Distributed Determination of Connected Components

M. Sinnema

Supervisors:
W.H. Hesselink and A. Meijster
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

1 Introduction .................................................. 3
   1.1 Images .................................................... 3
   1.2 Segmentation ............................................. 4
   1.3 Parallel Computing ...................................... 6

2 A sequential Union-Find algorithm for determination of components .............. 8
   2.1 Problem description ....................................... 8
   2.2 Tarjan's disjoint set algorithm ......................... 8
   2.3 Design of a sequential algorithm ....................... 11
   2.4 Harvest .................................................... 14

3 A distributed Union-Find algorithm for determination of components ................ 16
   3.1 Introduction ............................................... 16
   3.2 Sequential processing .................................... 16
   3.3 Parallel processing ...................................... 17
   3.4 Parallel Harvest ......................................... 21

4 Implementation .................................................. 23
   4.1 Introduction ............................................... 23
   4.2 Images ..................................................... 23
   4.3 Communication with the MPI interface .................. 23
   4.4 Distribution of the image ................................ 26
   4.5 Optimization .............................................. 26
   4.6 Translation to C .......................................... 32

5 Performance ..................................................... 36
   5.1 Contents of the image .................................... 36
   5.2 Expected performance .................................... 38
   5.3 Architecture .............................................. 39
   5.4 Timing ..................................................... 40
   5.5 Method ..................................................... 40
   5.6 Results .................................................... 40
   5.7 Conclusions .............................................. 41

6 Additional work .................................................. 45
   6.1 The area of connected components ...................... 45
   6.2 Distributed calculation of the distance transform ..... 46
   6.3 Merging of connected components ....................... 51

A Test results ................................................... 58
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.

$$\text{image} : D \rightarrow E.$$ 

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 \text{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 $D \subseteq \mathbb{N}^k$.

Figure 1.1: Some examples of images of different dimensions.
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 \( \text{image}[x] \) is equal to \( \text{image}[y] \), 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 \( S \subseteq \mathbb{Z}^k \). The set of neighbours \( Nb(p) \) of pixel \( p \) is defined as
1.2 Segmentation

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

- $S_a = \{(-1),(1)\}$
- $S_b = \{(-2),(2)\}$
- $S_c = S_a \cup S_b$
- $S_d = \{(u,v) \in \mathbb{Z}^2 | |u| + |v| = 1\}$
- $S_e = \{(u,v) \in \mathbb{Z}^2 | 1 \leq |u| + |v| \leq 2 \land |u| \leq 1 \land |v| \leq 1\}$, and
- $S_f = \{(u,v) \in \mathbb{Z}^2 | 1 \leq |u| + |v| \leq 2\}$

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.

$$E = \{(x,y) \in D \times D | x \in Nb(y) \land image[x] = image[y]\}.$$

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

\[
\text{send } amsg \text{ to } y;
\]

\[
\text{receive } amsg,
\]

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

\[
\text{send } amsg(s,t) \text{ to } y,
\]

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,

\[
\text{receive } amsg(s,t)
\]

is used.

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

\[
\begin{align*}
\text{in } mtx(a,b) & \rightarrow \\
& A; \\
& [] mty \rightarrow
\end{align*}
\]
This means that three different types of messages can arrive, namely a $\text{mtx}$, a $\text{mty}$, or a $\text{mtz}$ 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 $\text{mtx}$ message, which has two arguments, code fragment $A$ is executed. The arrival of a $\text{mty}$ message, with no arguments, results in executing fragment $B$. Finally, a $\text{mtz}$ 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

$$
P_a: \quad \text{send } mX \text{ to } P_b; \\
\quad \text{send } mY(1) \text{ to } P_b; \\
\quad \text{receive } mZ; \\
\quad \text{receive } mZ,
$$

and for process $P_b$,

$$
P_b: \quad \text{for } i := 1 \text{ to } 2 \text{ do} \\
\quad \text{in } mX \rightarrow \\
\quad \quad \text{print } 'A'; \\
\quad \quad \text{send } mZ \text{ to } P_a; \\
\quad \text{mY}(j) \rightarrow \\
\quad \quad \text{print } j; \\
\quad \quad \text{send } mZ \text{ to } P_a; \\
\quad \text{ni;} \\
\text{od.}
$$

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, \( f : D \rightarrow \mathbb{N} \) is a function from a finite domain \( D \subseteq \mathbb{N}^k \). Let \( S \subseteq \mathbb{Z}^k \) 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 \( \text{Nb}(p) \), i.e.

\[
\text{Nb}(p) = D \cap (p + S).
\]

A finite sequence of points \([x_1, \ldots, x_n]\) is called an iso-level path from \( p \) to \( q \) iff \( p = x_1, q = x_n \) and \( x_{i+1} \in \text{Nb}(x_i) \) 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 \( \text{Conn}(X) \). A connected set \( X \) is called a connected component if the set is maximal, which is denoted by \( \text{CC}(X) \), i.e.

\[
\text{CC}(X) \equiv \forall(Y : \text{Conn}(Y) \land X \subseteq Y : X = Y).
\]

Clearly, the connected components of an image partition the domain \( D \). A function \( \text{lab} : D \rightarrow \mathbb{N} \), 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

\[
\text{lab}(p) = \text{lab}(q) \quad \equiv \quad q \in X.
\]

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.
2.2 Tarjan's disjoint set algorithm

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.

- \textit{MakeSet}(x), which creates a new singleton set \{x\}. This operation assumes
  that x is not a member of any other set.
- \textit{Find}(x), which returns the canonical element of the set containing x.
- \textit{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 \textit{Find} operation reduces to finding the root node of a tree, and is therefore
called \textit{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 parent[x] is the parent of x in the tree x is contained in. When x is a
canonical element, we have parent[x] = x.

Time complexity

Obviously, the operation \textit{MakeSet}(x) can be performed in constant time, but the op-
erations \textit{FindRoot}(x), and \textit{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 \textit{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 \textit{path compression}. Every time the operation \textit{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 \textit{FindRoot}(x), a second operation \textit{FindRoot}(y), with y on the root-path of
x, takes constant time.

The second technique is called \textit{union by rank}. This technique is used in the operation
\textit{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 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.
2.2 Tarjan's disjoint set algorithm

Tarjan shows that in an intermixed sequence of \( m \) operations, of which there are \( n \) \textit{MakeSet} operations (and hence at most \( n - 1 \) \textit{Union} operations) and \( f \) \textit{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.

\begin{align*}
\text{\textit{MakeSet}(x):} & \quad \text{parent}[x] := x; \\
& \quad \text{rank}[x] := 0;
\end{align*}

\begin{align*}
\text{\textit{FindRoot}(x):} & \quad \text{if } x \neq \text{parent}[x] \text{ then} \\
& \quad \text{parent}[x] := \text{\textit{FindRoot}(parent[x])}; \\
& \quad \text{fi}; \\
& \quad \text{return parent}[x];
\end{align*}

\begin{align*}
\text{\textit{Link}(x,y):} & \quad \text{parent}[x] := y;
\end{align*}

\begin{align*}
\text{\textit{Union}(x,y):} & \quad p := \text{\textit{FindRoot}(x)}; \\
& \quad q := \text{\textit{FindRoot}(y)}; \\
& \quad \text{if rank}[p] > \text{rank}[q] \text{ then} \\
& \quad \text{Link}(q,p); \\
& \quad \text{else if rank}[p] < \text{rank}[q] \text{ then} \\
& \quad \text{Link}(p,q); \\
& \quad \text{else} \\
& \quad \text{Link}(p,q); \\
& \quad \text{rank}[q] := \text{rank}[q] + 1; \\
& \quad \text{fi};
\end{align*}

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. \textit{MakeSet}(0); \ldots; \textit{MakeSet}(5): all ranks are set to 0.
2.3 Design of a sequential algorithm

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

\[ \text{par} : \text{array} \ D \text{ of } D. \]

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 \( 3 \times 3 \) 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 \( \text{par} \) to \( x \), i.e.

\[ \text{root}(x) = \text{if } \text{par}[x] = x \text{ then } x \text{ else } \text{root}(\text{par}[x]) \]
2.3 Design of a sequential algorithm

Vertices \( x \) and \( y \) belong to the same component if \( \text{root}(x) = \text{root}(y) \).

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 \( \text{Edges} \) to consist of the pairs \((x, y) \in E \) with \( x > y \), i.e.

\[
\text{Edges} = \{(x, y) \in D \times D | (x, y) \in E \land x > y\}.
\]

To make reasoning about already processed edges easier, we partition the set \( \text{Edges} \) 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

\[ J_2 \quad E_p \cup E_n = \text{Edges} \land E_p \cap E_n = \emptyset. \]

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

\[ J_3 \quad \forall (x, y) \in D :: \pi_{E_p}(x, y) \equiv \text{root}(x) = \text{root}(y). \]

The invariants are initialized by the following fragment

\[ \text{Tarseqinit:} \quad \text{for all } x \in D \text{ do} \]
\[ \quad \text{par}[x] := x; \]
\[ \quad E_n := \text{Edges}; \]
\[ \quad E_p := \emptyset; \]

In the main fragment \( \text{Tarseqmain} \), repeatedly an edge in \( E_n \) is processed, and is moved from \( E_n \) to \( E_p \). This preserves invariant \( J_2 \). When the set \( E_n \) is empty, all edges have been processed.

\[ \text{Tarseqmain:} \quad \text{while } E_n \neq \emptyset \text{ do} \]
\[ \quad \text{choose } (x, y) \in E_n; \]
\[ \quad \text{Extend}(x, y); \]
\[ \quad E_p := E_p \cup \{(x, y)\}; \]
\[ \quad E_n := E_n \setminus \{(x, y)\}; \]
\[ \quad \text{od}; \]

The function \( \text{Extend} \) should update \( \text{par} \) if necessary, in order to maintain invariant \( J_3 \). 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

\[ \text{Tarseq:} \quad \text{for all } y \in D \text{ in decreasing order do} \]
\[ \quad \text{for all } x \in D \text{ such that } (x, y) \in E_n \]
\[ \quad \text{Extend}(x, y); \]
\[ \quad E_p := E_p \cup \{(x, y)\}; \]
\[ E_n := E_n \setminus \{(x,y)\}; \]
\[ \text{od}; \]
\[ \text{od}. \]

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 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 par\([\text{root}(x)]\) to \(y\). In order to find the root of vertex \(x\) we introduce the following fragment

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

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

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

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

\[ \text{Extend}(x,y): \quad \text{do} \]
\[ p := \text{par}[x]; \]
\[ \text{par}[x] := y; \]
\[ x := p; \]
\[ \text{while } \text{par}[x] \neq y; \]

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.
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 \( x \in D \)

\[
\text{lab}[x] = \text{root}(x).
\]

In fact this labelling is obtained by simply performing \( \text{lab}[x] := \text{FindRoot}(x) \) for all \( x \in D \).

