.7%	92.7%	98.0%	96.3%	98.3%

Table A.18: The efficiency calculated from table A.17

		Timings (ms)					
image	#P	128	256	512	1024	2048	4096
empty	1	19	80	330	1432	5920	23736
	2	11	43	163	739	3019	12024
	3	10	30	113	497	2062	8107
	4	7	23	85	369	1482	6054
vertical	1	18	70	286	1254	5144	20737
	2	19	51	189	814	3689	14928
	3	24	45	141	655	2992	12272
	4	25	41	111	478	2198	9175
horizontal	1	16	68	275	1310	5140	20985
	2	10	36	141	647	2602	10711
	3	7	26	94	447	1784	7075
	4	6	20	72	377	1323	5350
comb	1	18	73	296	1292	5295	21512
	2	21	58	216	955	4305	17614
	3	26	56	170	736	3321	13718
	4	33	52	144	595	2721	11470
random	1	19	78	319	1432	5838	23752
	2	11	43	166	715	3010	11978
	3	12	33	112	489	2009	8099
	4	9	28	85	363	1482	6404
music	1	18	79	319	1423	5853	23792
	2	11	42	166	717	3016	12465
	3	11	33	118	484	2011	8022
	4	8	25	83	363	1472	5987
ct	1	17	71	306	1382	5750	23739
	2	11	39	156	710	2943	11956
	3	11	31	111	479	2024	8118
	4	9	24	81	354	1464	5965

Table A.19: Performance test of the implementation of Tarjan's disjoint set algorithm. The backward version is used with a *mpart* of 64.

		Efficiency per processor					
image	#P	128	256	512	1024	2048	4096
empty	2	86.4%	93.0%	101.2%	96.9%	98.0%	98.7%
	3	63.3%	88.9%	97.3%	96.0%	95.7%	97.6%
	4	67.9%	87.0%	97.1%	97.0%	99.9%	98.0%
vertical	2	47.4%	68.6%	75.7%	77.0%	69.7%	69.5%
	3	25.0%	51.9%	67.6%	63.8%	57.3%	56.3%
	4	18.0%	42.7%	64.4%	65.6%	58.5%	56.5%
horizontal	2	80.0%	94.4%	97.5%	101.2%	98.8%	98.0%
	3	76.2%	87.2%	97.5%	97.7%	96.0%	98.9%
	4	66.7%	85.0%	95.5%	86.9%	97.1%	98.1%
comb	2	42.9%	62.9%	68.5%	67.6%	61.5%	61.1%
	3	23.1%	43.5%	58.0%	58.5%	53.1%	52.3%
	4	13.6%	35.1%	51.4%	54.3%	48.6%	46.9%
random	2	86.4%	90.7%	96.1%	100.1%	97.0%	99.1%
	3	52.8%	78.8%	94.9%	97.6%	96.9%	97.8%
	4	52.8%	69.6%	93.8%	98.6%	98.5%	92.7%
music	2	81.8%	94.0%	96.1%	99.2%	97.0%	95.4%
	3	54.5%	79.8%	90.1%	98.0%	97.0%	98.9%
	4	56.2%	79.0%	96.1%	98.0%	99.4%	99.3%
ct	2	77.3%	91.0%	98.1%	97.3%	97.7%	99.3%
	3	51.5%	76.3%	91.9%	96.2%	94.7%	97.5%
	4	47.2%	74.0%	94.4%	97.6%	98.2%	99.5%

Table A.20: The efficiency calculated from table A.19

		Timings (ms)					
image	#P	128	256	512	1024	2048	4096
empty	1	20	80	326	1444	5913	23673
	2	11	43	167	703	3042	12188
	3	9	30	113	487	2026	8086
	4	7	23	85	362	1491	6291
vertical	1	18	70	285	1252	5251	20861
	2	30	83	211	844	3733	14953
	3	33	100	192	720	3157	13003
	4	49	111	173	555	2321	9464
horizontal	1	16	67	274	1285	5304	20993
	2	9	37	142	647	2648	10655
	3	8	26	95	430	1791	7157
	4	6	20	72	324	1299	5368
comb	1	17	72	297	1301	5361	21624
	2	35	82	235	970	4286	17720
	3	39	108	243	811	3552	14646
	4	89	130	220	683	2873	12282
random	1	18	78	324	1417	5823	23783
	2	12	44	165	720	2995	12015
	3	11	33	117	483	2022	8067
	4	11	28	86	365	1490	6026
music	1	19	78	319	1435	5868	23493
	2	12	43	164	723	3023	12042
	3	12	36	114	484	2036	7933
	4	8	27	85	360	1480	6051
ct	1	17	71	308	1384	5736	23508
	2	12	44	157	699	2978	11951
	3	14	42	118	479	1993	7989
	4	12	27	85	361	1480	6076

Table A.21: Performance test of the implementation of Tarjan's disjoint set algorithm. The backward version is used with a *mpart* of 256.