However, we can do this more efficiently by using invariant \( J1 \) and a raster scan algorithm. This leads to the following fragment:

\[
\text{Harvseq:} \quad \text{for all } x \in D \text{ do in increasing order}
\]

\[
\begin{align*}
&\text{if } \text{par}[x] = x \text{ then} \\
&\quad \text{lab}[x] := x; \\
&\text{else} \\
&\quad \text{lab}[x] := \text{lab}[\text{par}[x]]; \\
&\quad \text{fi}
\end{align*}
\]

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

\[
\text{Labelseq : } \text{Tarseqinit};
\text{Tarseq; }
\text{Harvseq; }
\]
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 $par[x]$ 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 $Tar_{seq}$ to the set $InEdges(k)$. The invariant $J2$ has to be redefined for this parallel situation

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

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

From $J3$ and the partition on $Edges$ we can conclude that after $Tarseq$

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

### 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 $\text{Union}(x, y)$ 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

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

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

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

After $Tarseq$, $J4$ is easily initialized by the statement

$\text{Tarparinit:}$

$$F := \bigcup (k \in Processes :: E_p(k)).$$

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.

$\text{Tarparmain:}$

$$\text{while } OutEdges(k) \neq \emptyset \text{ do}
\quad \text{choose } (x, y) \in OutEdges(k);
\quad \text{Extend}(x, y);
\quad F := F \cup \{(x, y)\};
\quad OutEdges(k) := OutEdges(k) \setminus \{(x, y)\};$$
$$\text{od;}$$

In the first two statements of $\text{Union}(x, y)$ a $\text{FindRoot}$ is done for both $x$ and $y$. In $\text{Extend}(x, y)$, the roots of $x$ and $y$ are searched for simultaneously by the fragment $\text{Search}$ below.

The invariant

$$J5 \quad y \leq x$$
3.3 Parallel processing

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.

\[
\text{Search}(x, y): \quad \begin{cases} \text{if } \text{par}[x] \neq x \land x \neq y \text{ then} \\ x := \text{par}[x]; \\ \quad \begin{cases} \text{if } x < y \text{ then } x, y := y, x & \text{fi}; \\ \quad \text{Search}(x, y); \end{cases} \\ \text{fi}; \end{cases}
\]

The \( \text{Extend}(x, y) \) 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 \( \text{par}[x] = y \). Without considering the fact that the array \( \text{par} \) is distributed, we can define

\[
\text{Extend}(x, y): \quad \begin{cases} \text{Search}(x, y); \\ \text{if } x \neq y \text{ then} \\ \quad \text{par}[x] = y; \\ \text{fi}, \end{cases}
\]

which maintains all invariants. Note that in Extend the trees are only linked when \( x \neq y \), and \( \text{par}[y] \) is not always equal to \( y \).

In the following fragments, algorithms are indexed by process numbers, e.g. Search\(_k\). This \( k \in \text{Processes} \) is the process that executes the fragment, and can be used in the body.

Because in the fragment Search the value of the array \( \text{par} \) 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.

\[
\text{Search}_k(x, y): \quad \begin{cases} \text{if } \text{par}[x] \neq x \land x \neq y \text{ then} \\ x := \text{par}[x]; \\ \quad \begin{cases} \text{if } x < y \text{ then } x, y := y, x & \text{fi}; \\ \quad \text{Search}_{\text{owner}(x)}(x, y); \end{cases} \\ \text{fi}. \end{cases}
\]

In Extend\((x, y)\), after Search\((x, y)\), \( \text{par}[x] \) 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

\[
\text{Extend}_k(x, y): \quad \begin{cases} \text{if } \text{par}[x] \neq x \land x \neq y \text{ then} \\ x := \text{par}[x]; \\ \quad \begin{cases} \text{if } x < y \text{ then } x, y := y, x & \text{fi}; \\ \quad \text{Extend}_{\text{owner}(x)}(x, y); \end{cases} \\ \text{else if } x \neq y \text{ then} \\ \quad \text{par}[x] = y; \end{cases} \end{cases}
\]

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

Implementation with messages

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

\[
\text{Extend}_k(x, y): \quad \begin{cases} 
\text{par}[x] \neq x \land x \neq y & \text{then} \\
\quad x := \text{par}[x]; \\
\quad \text{if } x < y \text{ then } x,y := y,x \text{ fi;}
\end{cases}
\]

\[
\text{send edge}(x, y) \text{ to } \text{owner}(x);
\]

\[
\begin{cases} 
\text{else if } x \neq y & \text{then} \\
\quad \text{par}[x] := y; \\
\end{cases}
\]

All processes repeatedly receive edge messages and execute \(\text{Extend}\) for each edge.

\[
\text{Tarparmain}_k: \quad \text{while \; TRUE \; do} \\
\quad \text{receive edge}(x, y); \\
\quad \text{Extend}_k(x, y); \\
\quad \text{od.}
\]

All processes \(k\) initialize the parallel processing with the following fragment

\[
\text{Tarparinit}_k: \quad F := \bigcup(k \in \text{Processes} :: E_p(k)); \\
\text{for all } (x, y) \in \text{OutEdges}(k) \text{ do} \\
\quad \text{send edge}(x, y) \text{ to } k; \\
\quad \text{od},
\]

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

\[
\text{Tarpar:} \quad \bigparallel_k \text{Tarpar}_k,
\]

which is the parallel composition of all processes \(k\) executing \(\text{Tarpar}_k\). Here, \(\text{Tarpar}_k\) is defined as follows

\[
\text{Tarpar}_k: \quad \text{Tarseq}(\text{InEdges}(k)); \\
\text{Tarparinit}_k(\text{OutEdges}(k)); \\
\text{Tarparmain}_k.
\]

Termination

The fragment \(\text{Tarparmain}_k\) 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 \(F = \text{Edges}\). \(F\) 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 \(\text{Tarparinit}_k\). An edge message is denoted by \(\text{edge}(x, y, \text{origin})\).
3.3 Parallel processing

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

$$\text{Tarparinit}_k: \quad F := \bigcup \{ k \in \text{Processes} : E_p(k) \};$$
$$ctok := 0;$$
$$\text{for all } (x,y) \in \text{OutEdges}(k) \text{ do}$$
$$\quad \text{send } \text{edge}(x,y,k) \text{ to } k;$$
$$\quad ctok := ctok + 1;$$
$$\text{od;}$$
$$\text{if } ctok = 0 \text{ then}$$
$$\quad \text{send } \text{gcdown} \text{ to } \text{adm};$$
$$\text{fi.}$$

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 $\text{adm} \in \text{Processes}$ 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 $\text{adm}$ when its $ctok$ value is zero. At the arrival of a GcDown message $\text{adm}$ 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

$$\text{continue } \equiv \ F \neq \text{Edges.}$$

In $\text{Extend}_k(x,y,\text{origin})$ 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.

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

The complete $\text{Tarparmain}_k$ is given below.

$$\text{Tarparmain}_k \quad \text{while } \text{continue} \text{ do}$$
$$\text{in } \text{edge}(x,y,\text{origin}) \rightarrow$$
$$\quad \text{Extend}_k(x,y,\text{origin});$$
$$\text{down } \rightarrow$$
$$\quad \text{ctok} := \text{ctok} - 1;$$
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.

\[\text{OutPar}(k) = \{x \in D \mid \text{owner}(x) = k \land \text{owner}(\text{par}[x]) \neq k\}.\]

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

\[
\text{Harvparinit}_k: \quad \text{for all } x \in \{x \in D \mid \text{owner}(x) = k\} \text{ do } \text{lab}[x] := \bot \quad \text{od};
\]
\[
ctok := 0;
\]
\[
\text{for all } x \in \text{OutPar}(k) \text{ do}
\]
\[
\quad \text{send } \text{query}(\text{par}[x],x) \text{ to } \text{owner}(\text{par}[x]);
\]
\[
ctok := ctok + 1;
\]
\[
\text{od};
\]
\[
\text{if } ctok = 0 \text{ then}
\]
\[
\quad \text{send } \text{gcdown} \text{ to } \text{adm};
\]
\[
\text{fi.}
\]

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
3.4 Parallel Harvest

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.

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

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

Harvparlocal\(_k\): for all \(x \in \{x \in D \mid \text{owner}(x) = k\} \) do in increasing order
    if lab[n] = \bot then
        if par[n] = n then lab[n] := n
        else lab[n] := lab[par[n]] fi
    fi od.

The complete parallel harvest algorithm is

Harvpar\(_k\): Harvparinit\(_k\);
Harvparmain\(_k\);
Harvparlocal\(_k\).
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 \( \text{im} \) of size \( \text{npixels} \) and of type integer, i.e.

\[
\text{int } \ast \text{im} = \text{malloc} (\text{sizeof} (\text{int}) \ast \text{npixels}),
\]

where \( \text{npixels} \) is the total number of pixels in the image. In a 2D image of size \( m \times n \), \( \text{npixels} = mn \). 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, \( \text{npixels} = 3 \times 4 = 12 \).

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, \text{MPI}._\text{COMM}._\text{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 \( \text{Processes} \) 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.
4.3 Communication with the MPI interface

Distribution of memory

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

\[ \text{MPI\_Scatter}(\text{sendbuf}, \text{sendcount}, \text{sendtype}, \text{recvbuf}, \text{recvcount}, \text{recvtype}, \text{root}, \text{communicator}) \]

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

\[ \text{MPI\_Gather}(\text{sendbuf}, \text{sendcount}, \text{sendtype}, \text{recvbuf}, \text{recvcount}, \text{recvtype}, \text{root}, \text{communicator}) \]

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

\[ \text{MPI\_Isend}(\text{outmessage}, \text{size}, \text{type}, \text{dest}, \text{tag}, \text{communicator}, \text{request}) \]
4.3 Communication with the MPI Interface

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

\[
\text{MPI\_Recv(outmessage, size, type, source, tag, communicator, &status),}
\]

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
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 \( \text{owner}(x) = k \). The MPI.Scatter procedure is used for the distribution of the array \( \text{im} \). 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 \( x, y \in D \)

\[ T1 \quad x \leq y \Rightarrow \text{owner}(x) \leq \text{owner}(y). \]

From \( T1 \) and the invariant \( \text{par}[x] \leq x \) we conclude that for all \( k \in \text{Processes} \)

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

In figure 4.3 an example of this distribution is shown. Note that the concatenation of all parts of array \( \text{im} \) is equal to \( \text{im} \) 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 \( \text{im} \) belonging to \( k \), can be done without communication.

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

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

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

Local FindRoot

In both Tarparmain and Harvparmain there are many messages sent from a process to itself. In both Tarparmain and Harvparmain 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 par[x] ≤ x. The pseudo code fragment for FindLocalRoot is

\[
\text{FindLocalRoot}_k(x):
\]

\[
r := x;
\]

\[
\text{while } \text{par}[r] \neq r \land \text{owner} (\text{par}[r]) = k \text{ do}
\]

\[
r := \text{par}[r];
\]

\[
\text{od;}
\]

\[
\text{return } r.
\]

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 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 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.
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 Harvparmain 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 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
4.5 Optimization

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 messages.

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 \( p \in \text{Processes} \). 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 \( \text{MAXSIZE} \) the messagelist is sent to process \( p \). We introduce the new keyword \texttt{add}, which adds a single message to a messagelist, i.e.

\[
\text{add } msg \text{ to } \text{MsgList}(p):
\]

\[
\text{MsgList}(p) := \text{MsgList}(p) \cup msg;
\]

if \( \#\text{MsgList}(p) = \text{MAXSIZE} \) then
4.5 Optimization

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.

**TarparInit**:
\[
ctok := 0;
\]
\[
\text{for all } (x, y) \in \text{OutEdges}(k) \text{ do}
ctok := ctok + 1;
\]
\[
\text{add edge}(x, y, k) \text{ to EdgeList}(k);
\]
\[
\text{od};
\]
\[
\text{if } ctok = 0 \text{ then}
\]
\[
\text{send } gcdown \text{ to adm}
\]
\[
\text{else}
\]
\[
\text{send } \text{EdgeList}(k) \text{ to } k
\]
\[
\text{fi}.
\]

**TarparMain**:
\[
\text{while continue do}
\]
\[
\text{for all } p \in \text{Processes do}
\]
\[
\text{if not empty(EdgeList}(p)) \text{ then}
\]
\[
\text{send } \text{EdgeList}(p) \text{ to } p;
\]
\[
\text{fi;}
\]
\[
\text{if not empty(DownList}(p)) \text{ then}
\]

send DownList(p) to p;
fi;

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;

Harvparinitk:
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.

Harvparmank:
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

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

outmessage[0] = a;
outmessage[1] = b;
outmessage[2] = c;
```
4.6 Translation to C

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

Receiving a single message

The corresponding pseudo code call to

\[
\text{receive amsg}(a,b,c) \text{ from } x
\]

is translated to

```c
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

\[
\text{inxmsg \rightarrow \ \\
fragA; \ \\
ymsg(a) \rightarrow \ \\
fragB(a) \ \\
zmsg(b,c) \rightarrow \ \\
fragC(b,c); \ \\
ni,}
\]

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

```c
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;
}
```
4.6 Translation to C

Sending of grouped messages

We define a new C structure for grouped messages

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

One messagelist is initialized as follows.

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

The add procedure

```c
addmsg(x,y)toMsgList:
    MsgList := MsgList U msg(x,y);
    if #MsgList = MAXSIZE then
        send MsgList to procR;
    fi.
```

is translated to

```c
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.

```c
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

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

is translated to the C code fragment

```c
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 x 256 2D image and compare it to the time it takes to analyse a 512 x 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.
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 $32 \times 32$ for visibility reasons. The other images have the original size $256 \times 256$. 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.
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

\[
\text{speedup} = \frac{t_1}{t_2} = \frac{nprocs_2}{nprocs_1},
\]

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

\[
\text{speedup} = \frac{t_1}{t_2} = \left(\frac{n_1}{n_2}\right)^2,
\]

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 [WHHO1], a sequential solution is given, which we have parallelized 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 `ParTree` 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 ∈ \{empty, horizontal, vertical, comb, random, music, ct\} do
  for all size ∈ \{128, 256, 512, 1024, 2048, 4096\} do
    for all nprocs ∈ \{1, 2, 3, 4\} do
      for all mpart ∈ \{1, 4, 16, 64, 256, ∞\} do
        for all method ∈ \{forward, backward\} do
          time segmentation(image, size, nprocs, mpart, method);
        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.

\[
Efficiency = \frac{t_1}{t_n} \times 100\%,
\]

where \( t_n \) is the time spent using \( n \) processes.
5.7 Conclusions

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

<table>
<thead>
<tr>
<th>method</th>
<th>processing method</th>
<th>mpart</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>forward</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>forward</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>forward</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>forward</td>
<td>64</td>
</tr>
<tr>
<td>4</td>
<td>forward</td>
<td>256</td>
</tr>
<tr>
<td>5</td>
<td>forward</td>
<td>∞</td>
</tr>
<tr>
<td>6</td>
<td>backward</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>backward</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>backward</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>backward</td>
<td>64</td>
</tr>
<tr>
<td>10</td>
<td>backward</td>
<td>256</td>
</tr>
<tr>
<td>11</td>
<td>backward</td>
<td>∞</td>
</tr>
</tbody>
</table>

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.
5.7 Conclusions

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 efficiency graphs. 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 = ∞.
Figure 5.5: Efficiency graph of the test with 512 x 512 images.

Figure 5.6: Efficiency graph of the test with 1024 x 1024 images.
5.7 Conclusions

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

Figure 5.8: Efficiency graph of the test with 4096 x 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 \( \text{area} \). The area of the connected component of \( x \) can therefore be found in \( \text{area}[\text{root}(x)] \). Besides the invariants defined in section 2.3 we introduce the following new invariant

\[ J_0: \quad \forall (x \in D :: \text{area}[x] = \# \{ y \mid y \in D \land \text{root}(y) = x \}). \]

This invariant is initialized by

\[
\text{Tarea} \text{init: } \quad \text{for all } x \in D \text{ do } \\
\text{area}[x] = 1; \\
\text{par}[x] = x; \\
\text{od}. 
\]

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

\[
\text{Extend}(x, y): \quad \text{while } \text{par}[x] \neq x \text{ do } \\
\quad p := \text{par}[x]; \\
\quad \text{par}[x] := y; \\
\quad x := p; \\
\quad \text{od}; \\
\quad \text{if } x \neq y \text{ then } \\
\quad \quad \text{par}[x] = y; \\
\quad \quad \text{area}[y] := \text{area}[y] + \text{area}[x]; \\
\quad \quad \text{area}[x] := 0; \\
\quad \text{fi.}
\]
The same transformation can be made for the parallel algorithm. When in \( \text{Extend}(x,y, \text{origin}) \) the \( \text{par} \) 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 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.

\[
D = \{0, \ldots, m-1\} \times \{0, \ldots, n-1\},
\]

and there exists a binary image

\[
b : D \rightarrow \{\text{TRUE}, \text{FALSE}\}.
\]

We define the set \( B \) as

\[
B = \{(x,y) \in D \mid b(x,y) = \text{TRUE}\}.
\]

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. \( \sqrt{\text{EDT}(x,y)} \) is the exact Euclidean distance, \( \text{MDT}(x,y) \) is the Manhattan (city-block) distance, and \( \text{CDT}(x,y) \) is the Chessboard distance. These distances are defined by

\[
\begin{align*}
\text{EDT}(x,y) &= \min(i, j : 0 \leq i < m \land 0 \leq j < n \land b(i,j) : (x-i)^2 + (y-j)^2), \\
\text{MDT}(x,y) &= \min(i, j : 0 \leq i < m \land 0 \leq j < n \land b(i,j) : |x-i| + |y-j|), \\
\text{CDT}(x,y) &= \min(i, j : 0 \leq i < m \land 0 \leq j < n \land b(i,j) : |x-i| \max |y-j|).
\end{align*}
\]

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 \( \infty \), and use the rule \( z + \infty = \infty \) for all \( z \). Now we find with some calculation

\[
\begin{align*}
\text{EDT}(x,y) &= \min(i : 0 \leq i < m : (x-i)^2 + G(i,y)^2), \\
\text{MDT}(x,y) &= \min(i : 0 \leq i < m : |x-i| + G(i,y)), \\
\text{CDT}(x,y) &= \min(i : 0 \leq i < m : |x-i| \max G(i,y)),
\end{align*}
\]

where \( G(i,y) = \min(j : 0 \leq j < n \land b(i,j) : |y-j|) \).
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:

```plaintext
for all $x \in [0..m-1]$ do
    if $b(x,0)$ then
        $G[x,0] := 0$
    else
        $G[x,0] := \infty$
    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:

```plaintext
for all $y \in [0..n-1]$ do
    $q := 0$
    $s[0] := 0$
    $t[0] := 0$
    for $u := 1$ to $m-1$ do
        while $q \geq 0 \land f(t[q],y,s[q]) > f(t[q],y,u)$ do
            $q := q - 1$
        od;
        if $q < 0$ then
            $q := 0$
            $s[0] := u$
        else
            $w := 1 + \text{Sep}(s[q],y,u)$
            if $w < m$ then
                $q := q + 1$
            fi
        fi;
    od;
```
6.2 Distributed calculation of the distance transform

Figure 6.1: An example of the Manhattan distance transform.

\[
\begin{align*}
\text{s}[q] &:= u; \\
\text{t}[q] &:= w; \\
\text{f}; \\
\text{f}; \\
\text{od}; \\
\text{for } u := m - 1 \text{ downto } 0 \text{ do} \\
\text{dt}[u, y] &:= f(u, y, s[q]); \\
\text{if } u = t[q] \text{ then} \\
q &:= q - 1; \\
\text{f}; \\
\text{od}; \\
\text{od}; \\
\end{align*}
\]

The array \( \text{dt} \) 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 \( \text{Sep} \) depends on the metric that is used, i.e.

**EDT:**
\[
f(x, y, i) = (x - i)^2 + G[i, y]^2 \\
\text{Sep}(i, y, u) = \frac{(u^2 - i^2 + G[u, y])^2 - G[i, y]^2}{2(u - i)}
\]

**MDT:**
\[
f(x, y, i) = |x - i| + G[i, y] \\
\text{Sep}(i, y, u) = \begin{cases} 
G[u, y] + u - i & \text{if } G[i, y] \leq G[u, y] \\
-\infty & \text{if } G[i, y] > G[u, y] + u - i
\end{cases}
\]

**CDT:**
\[
f(x, y, i) = |x - i| \max G[i, y] \\
\text{Sep}(i, y, u) = \begin{cases} 
(i + G[u, y]) \max ((i + u) \div 2) & \text{if } G[i, y] \leq G[u, y] \\
(u - G[i, y]) \min ((i + u) \div 2) & \text{else}
\end{cases}
\]

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

$$DT_{par}:$$

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 $DT_{par}$. In this example, $\{C_5, C_6, D_5, D_6\}$ of $G$ are sent from process 1 to process 2. In e., $G$ is shown after Phase 2, statement 4 of $DT_{par}$.