		Efficiency per processor					
image	#P	128	256	512	1024	2048	4096
empty	2	90.9%	93.0%	97.6%	102.7%	97.2%	97.1%
	3	74.1%	88.9%	96.2%	98.8%	97.3%	97.6%
	4	71.4%	87.0%	95.9%	99.7%	99.1%	94.1%
vertical	2	30.0%	42.2%	67.5%	74.2%	70.3%	69.8%
	3	18.2%	23.3%	49.5%	58.0%	55.4%	53.5%
	4	9.2%	15.8%	41.2%	56.4%	56.6%	55.1%
horizontal	2	88.9%	90.5%	96.5%	99.3%	100.2%	98.5%
	3	66.7%	85.9%	96.1%	99.6%	98.7%	97.8%
	4	66.7%	83.8%	95.1%	99.2%	102.1%	97.8%
comb	2	24.3%	43.9%	63.2%	67.1%	62.5%	61.0%
	3	14.5%	22.2%	40.7%	53.5%	50.3%	49.2%
	4	4.8%	13.8%	33.8%	47.6%	46.6%	44.0%
random	2	75.0%	88.6%	98.2%	98.4%	97.2%	99.0%
	3	54.5%	78.8%	92.3%	97.8%	96.0%	98.3%
	4	40.9%	69.6%	94.2%	97.1%	97.7%	98.7%
music	2	79.2%	90.7%	97.3%	99.2%	97.1%	97.5%
	3	52.8%	72.2%	93.3%	98.8%	96.1%	98.7%
	4	59.4%	72.2%	93.8%	99.7%	99.1%	97.1%
ct	2	70.8%	80.7%	98.1%	99.0%	96.3%	98.4%
	3	40.5%	56.3%	87.0%	96.3%	95.9%	98.1%
	4	35.4%	65.7%	90.6%	95.8%	96.9%	96.7%

Table A.22: The efficiency calculated from table A.21

		Timings (ms)					
image	#P	128	256	512	1024	2048	4096
empty	1	19	80	330	1450	6013	24010
	2	12	44	170	730	3080	12035
	3	9	31	113	488	2042	8165
	4	7	24	86	367	1681	6640
vertical	1	17	78	286	1318	5254	21012
	2	29	79	258	958	3984	15630
	3	43	101	276	913	3681	14075
	4	47	105	283	799	2862	11283
horizontal	1	16	67	272	1246	5104	20751
	2	10	36	153	656	2663	10553
	3	8	26	96	436	1786	7074
	4	6	21	73	316	1333	5219
comb	1	18	75	306	1340	5356	21988
	2	32	82	280	1053	4570	18686
	3	40	99	294	1005	4081	15980
	4	51	124	315	883	3508	13682
random	1	19	78	320	1417	5917	23862
	2	12	43	168	723	3047	12241
	3	11	33	123	486	2055	8251
	4	10	29	90	369	1507	6072
music	1	19	80	331	1449	5962	23825
	2	13	43	168	725	3029	12070
	3	12	34	117	487	2058	8100
	4	8	29	88	369	1500	6063
ct	1	17	73	305	1403	5861	23830
	2	12	42	163	724	3023	12001
	3	15	42	132	513	2035	8161
	4	11	32	91	443	1456	6041

Table A.23: Performance test of the implementation of Tarjan's disjoint set algorithm. The backward version is used with a *mpart* of ∞.

		Efficiency per processor					
image	#P	128	256	512	1024	2048	4096
empty	2	79.2%	90.9%	97.1%	99.3%	97.6%	99.8%
	3	70.4%	86.0%	97.3%	99.0%	98.2%	98.0%
	4	67.9%	83.3%	95.9%	98.8%	89.4%	90.4%
vertical	2	29.3%	49.4%	55.4%	68.8%	65.9%	67.2%
	3	13.2%	25.7%	34.5%	48.1%	47.6%	49.8%
	4	9.0%	18.6%	25.3%	41.2%	45.9%	46.6%
horizontal	2	80.0%	93.1%	88.9%	95.0%	95.8%	98.3%
	3	66.7%	85.9%	94.4%	95.3%	95.3%	97.8%
	4	66.7%	79.8%	93.2%	98.6%	95.7%	99.4%
comb	2	28.1%	45.7%	54.6%	63.6%	58.6%	58.8%
	3	15.0%	25.3%	34.7%	44.4%	43.7%	45.9%
	4	8.8%	15.1%	24.3%	37.9%	38.2%	40.2%
random	2	79.2%	90.7%	95.2%	98.0%	97.1%	97.5%
	3	57.6%	78.8%	86.7%	97.2%	96.0%	96.4%
	4	47.5%	67.2%	88.9%	96.0%	98.2%	98.2%
music	2	73.1%	93.0%	98.5%	99.9%	98.4%	98.7%
	3	52.8%	78.4%	94.3%	99.2%	96.6%	98.0%
	4	59.4%	69.0%	94.0%	98.2%	99.4%	98.2%
ct	2	70.8%	86.9%	93.6%	96.9%	96.9%	99.3%
	3	37.8%	57.9%	77.0%	91.2%	96.0%	97.3%
	4	38.6%	57.0%	83.8%	79.2%	100.6%	98.6%

Table A.24: The efficiency calculated from table A.23