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;
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;
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;

Phase 2:
for all \(y \in \{0..n-1\}\) do
  \(q := 0;\)
  \(s[0] := 0;\)
  \(t[0] := 0;\)
for \(u := 1\) to \(m - 1\) do
  while \(q \geq 0 \land f(t[q], y, s[q]) > f(t[q], y, u)\) do
    \(q := q - 1;\)
  od;
  if \(q < 0\) then
    \(q := 0;\)
    \(s[0] := u;\)
  else
    \(w := 1 + \text{Sep}(s[q], y, u);\)
    if \(w < m\) then
      \(q := q + 1;\)
      \(s[q] := u;\)
      \(t[q] := w;\)
    fi;
  fi;
for \(u := m - 1\) downto 0 do
  \(f_t[u, y] := q[u, s[q]];\)
  \(d_c[u, y] := f(u, y, s[q]);\)
  if \(u = t[q]\) then
    \(q := q - 1;\)
  fi;
od;
end;

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

\[\text{run} \text{Phase 1 to Column} \text{ss}\text{; for all }p \in \text{Processors do send Column} s \cap \text{Row} p \text{ of } G \text{ to } p;\]

send Columnss \cap Rowsp of q to p;
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 \( \text{lab} \) and \( \text{area} \) are available, and define that there exists a function

\[
\text{keepsize} : \mathbb{Z} \rightarrow \{\text{TRUE, FALSE}\}.
\]

We define the function \( b \) from section 6.2 as follows.

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

The feature transform is rewritten as follows. \( \text{lab}[x] \) 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 \( \text{par} \) and \( \text{area} \).

Phase 1:

\[
\text{for all } x \in [0..m - 1] \text{ do}
\]

\[
\text{if } b(x, 0) \text{ then}
\]

\[
G[x, 0] := 0;
\]

\[
\text{else}
\]

\[
\text{lab}[x, 0] := \bot;
\]

\[
G[x, 0] := \infty;
\]

\[
\text{fi;}
\]

\[
\text{for } y := 1 \text{ to } n - 1 \text{ do}
\]

\[
\text{if } b(x, y) \text{ then}
\]

\[
G[x, y] := 0;
\]

\[
\text{else}
\]

\[
\text{lab}[x, y] := \text{lab}[x, y - 1];
\]

\[
G[x, y] := 1 + G[x, y - 1];
\]

\[
\text{fi;}
\]

\[
\text{od;}
\]

\[
\text{for } y := n - 2 \text{ downto } 0 \text{ do}
\]

\[
\text{if } G[x, y + 1] < G[x, y] \text{ then}
\]
6.3 Merging of connected components

\[ \text{lab}[x,y] := \text{lab}[x,y + 1]; \]
\[ G[x,y] := 1 + G[x,y + 1]; \]
\[ \text{fi}; \]
\[ \text{od}; \]
\[ \text{od}; \]

Phase 2:
\[ \text{for all } y \in [0..n - 1] \text{ do} \]
\[ q := 0; \]
\[ s[0] := 0; \]
\[ t[0] := 0; \]
\[ \text{for } u := 1 \text{ to } m - 1 \text{ do} \]
\[ \text{while } q \geq 0 \land f(t[q], y, s[q]) > f(t[q], y, u) \text{ do} \]
\[ q := q - 1; \]
\[ \text{od}; \]
\[ \text{if } q < 0 \text{ then} \]
\[ q := 0; \]
\[ s[0] := u; \]
\[ \text{else} \]
\[ w := 1 + \text{Sep}(s[q], y, u); \]
\[ \text{if } w < m \text{ then} \]
\[ q := q + 1; \]
\[ s[q] := u; \]
\[ t[q] := w; \]
\[ \text{fi}; \]
\[ \text{fi}; \]
\[ \text{od}; \]
\[ \text{for } u := m - 1 \text{ downto } 0 \text{ do} \]
\[ \text{lab}[u,y] := \text{lab}[u,s[q]]; \]
\[ dt[u,y] := f(u,y,s[q]); \]
\[ \text{if } u = t[q] \text{ then} \]
\[ q := q - 1; \]
\[ \text{fi}; \]
\[ \text{od}; \]
\[ \text{od}. \]

After Phase 2 has been executed, the array \text{lab} is the labelling of the new image.

\textbf{Note:} When one prefers to merge components in the original image, the array \text{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 \text{keeps}(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 \text{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 \text{lab} is shown for the pixels whose \text{lab} has been changed because of their size. Figure d. is the new array \text{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.

\[ \text{CCRpar}_k: \]

1: apply \text{Phase 1 to} \text{Columns}_k;
2: \text{forall} \ p \in \text{Processes} \ \text{do}
   \hspace{1em} \text{send} \text{Columns}_k \cap \text{Rows}_p \text{ of } G \text{ to } p;
   \hspace{1em} \text{send} \text{Columns}_k \cap \text{Rows}_p \text{ of } \text{lab} \text{ to } p;
   \text{od};
3: \text{forall} \ p \in \text{Processes} \ \text{do}
   \hspace{1em} \text{receive} \text{Rows}_k \cap \text{Columns}_p \text{ of } G \text{ from } p;
   \hspace{1em} \text{receive} \text{Rows}_k \cap \text{Columns}_p \text{ of } \text{lab} \text{ from } p;
   \text{od};
4: apply \text{Phase 2 to} \text{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.
6.3 Merging of connected components

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

Figure 6.5: An example of the component merging algorithm.
Bibliography


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.
<table>
<thead>
<tr>
<th>image</th>
<th>#P</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>1</td>
<td>19</td>
<td>82</td>
<td>338</td>
<td>1483</td>
<td>6208</td>
<td>24281</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>45</td>
<td>174</td>
<td>739</td>
<td>3067</td>
<td>12351</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>31</td>
<td>121</td>
<td>484</td>
<td>2149</td>
<td>8353</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>24</td>
<td>89</td>
<td>364</td>
<td>1533</td>
<td>6237</td>
</tr>
<tr>
<td>vertical</td>
<td>1</td>
<td>19</td>
<td>77</td>
<td>316</td>
<td>1419</td>
<td>5668</td>
<td>23010</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>44</td>
<td>162</td>
<td>710</td>
<td>2882</td>
<td>11637</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>10</td>
<td>31</td>
<td>110</td>
<td>480</td>
<td>1982</td>
<td>7761</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>25</td>
<td>86</td>
<td>357</td>
<td>1466</td>
<td>5724</td>
</tr>
<tr>
<td>horizontal</td>
<td>1</td>
<td>19</td>
<td>79</td>
<td>316</td>
<td>1397</td>
<td>5776</td>
<td>23536</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>40</td>
<td>166</td>
<td>712</td>
<td>2985</td>
<td>12112</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
<td>28</td>
<td>110</td>
<td>477</td>
<td>1963</td>
<td>8203</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>23</td>
<td>81</td>
<td>359</td>
<td>1464</td>
<td>6068</td>
</tr>
<tr>
<td>comb</td>
<td>1</td>
<td>18</td>
<td>78</td>
<td>316</td>
<td>1396</td>
<td>5715</td>
<td>22861</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>44</td>
<td>164</td>
<td>710</td>
<td>2967</td>
<td>11584</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>11</td>
<td>33</td>
<td>114</td>
<td>484</td>
<td>1939</td>
<td>7842</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>27</td>
<td>90</td>
<td>363</td>
<td>1448</td>
<td>5850</td>
</tr>
<tr>
<td>random</td>
<td>1</td>
<td>20</td>
<td>83</td>
<td>336</td>
<td>1473</td>
<td>5974</td>
<td>24208</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14</td>
<td>44</td>
<td>164</td>
<td>710</td>
<td>2967</td>
<td>11584</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>11</td>
<td>33</td>
<td>114</td>
<td>484</td>
<td>1939</td>
<td>7842</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>27</td>
<td>90</td>
<td>363</td>
<td>1448</td>
<td>5850</td>
</tr>
<tr>
<td>music</td>
<td>1</td>
<td>20</td>
<td>82</td>
<td>330</td>
<td>1473</td>
<td>5974</td>
<td>24208</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14</td>
<td>44</td>
<td>172</td>
<td>710</td>
<td>2967</td>
<td>11584</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>11</td>
<td>33</td>
<td>117</td>
<td>494</td>
<td>2049</td>
<td>8263</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>27</td>
<td>90</td>
<td>363</td>
<td>1448</td>
<td>5850</td>
</tr>
<tr>
<td>ct</td>
<td>1</td>
<td>18</td>
<td>75</td>
<td>325</td>
<td>1467</td>
<td>6025</td>
<td>24172</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>42</td>
<td>168</td>
<td>732</td>
<td>3069</td>
<td>12391</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>30</td>
<td>115</td>
<td>490</td>
<td>2180</td>
<td>8434</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>23</td>
<td>86</td>
<td>368</td>
<td>1467</td>
<td>6255</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>image</th>
<th>#P</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>2</td>
<td>79.2%</td>
<td>91.1%</td>
<td>97.1%</td>
<td>100.3%</td>
<td>101.2%</td>
<td>98.3%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>70.4%</td>
<td>88.2%</td>
<td>93.1%</td>
<td>102.1%</td>
<td>96.3%</td>
<td>96.9%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>67.9%</td>
<td>85.4%</td>
<td>94.9%</td>
<td>101.9%</td>
<td>96.6%</td>
<td>97.3%</td>
</tr>
<tr>
<td>vertical</td>
<td>2</td>
<td>79.2%</td>
<td>89.5%</td>
<td>97.5%</td>
<td>99.9%</td>
<td>98.3%</td>
<td>98.9%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>63.3%</td>
<td>82.8%</td>
<td>95.8%</td>
<td>98.5%</td>
<td>95.3%</td>
<td>98.8%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>59.4%</td>
<td>77.0%</td>
<td>91.9%</td>
<td>99.4%</td>
<td>96.7%</td>
<td>100.5%</td>
</tr>
<tr>
<td>horizontal</td>
<td>2</td>
<td>86.4%</td>
<td>94.0%</td>
<td>95.2%</td>
<td>98.1%</td>
<td>96.8%</td>
<td>97.2%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>79.2%</td>
<td>94.0%</td>
<td>95.8%</td>
<td>97.6%</td>
<td>98.1%</td>
<td>96.5%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>67.9%</td>
<td>85.9%</td>
<td>97.5%</td>
<td>97.3%</td>
<td>98.6%</td>
<td>97.0%</td>
</tr>
<tr>
<td>comb</td>
<td>2</td>
<td>75.0%</td>
<td>88.6%</td>
<td>96.3%</td>
<td>98.3%</td>
<td>96.3%</td>
<td>98.7%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>54.5%</td>
<td>78.8%</td>
<td>92.4%</td>
<td>96.1%</td>
<td>98.2%</td>
<td>97.2%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>45.0%</td>
<td>72.2%</td>
<td>87.8%</td>
<td>96.1%</td>
<td>98.7%</td>
<td>97.7%</td>
</tr>
<tr>
<td>random</td>
<td>2</td>
<td>90.9%</td>
<td>92.2%</td>
<td>97.1%</td>
<td>99.3%</td>
<td>98.4%</td>
<td>97.6%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>86.7%</td>
<td>86.5%</td>
<td>96.6%</td>
<td>99.8%</td>
<td>96.6%</td>
<td>99.4%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>55.6%</td>
<td>86.5%</td>
<td>95.6%</td>
<td>98.2%</td>
<td>99.9%</td>
<td>98.8%</td>
</tr>
<tr>
<td>music</td>
<td>2</td>
<td>83.3%</td>
<td>93.2%</td>
<td>107.6%</td>
<td>98.0%</td>
<td>100.6%</td>
<td>98.0%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>60.6%</td>
<td>85.4%</td>
<td>105.4%</td>
<td>99.5%</td>
<td>98.6%</td>
<td>98.8%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>71.4%</td>
<td>85.4%</td>
<td>106.3%</td>
<td>101.2%</td>
<td>102.2%</td>
<td>96.0%</td>
</tr>
<tr>
<td>ct</td>
<td>2</td>
<td>81.8%</td>
<td>89.3%</td>
<td>96.7%</td>
<td>100.2%</td>
<td>97.5%</td>
<td>97.5%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>66.7%</td>
<td>83.3%</td>
<td>94.2%</td>
<td>99.8%</td>
<td>92.1%</td>
<td>95.5%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>56.2%</td>
<td>81.5%</td>
<td>94.5%</td>
<td>99.7%</td>
<td>101.3%</td>
<td>96.6%</td>
</tr>
</tbody>
</table>

Table A.2: The efficiency calculated from table A.1
### Table A.3: Performance test of the implementation of Tarjan's disjoint set algorithm.
The forward version is used with a mpart of 4.

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

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

<table>
<thead>
<tr>
<th>image</th>
<th>#P</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>2</td>
<td>90.9%</td>
<td>93.2%</td>
<td>101.2%</td>
<td>99.1%</td>
<td>102.7%</td>
<td>101.2%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>74.1%</td>
<td>86.2%</td>
<td>98.6%</td>
<td>100.8%</td>
<td>99.5%</td>
<td>98.0%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>71.4%</td>
<td>85.4%</td>
<td>98.3%</td>
<td>100.8%</td>
<td>103.0%</td>
<td>98.8%</td>
</tr>
<tr>
<td>vertical</td>
<td>2</td>
<td>75.0%</td>
<td>93.9%</td>
<td>99.1%</td>
<td>97.3%</td>
<td>98.1%</td>
<td>97.9%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>54.5%</td>
<td>77.8%</td>
<td>92.7%</td>
<td>98.2%</td>
<td>99.5%</td>
<td>100.8%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>40.9%</td>
<td>71.3%</td>
<td>89.0%</td>
<td>96.8%</td>
<td>99.8%</td>
<td>97.0%</td>
</tr>
<tr>
<td>horizontal</td>
<td>2</td>
<td>86.4%</td>
<td>94.0%</td>
<td>99.7%</td>
<td>99.4%</td>
<td>97.1%</td>
<td>100.5%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>70.4%</td>
<td>87.8%</td>
<td>98.8%</td>
<td>98.3%</td>
<td>97.6%</td>
<td>102.4%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>67.9%</td>
<td>85.9%</td>
<td>98.5%</td>
<td>99.2%</td>
<td>99.3%</td>
<td>101.5%</td>
</tr>
<tr>
<td>comb</td>
<td>2</td>
<td>79.2%</td>
<td>89.5%</td>
<td>96.0%</td>
<td>97.5%</td>
<td>97.8%</td>
<td>96.6%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>57.6%</td>
<td>77.6%</td>
<td>91.5%</td>
<td>95.6%</td>
<td>95.5%</td>
<td>94.1%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>47.5%</td>
<td>68.8%</td>
<td>80.7%</td>
<td>96.1%</td>
<td>98.8%</td>
<td>97.4%</td>
</tr>
<tr>
<td>random</td>
<td>2</td>
<td>83.3%</td>
<td>93.3%</td>
<td>102.9%</td>
<td>97.1%</td>
<td>94.6%</td>
<td>99.6%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>66.7%</td>
<td>87.5%</td>
<td>96.2%</td>
<td>98.4%</td>
<td>99.1%</td>
<td>101.5%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>55.6%</td>
<td>84.0%</td>
<td>100.0%</td>
<td>97.2%</td>
<td>97.7%</td>
<td>97.3%</td>
</tr>
<tr>
<td>music</td>
<td>2</td>
<td>83.3%</td>
<td>95.6%</td>
<td>98.3%</td>
<td>101.6%</td>
<td>106.1%</td>
<td>99.8%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>66.7%</td>
<td>87.5%</td>
<td>98.0%</td>
<td>101.2%</td>
<td>107.1%</td>
<td>100.3%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>38.5%</td>
<td>87.5%</td>
<td>96.0%</td>
<td>102.5%</td>
<td>110.5%</td>
<td>99.6%</td>
</tr>
<tr>
<td>ct</td>
<td>2</td>
<td>81.8%</td>
<td>92.7%</td>
<td>98.5%</td>
<td>100.5%</td>
<td>99.9%</td>
<td>97.4%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>66.7%</td>
<td>81.7%</td>
<td>94.0%</td>
<td>100.1%</td>
<td>101.8%</td>
<td>94.9%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>64.3%</td>
<td>82.6%</td>
<td>94.0%</td>
<td>99.2%</td>
<td>101.7%</td>
<td>97.2%</td>
</tr>
</tbody>
</table>
Table A.5: Performance test of the implementation of Tarjan’s disjoint set algorithm. The forward version is used with a mpart of 16.

<table>
<thead>
<tr>
<th>image</th>
<th>#P</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>1</td>
<td>20</td>
<td>84</td>
<td>337</td>
<td>1464</td>
<td>6024</td>
<td>24087</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>44</td>
<td>176</td>
<td>750</td>
<td>3009</td>
<td>12914</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>31</td>
<td>116</td>
<td>486</td>
<td>2136</td>
<td>8256</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>25</td>
<td>89</td>
<td>373</td>
<td>1517</td>
<td>6342</td>
</tr>
<tr>
<td>vertical</td>
<td>1</td>
<td>19</td>
<td>77</td>
<td>317</td>
<td>1378</td>
<td>5676</td>
<td>22962</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13</td>
<td>43</td>
<td>164</td>
<td>716</td>
<td>2907</td>
<td>11762</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>12</td>
<td>32</td>
<td>116</td>
<td>473</td>
<td>1956</td>
<td>8243</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>14</td>
<td>29</td>
<td>88</td>
<td>361</td>
<td>1442</td>
<td>5866</td>
</tr>
<tr>
<td>horizontal</td>
<td>1</td>
<td>18</td>
<td>79</td>
<td>321</td>
<td>1427</td>
<td>5938</td>
<td>23282</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>42</td>
<td>165</td>
<td>710</td>
<td>3008</td>
<td>11866</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>29</td>
<td>106</td>
<td>475</td>
<td>1972</td>
<td>7849</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>23</td>
<td>85</td>
<td>357</td>
<td>1469</td>
<td>6079</td>
</tr>
<tr>
<td>comb</td>
<td>1</td>
<td>18</td>
<td>78</td>
<td>315</td>
<td>1376</td>
<td>5608</td>
<td>22742</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13</td>
<td>42</td>
<td>164</td>
<td>705</td>
<td>2913</td>
<td>12177</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>18</td>
<td>37</td>
<td>116</td>
<td>484</td>
<td>1962</td>
<td>7857</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>21</td>
<td>34</td>
<td>93</td>
<td>365</td>
<td>1440</td>
<td>5887</td>
</tr>
<tr>
<td>random</td>
<td>1</td>
<td>20</td>
<td>85</td>
<td>351</td>
<td>1445</td>
<td>6197</td>
<td>24404</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>45</td>
<td>172</td>
<td>755</td>
<td>3058</td>
<td>12387</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>11</td>
<td>32</td>
<td>116</td>
<td>492</td>
<td>2050</td>
<td>8389</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>11</td>
<td>25</td>
<td>88</td>
<td>375</td>
<td>1530</td>
<td>6301</td>
</tr>
<tr>
<td>music</td>
<td>1</td>
<td>20</td>
<td>83</td>
<td>339</td>
<td>1449</td>
<td>6197</td>
<td>24404</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13</td>
<td>44</td>
<td>171</td>
<td>727</td>
<td>3124</td>
<td>12931</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>11</td>
<td>31</td>
<td>115</td>
<td>492</td>
<td>2032</td>
<td>8274</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>24</td>
<td>89</td>
<td>374</td>
<td>1519</td>
<td>6314</td>
</tr>
<tr>
<td>ct</td>
<td>1</td>
<td>18</td>
<td>76</td>
<td>327</td>
<td>1458</td>
<td>6059</td>
<td>25463</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>43</td>
<td>170</td>
<td>753</td>
<td>3082</td>
<td>12373</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>14</td>
<td>32</td>
<td>122</td>
<td>500</td>
<td>2021</td>
<td>8183</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>23</td>
<td>85</td>
<td>370</td>
<td>1522</td>
<td>6392</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>image</th>
<th>#P</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>1</td>
<td>83.3%</td>
<td>95.5%</td>
<td>95.7%</td>
<td>97.6%</td>
<td>100.1%</td>
<td>93.3%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>74.1%</td>
<td>90.3%</td>
<td>96.8%</td>
<td>100.4%</td>
<td>94.0%</td>
<td>97.3%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>62.5%</td>
<td>84.0%</td>
<td>94.7%</td>
<td>98.1%</td>
<td>99.3%</td>
<td>96.0%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>52.8%</td>
<td>80.2%</td>
<td>91.1%</td>
<td>97.8%</td>
<td>96.7%</td>
<td>92.9%</td>
</tr>
<tr>
<td>vertical</td>
<td>1</td>
<td>73.1%</td>
<td>89.5%</td>
<td>96.6%</td>
<td>96.2%</td>
<td>97.6%</td>
<td>97.7%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>52.9%</td>
<td>65.7%</td>
<td>91.1%</td>
<td>97.8%</td>
<td>96.7%</td>
<td>92.9%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>43.9%</td>
<td>56.4%</td>
<td>90.1%</td>
<td>94.8%</td>
<td>96.4%</td>
<td>97.9%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>33.9%</td>
<td>56.6%</td>
<td>90.1%</td>
<td>94.8%</td>
<td>96.4%</td>
<td>97.9%</td>
</tr>
<tr>
<td>horizontal</td>
<td>1</td>
<td>81.6%</td>
<td>94.0%</td>
<td>97.3%</td>
<td>100.5%</td>
<td>98.7%</td>
<td>98.1%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>66.7%</td>
<td>88.0%</td>
<td>99.0%</td>
<td>100.1%</td>
<td>100.4%</td>
<td>96.9%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>64.3%</td>
<td>55.9%</td>
<td>94.4%</td>
<td>99.9%</td>
<td>101.1%</td>
<td>95.7%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>69.2%</td>
<td>59.5%</td>
<td>96.0%</td>
<td>97.6%</td>
<td>96.3%</td>
<td>93.4%</td>
</tr>
<tr>
<td>comb</td>
<td>1</td>
<td>72.9%</td>
<td>92.9%</td>
<td>96.0%</td>
<td>97.6%</td>
<td>96.3%</td>
<td>93.4%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>66.7%</td>
<td>88.0%</td>
<td>99.0%</td>
<td>100.1%</td>
<td>100.4%</td>
<td>96.9%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>64.3%</td>
<td>55.9%</td>
<td>94.4%</td>
<td>99.9%</td>
<td>101.1%</td>
<td>95.7%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>69.2%</td>
<td>59.5%</td>
<td>96.0%</td>
<td>97.6%</td>
<td>96.3%</td>
<td>93.4%</td>
</tr>
<tr>
<td>random</td>
<td>1</td>
<td>83.3%</td>
<td>94.4%</td>
<td>102.0%</td>
<td>95.7%</td>
<td>101.5%</td>
<td>98.5%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>60.6%</td>
<td>88.5%</td>
<td>100.9%</td>
<td>97.9%</td>
<td>100.8%</td>
<td>97.0%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>45.5%</td>
<td>85.0%</td>
<td>99.7%</td>
<td>96.3%</td>
<td>101.3%</td>
<td>96.6%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>66.3%</td>
<td>86.5%</td>
<td>99.9%</td>
<td>96.3%</td>
<td>101.3%</td>
<td>96.6%</td>
</tr>
<tr>
<td>music</td>
<td>1</td>
<td>76.9%</td>
<td>94.3%</td>
<td>99.1%</td>
<td>97.7%</td>
<td>96.3%</td>
<td>97.7%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>60.6%</td>
<td>89.2%</td>
<td>98.3%</td>
<td>98.2%</td>
<td>97.2%</td>
<td>98.6%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>62.5%</td>
<td>86.5%</td>
<td>95.2%</td>
<td>96.9%</td>
<td>97.5%</td>
<td>96.9%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>69.2%</td>
<td>92.9%</td>
<td>96.0%</td>
<td>97.6%</td>
<td>96.3%</td>
<td>93.4%</td>
</tr>
<tr>
<td>ct</td>
<td>1</td>
<td>81.8%</td>
<td>88.4%</td>
<td>96.2%</td>
<td>96.8%</td>
<td>98.3%</td>
<td>102.9%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>42.9%</td>
<td>79.2%</td>
<td>89.3%</td>
<td>97.2%</td>
<td>99.9%</td>
<td>103.7%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>64.3%</td>
<td>82.6%</td>
<td>95.1%</td>
<td>98.5%</td>
<td>99.5%</td>
<td>99.6%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>69.2%</td>
<td>92.9%</td>
<td>96.0%</td>
<td>96.8%</td>
<td>98.3%</td>
<td>102.9%</td>
</tr>
<tr>
<td>Image</td>
<td>#P</td>
<td>128</td>
<td>256</td>
<td>512</td>
<td>1024</td>
<td>2048</td>
<td>4096</td>
</tr>
<tr>
<td>---------</td>
<td>----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>empty</td>
<td>1</td>
<td>20</td>
<td>88</td>
<td>351</td>
<td>1510</td>
<td>6019</td>
<td>24735</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>44</td>
<td>172</td>
<td>752</td>
<td>3107</td>
<td>12684</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>31</td>
<td>117</td>
<td>507</td>
<td>2096</td>
<td>8274</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>23</td>
<td>89</td>
<td>382</td>
<td>1504</td>
<td>6284</td>
</tr>
<tr>
<td>vertical</td>
<td>1</td>
<td>18</td>
<td>77</td>
<td>315</td>
<td>1405</td>
<td>5757</td>
<td>23150</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>21</td>
<td>49</td>
<td>171</td>
<td>716</td>
<td>2920</td>
<td>11601</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>23</td>
<td>43</td>
<td>125</td>
<td>481</td>
<td>1918</td>
<td>7685</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>22</td>
<td>40</td>
<td>101</td>
<td>362</td>
<td>1434</td>
<td>5747</td>
</tr>
<tr>
<td>horizontal</td>
<td>1</td>
<td>19</td>
<td>77</td>
<td>321</td>
<td>1406</td>
<td>5719</td>
<td>23413</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>42</td>
<td>166</td>
<td>718</td>
<td>3012</td>
<td>11924</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
<td>29</td>
<td>107</td>
<td>470</td>
<td>1975</td>
<td>7896</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>23</td>
<td>83</td>
<td>360</td>
<td>1455</td>
<td>5966</td>
</tr>
<tr>
<td>comb</td>
<td>1</td>
<td>19</td>
<td>76</td>
<td>313</td>
<td>1378</td>
<td>5610</td>
<td>23210</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>21</td>
<td>48</td>
<td>169</td>
<td>721</td>
<td>2904</td>
<td>11437</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>23</td>
<td>47</td>
<td>125</td>
<td>482</td>
<td>1976</td>
<td>7784</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>39</td>
<td>45</td>
<td>106</td>
<td>373</td>
<td>1560</td>
<td>5871</td>
</tr>
<tr>
<td>random</td>
<td>1</td>
<td>12</td>
<td>45</td>
<td>174</td>
<td>750</td>
<td>3039</td>
<td>12586</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10</td>
<td>34</td>
<td>121</td>
<td>505</td>
<td>2004</td>
<td>8136</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>25</td>
<td>94</td>
<td>381</td>
<td>1687</td>
<td>6288</td>
</tr>
<tr>
<td>music</td>
<td>1</td>
<td>20</td>
<td>83</td>
<td>335</td>
<td>1498</td>
<td>6047</td>
<td>24490</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>44</td>
<td>175</td>
<td>749</td>
<td>3082</td>
<td>12195</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>10</td>
<td>35</td>
<td>119</td>
<td>492</td>
<td>2075</td>
<td>8254</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>26</td>
<td>88</td>
<td>379</td>
<td>1492</td>
<td>6181</td>
</tr>
<tr>
<td>ct</td>
<td>1</td>
<td>18</td>
<td>76</td>
<td>329</td>
<td>1478</td>
<td>6011</td>
<td>24153</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>42</td>
<td>171</td>
<td>743</td>
<td>3090</td>
<td>12586</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>12</td>
<td>33</td>
<td>117</td>
<td>491</td>
<td>2076</td>
<td>8256</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>25</td>
<td>86</td>
<td>372</td>
<td>1526</td>
<td>6462</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Image</th>
<th>#P</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>2</td>
<td>83.3%</td>
<td>100.0%</td>
<td>102.0%</td>
<td>100.4%</td>
<td>98.9%</td>
<td>97.5%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>74.1%</td>
<td>94.6%</td>
<td>100.0%</td>
<td>99.3%</td>
<td>95.7%</td>
<td>99.6%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>71.4%</td>
<td>95.7%</td>
<td>98.6%</td>
<td>98.8%</td>
<td>100.0%</td>
<td>98.4%</td>
</tr>
<tr>
<td>vertical</td>
<td>2</td>
<td>42.9%</td>
<td>78.6%</td>
<td>92.1%</td>
<td>98.1%</td>
<td>98.6%</td>
<td>99.8%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>26.1%</td>
<td>59.7%</td>
<td>84.0%</td>
<td>97.4%</td>
<td>100.1%</td>
<td>100.4%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>20.5%</td>
<td>48.1%</td>
<td>78.0%</td>
<td>97.0%</td>
<td>100.4%</td>
<td>100.7%</td>
</tr>
<tr>
<td>horizontal</td>
<td>2</td>
<td>86.4%</td>
<td>91.7%</td>
<td>96.7%</td>
<td>97.5%</td>
<td>94.9%</td>
<td>98.2%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>79.2%</td>
<td>88.5%</td>
<td>100.0%</td>
<td>99.7%</td>
<td>96.5%</td>
<td>98.8%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>67.9%</td>
<td>83.7%</td>
<td>96.7%</td>
<td>97.6%</td>
<td>98.3%</td>
<td>98.1%</td>
</tr>
<tr>
<td>comb</td>
<td>2</td>
<td>45.2%</td>
<td>79.2%</td>
<td>92.6%</td>
<td>95.6%</td>
<td>96.6%</td>
<td>101.5%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>27.5%</td>
<td>53.9%</td>
<td>82.8%</td>
<td>95.3%</td>
<td>94.6%</td>
<td>99.4%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>12.2%</td>
<td>42.2%</td>
<td>73.8%</td>
<td>92.4%</td>
<td>89.9%</td>
<td>98.8%</td>
</tr>
<tr>
<td>random</td>
<td>2</td>
<td>63.3%</td>
<td>91.1%</td>
<td>97.7%</td>
<td>98.3%</td>
<td>100.4%</td>
<td>94.0%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>66.7%</td>
<td>80.4%</td>
<td>93.7%</td>
<td>95.4%</td>
<td>101.5%</td>
<td>96.9%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>55.6%</td>
<td>82.0%</td>
<td>90.4%</td>
<td>94.8%</td>
<td>90.4%</td>
<td>94.1%</td>
</tr>
<tr>
<td>music</td>
<td>2</td>
<td>83.3%</td>
<td>94.3%</td>
<td>95.7%</td>
<td>100.0%</td>
<td>98.1%</td>
<td>100.4%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>66.7%</td>
<td>79.0%</td>
<td>93.8%</td>
<td>101.5%</td>
<td>97.1%</td>
<td>98.9%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>71.4%</td>
<td>79.8%</td>
<td>95.2%</td>
<td>98.8%</td>
<td>101.3%</td>
<td>99.1%</td>
</tr>
<tr>
<td>ct</td>
<td>2</td>
<td>81.8%</td>
<td>90.5%</td>
<td>96.2%</td>
<td>99.5%</td>
<td>97.3%</td>
<td>97.7%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>50.0%</td>
<td>76.8%</td>
<td>93.7%</td>
<td>100.3%</td>
<td>95.5%</td>
<td>97.5%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>45.0%</td>
<td>76.0%</td>
<td>95.6%</td>
<td>99.3%</td>
<td>98.5%</td>
<td>93.4%</td>
</tr>
</tbody>
</table>

Table A.8: The efficiency calculated from table A.7
Table A.9: Performance test of the implementation of Tarjan's disjoint set algorithm. The forward version is used with a *mpart* of 256.

<table>
<thead>
<tr>
<th>image</th>
<th>#P</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>1</td>
<td>20</td>
<td>84</td>
<td>340</td>
<td>1501</td>
<td>6094</td>
<td>23955</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>47</td>
<td>173</td>
<td>757</td>
<td>3005</td>
<td>12723</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>32</td>
<td>117</td>
<td>483</td>
<td>2073</td>
<td>8664</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>24</td>
<td>89</td>
<td>382</td>
<td>1520</td>
<td>6205</td>
</tr>
<tr>
<td>vertical</td>
<td>1</td>
<td>19</td>
<td>77</td>
<td>316</td>
<td>1386</td>
<td>5654</td>
<td>22847</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>26</td>
<td>72</td>
<td>195</td>
<td>740</td>
<td>2934</td>
<td>11601</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>39</td>
<td>87</td>
<td>165</td>
<td>537</td>
<td>1979</td>
<td>7782</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>46</td>
<td>117</td>
<td>177</td>
<td>442</td>
<td>1725</td>
<td>5913</td>
</tr>
<tr>
<td>horizontal</td>
<td>1</td>
<td>19</td>
<td>78</td>
<td>324</td>
<td>1426</td>
<td>5808</td>
<td>23628</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>42</td>
<td>188</td>
<td>714</td>
<td>3013</td>
<td>11897</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
<td>28</td>
<td>110</td>
<td>484</td>
<td>2009</td>
<td>7909</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>23</td>
<td>83</td>
<td>358</td>
<td>1476</td>
<td>6076</td>
</tr>
<tr>
<td>comb</td>
<td>1</td>
<td>19</td>
<td>77</td>
<td>313</td>
<td>1379</td>
<td>5762</td>
<td>24518</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>26</td>
<td>69</td>
<td>194</td>
<td>731</td>
<td>2915</td>
<td>11721</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>40</td>
<td>98</td>
<td>179</td>
<td>532</td>
<td>2036</td>
<td>7726</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>48</td>
<td>115</td>
<td>198</td>
<td>463</td>
<td>1581</td>
<td>5902</td>
</tr>
<tr>
<td>random</td>
<td>1</td>
<td>20</td>
<td>84</td>
<td>339</td>
<td>1462</td>
<td>5938</td>
<td>24632</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14</td>
<td>45</td>
<td>175</td>
<td>722</td>
<td>2986</td>
<td>12487</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>12</td>
<td>35</td>
<td>118</td>
<td>505</td>
<td>2012</td>
<td>8239</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>11</td>
<td>31</td>
<td>90</td>
<td>361</td>
<td>1501</td>
<td>5338</td>
</tr>
<tr>
<td>music</td>
<td>1</td>
<td>20</td>
<td>83</td>
<td>336</td>
<td>1470</td>
<td>5967</td>
<td>24292</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13</td>
<td>45</td>
<td>172</td>
<td>752</td>
<td>3102</td>
<td>12235</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>16</td>
<td>38</td>
<td>122</td>
<td>515</td>
<td>2069</td>
<td>8697</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>29</td>
<td>90</td>
<td>368</td>
<td>1488</td>
<td>6203</td>
</tr>
<tr>
<td>ct</td>
<td>1</td>
<td>18</td>
<td>75</td>
<td>327</td>
<td>1467</td>
<td>5972</td>
<td>24598</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14</td>
<td>45</td>
<td>168</td>
<td>729</td>
<td>3107</td>
<td>12755</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>13</td>
<td>49</td>
<td>206</td>
<td>499</td>
<td>2108</td>
<td>8303</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>13</td>
<td>32</td>
<td>92</td>
<td>376</td>
<td>1522</td>
<td>6120</td>
</tr>
</tbody>
</table>

Table A.10: The efficiency calculated from table A.9
<table>
<thead>
<tr>
<th>image</th>
<th>#P</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>1</td>
<td>19</td>
<td>83</td>
<td>352</td>
<td>1472</td>
<td>6260</td>
<td>24401</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>46</td>
<td>170</td>
<td>746</td>
<td>3106</td>
<td>12631</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>33</td>
<td>118</td>
<td>499</td>
<td>2266</td>
<td>8189</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>25</td>
<td>88</td>
<td>376</td>
<td>1521</td>
<td>6337</td>
</tr>
<tr>
<td>vertical</td>
<td>1</td>
<td>19</td>
<td>76</td>
<td>317</td>
<td>1383</td>
<td>5759</td>
<td>23071</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>30</td>
<td>73</td>
<td>228</td>
<td>835</td>
<td>3197</td>
<td>12565</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>42</td>
<td>83</td>
<td>241</td>
<td>722</td>
<td>2478</td>
<td>9025</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>45</td>
<td>112</td>
<td>263</td>
<td>753</td>
<td>2150</td>
<td>7379</td>
</tr>
<tr>
<td>horizontal</td>
<td>1</td>
<td>19</td>
<td>79</td>
<td>312</td>
<td>1426</td>
<td>5806</td>
<td>23300</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>41</td>
<td>167</td>
<td>716</td>
<td>3009</td>
<td>11853</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>30</td>
<td>113</td>
<td>479</td>
<td>1998</td>
<td>8377</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>23</td>
<td>84</td>
<td>358</td>
<td>1454</td>
<td>5934</td>
</tr>
<tr>
<td>comb</td>
<td>1</td>
<td>18</td>
<td>76</td>
<td>316</td>
<td>1377</td>
<td>5703</td>
<td>22622</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>25</td>
<td>71</td>
<td>228</td>
<td>828</td>
<td>3144</td>
<td>12296</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>39</td>
<td>100</td>
<td>242</td>
<td>727</td>
<td>2493</td>
<td>8867</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>49</td>
<td>117</td>
<td>271</td>
<td>765</td>
<td>2200</td>
<td>7595</td>
</tr>
<tr>
<td>random</td>
<td>1</td>
<td>20</td>
<td>83</td>
<td>350</td>
<td>1427</td>
<td>6129</td>
<td>24842</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13</td>
<td>46</td>
<td>173</td>
<td>737</td>
<td>3049</td>
<td>12938</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>11</td>
<td>36</td>
<td>120</td>
<td>484</td>
<td>2066</td>
<td>8384</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>12</td>
<td>34</td>
<td>92</td>
<td>373</td>
<td>1500</td>
<td>6282</td>
</tr>
<tr>
<td>music</td>
<td>1</td>
<td>20</td>
<td>82</td>
<td>332</td>
<td>1497</td>
<td>5915</td>
<td>24692</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>45</td>
<td>173</td>
<td>729</td>
<td>3045</td>
<td>12469</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>11</td>
<td>36</td>
<td>121</td>
<td>493</td>
<td>2065</td>
<td>8478</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>29</td>
<td>93</td>
<td>374</td>
<td>1515</td>
<td>6120</td>
</tr>
<tr>
<td>ct</td>
<td>1</td>
<td>18</td>
<td>75</td>
<td>327</td>
<td>1471</td>
<td>5972</td>
<td>24547</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13</td>
<td>44</td>
<td>169</td>
<td>739</td>
<td>3063</td>
<td>12524</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>14</td>
<td>43</td>
<td>141</td>
<td>530</td>
<td>2064</td>
<td>8236</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>12</td>
<td>29</td>
<td>93</td>
<td>375</td>
<td>1518</td>
<td>6411</td>
</tr>
</tbody>
</table>

Table A.11: Performance test of the implementation of Tarjan's disjoint set algorithm. The forward version is used with a `mpart` of ∞.
Table A.13: Performance test of the implementation of Tarjan’s disjoint set algorithm. The backward version is used with a mpart of 1.

<table>
<thead>
<tr>
<th>Image</th>
<th>#P</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>1</td>
<td>19</td>
<td>79</td>
<td>328</td>
<td>1445</td>
<td>5967</td>
<td>24073</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>67</td>
<td>168</td>
<td>737</td>
<td>3143</td>
<td>12670</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>30</td>
<td>113</td>
<td>489</td>
<td>2041</td>
<td>8090</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>23</td>
<td>88</td>
<td>371</td>
<td>1494</td>
<td>6281</td>
</tr>
<tr>
<td>vertical</td>
<td>1</td>
<td>17</td>
<td>70</td>
<td>285</td>
<td>1257</td>
<td>5191</td>
<td>20783</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13</td>
<td>48</td>
<td>180</td>
<td>820</td>
<td>3710</td>
<td>14966</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>11</td>
<td>35</td>
<td>126</td>
<td>564</td>
<td>2505</td>
<td>10249</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>9</td>
<td>27</td>
<td>96</td>
<td>421</td>
<td>1839</td>
<td>7619</td>
</tr>
<tr>
<td>horizontal</td>
<td>1</td>
<td>16</td>
<td>67</td>
<td>274</td>
<td>1255</td>
<td>5200</td>
<td>21071</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>9</td>
<td>36</td>
<td>141</td>
<td>648</td>
<td>2655</td>
<td>10617</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>26</td>
<td>95</td>
<td>431</td>
<td>1812</td>
<td>7052</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>20</td>
<td>71</td>
<td>320</td>
<td>1330</td>
<td>5319</td>
</tr>
<tr>
<td>comb</td>
<td>1</td>
<td>17</td>
<td>73</td>
<td>296</td>
<td>1423</td>
<td>5846</td>
<td>23787</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>15</td>
<td>55</td>
<td>214</td>
<td>949</td>
<td>4309</td>
<td>17706</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>14</td>
<td>48</td>
<td>180</td>
<td>830</td>
<td>3660</td>
<td>16344</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>13</td>
<td>42</td>
<td>163</td>
<td>737</td>
<td>3388</td>
<td>14730</td>
</tr>
<tr>
<td>random</td>
<td>1</td>
<td>19</td>
<td>78</td>
<td>321</td>
<td>1423</td>
<td>5846</td>
<td>23787</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>43</td>
<td>165</td>
<td>722</td>
<td>2988</td>
<td>11964</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>30</td>
<td>112</td>
<td>488</td>
<td>2006</td>
<td>8032</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>24</td>
<td>83</td>
<td>363</td>
<td>1497</td>
<td>6091</td>
</tr>
<tr>
<td>music</td>
<td>1</td>
<td>18</td>
<td>78</td>
<td>319</td>
<td>1422</td>
<td>5833</td>
<td>23757</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>42</td>
<td>194</td>
<td>722</td>
<td>3039</td>
<td>11900</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>30</td>
<td>111</td>
<td>488</td>
<td>2011</td>
<td>8066</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>9</td>
<td>24</td>
<td>84</td>
<td>358</td>
<td>1481</td>
<td>6040</td>
</tr>
<tr>
<td>ct</td>
<td>1</td>
<td>17</td>
<td>71</td>
<td>305</td>
<td>1421</td>
<td>5744</td>
<td>23502</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10</td>
<td>39</td>
<td>159</td>
<td>704</td>
<td>2943</td>
<td>11839</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>14</td>
<td>29</td>
<td>108</td>
<td>475</td>
<td>1978</td>
<td>8172</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>23</td>
<td>81</td>
<td>357</td>
<td>1639</td>
<td>6008</td>
</tr>
</tbody>
</table>

Table A.14: The efficiency calculated from table A.13
### Table A.15: Performance test of the implementation of Tarjan's disjoint set algorithm.
The backward version is used with a mpart of 4.

<table>
<thead>
<tr>
<th>image</th>
<th>#P</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>1</td>
<td>20</td>
<td>80</td>
<td>327</td>
<td>1446</td>
<td>5887</td>
<td>24077</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>43</td>
<td>167</td>
<td>843</td>
<td>3050</td>
<td>12087</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>30</td>
<td>111</td>
<td>485</td>
<td>2031</td>
<td>8290</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>23</td>
<td>86</td>
<td>361</td>
<td>1513</td>
<td>6263</td>
</tr>
<tr>
<td>vertical</td>
<td>1</td>
<td>16</td>
<td>71</td>
<td>288</td>
<td>1243</td>
<td>5163</td>
<td>21481</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>46</td>
<td>183</td>
<td>820</td>
<td>3689</td>
<td>15195</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>11</td>
<td>34</td>
<td>125</td>
<td>557</td>
<td>2493</td>
<td>10128</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>9</td>
<td>27</td>
<td>104</td>
<td>419</td>
<td>1836</td>
<td>7661</td>
</tr>
<tr>
<td>horizontal</td>
<td>1</td>
<td>17</td>
<td>69</td>
<td>275</td>
<td>1280</td>
<td>5207</td>
<td>21001</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>9</td>
<td>36</td>
<td>140</td>
<td>645</td>
<td>2678</td>
<td>10565</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>25</td>
<td>95</td>
<td>429</td>
<td>1813</td>
<td>7083</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>20</td>
<td>72</td>
<td>323</td>
<td>1324</td>
<td>5316</td>
</tr>
<tr>
<td>comb</td>
<td>1</td>
<td>18</td>
<td>72</td>
<td>295</td>
<td>1299</td>
<td>5340</td>
<td>21419</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14</td>
<td>54</td>
<td>211</td>
<td>943</td>
<td>4277</td>
<td>17595</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>14</td>
<td>45</td>
<td>168</td>
<td>777</td>
<td>3522</td>
<td>14771</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>12</td>
<td>37</td>
<td>142</td>
<td>642</td>
<td>2901</td>
<td>12669</td>
</tr>
<tr>
<td>random</td>
<td>1</td>
<td>19</td>
<td>78</td>
<td>318</td>
<td>1421</td>
<td>6250</td>
<td>23589</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>42</td>
<td>166</td>
<td>718</td>
<td>3050</td>
<td>12255</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>30</td>
<td>111</td>
<td>483</td>
<td>1999</td>
<td>8060</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>24</td>
<td>87</td>
<td>361</td>
<td>1475</td>
<td>6043</td>
</tr>
<tr>
<td>music</td>
<td>1</td>
<td>18</td>
<td>78</td>
<td>320</td>
<td>1432</td>
<td>5854</td>
<td>23502</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>42</td>
<td>166</td>
<td>723</td>
<td>3014</td>
<td>11964</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>30</td>
<td>113</td>
<td>483</td>
<td>1999</td>
<td>8060</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>23</td>
<td>83</td>
<td>363</td>
<td>1459</td>
<td>5985</td>
</tr>
<tr>
<td>ct</td>
<td>1</td>
<td>16</td>
<td>72</td>
<td>307</td>
<td>1396</td>
<td>5704</td>
<td>23653</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10</td>
<td>39</td>
<td>158</td>
<td>701</td>
<td>2959</td>
<td>12010</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
<td>29</td>
<td>108</td>
<td>472</td>
<td>1978</td>
<td>8004</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>22</td>
<td>82</td>
<td>356</td>
<td>1454</td>
<td>6196</td>
</tr>
</tbody>
</table>

### Table A.16: The efficiency calculated from table A.15
### Table A.17: Performance test of the implementation of Tarjan's disjoint set algorithm. The backward version is used with a `mpart` of 16.

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

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

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

<table>
<thead>
<tr>
<th>image</th>
<th>#P</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>1</td>
<td>19</td>
<td>60</td>
<td>330</td>
<td>1432</td>
<td>5920</td>
<td>23736</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>43</td>
<td>163</td>
<td>739</td>
<td>3019</td>
<td>12024</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>10</td>
<td>30</td>
<td>113</td>
<td>497</td>
<td>2062</td>
<td>8107</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>23</td>
<td>85</td>
<td>369</td>
<td>1482</td>
<td>6054</td>
</tr>
<tr>
<td>vertical</td>
<td>1</td>
<td>18</td>
<td>70</td>
<td>286</td>
<td>1254</td>
<td>5144</td>
<td>20737</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>16</td>
<td>51</td>
<td>189</td>
<td>814</td>
<td>3689</td>
<td>14928</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>24</td>
<td>45</td>
<td>141</td>
<td>655</td>
<td>2592</td>
<td>12272</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>25</td>
<td>41</td>
<td>111</td>
<td>478</td>
<td>1988</td>
<td>9175</td>
</tr>
<tr>
<td>horizontal</td>
<td>1</td>
<td>68</td>
<td>275</td>
<td>1310</td>
<td>5140</td>
<td>20985</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10</td>
<td>36</td>
<td>141</td>
<td>647</td>
<td>2602</td>
<td>10711</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>7</td>
<td>26</td>
<td>94</td>
<td>447</td>
<td>1784</td>
<td>7075</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>20</td>
<td>72</td>
<td>377</td>
<td>1323</td>
<td>5550</td>
</tr>
<tr>
<td>comb</td>
<td>1</td>
<td>18</td>
<td>73</td>
<td>296</td>
<td>1292</td>
<td>5295</td>
<td>21512</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>21</td>
<td>58</td>
<td>216</td>
<td>955</td>
<td>4305</td>
<td>17614</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>26</td>
<td>56</td>
<td>170</td>
<td>736</td>
<td>3321</td>
<td>13718</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>33</td>
<td>52</td>
<td>144</td>
<td>596</td>
<td>2721</td>
<td>11470</td>
</tr>
<tr>
<td>random</td>
<td>1</td>
<td>19</td>
<td>78</td>
<td>319</td>
<td>1432</td>
<td>5838</td>
<td>23752</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>43</td>
<td>166</td>
<td>715</td>
<td>3010</td>
<td>11978</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>12</td>
<td>33</td>
<td>112</td>
<td>489</td>
<td>2029</td>
<td>8099</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>9</td>
<td>28</td>
<td>85</td>
<td>363</td>
<td>1482</td>
<td>6040</td>
</tr>
<tr>
<td>music</td>
<td>1</td>
<td>18</td>
<td>79</td>
<td>319</td>
<td>1423</td>
<td>5832</td>
<td>23792</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>42</td>
<td>166</td>
<td>717</td>
<td>3016</td>
<td>12465</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>11</td>
<td>33</td>
<td>118</td>
<td>484</td>
<td>2011</td>
<td>8022</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>25</td>
<td>83</td>
<td>363</td>
<td>1472</td>
<td>5987</td>
</tr>
<tr>
<td>ct</td>
<td>1</td>
<td>17</td>
<td>71</td>
<td>306</td>
<td>1382</td>
<td>5750</td>
<td>23759</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>39</td>
<td>156</td>
<td>710</td>
<td>2943</td>
<td>11966</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>11</td>
<td>31</td>
<td>111</td>
<td>479</td>
<td>2024</td>
<td>8118</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>9</td>
<td>24</td>
<td>81</td>
<td>354</td>
<td>1464</td>
<td>5965</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>image</th>
<th>#P</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>2</td>
<td>86.4%</td>
<td>93.0%</td>
<td>101.2%</td>
<td>96.9%</td>
<td>98.0%</td>
<td>98.7%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>63.3%</td>
<td>88.9%</td>
<td>97.3%</td>
<td>96.0%</td>
<td>95.7%</td>
<td>97.6%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>67.9%</td>
<td>87.0%</td>
<td>97.1%</td>
<td>97.0%</td>
<td>99.9%</td>
<td>98.0%</td>
</tr>
<tr>
<td>vertical</td>
<td>2</td>
<td>47.4%</td>
<td>66.8%</td>
<td>75.7%</td>
<td>77.0%</td>
<td>69.7%</td>
<td>68.5%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>25.0%</td>
<td>51.9%</td>
<td>67.6%</td>
<td>63.8%</td>
<td>57.0%</td>
<td>56.3%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>18.0%</td>
<td>42.7%</td>
<td>64.4%</td>
<td>65.6%</td>
<td>58.5%</td>
<td>56.5%</td>
</tr>
<tr>
<td>horizontal</td>
<td>2</td>
<td>80.0%</td>
<td>94.4%</td>
<td>97.5%</td>
<td>101.2%</td>
<td>98.6%</td>
<td>98.0%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>76.2%</td>
<td>87.2%</td>
<td>97.5%</td>
<td>97.7%</td>
<td>96.0%</td>
<td>98.9%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>66.7%</td>
<td>85.0%</td>
<td>95.5%</td>
<td>86.9%</td>
<td>97.1%</td>
<td>98.1%</td>
</tr>
<tr>
<td>comb</td>
<td>2</td>
<td>42.9%</td>
<td>62.9%</td>
<td>68.5%</td>
<td>67.6%</td>
<td>61.5%</td>
<td>61.1%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>23.1%</td>
<td>43.5%</td>
<td>58.0%</td>
<td>58.6%</td>
<td>53.1%</td>
<td>52.3%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>13.6%</td>
<td>35.1%</td>
<td>51.4%</td>
<td>54.3%</td>
<td>48.6%</td>
<td>46.9%</td>
</tr>
<tr>
<td>random</td>
<td>2</td>
<td>86.4%</td>
<td>90.7%</td>
<td>96.1%</td>
<td>100.1%</td>
<td>97.0%</td>
<td>99.1%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>52.8%</td>
<td>78.8%</td>
<td>94.9%</td>
<td>97.6%</td>
<td>96.9%</td>
<td>97.8%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>52.8%</td>
<td>69.6%</td>
<td>93.6%</td>
<td>98.6%</td>
<td>98.5%</td>
<td>92.7%</td>
</tr>
<tr>
<td>music</td>
<td>2</td>
<td>81.6%</td>
<td>94.0%</td>
<td>96.1%</td>
<td>99.2%</td>
<td>97.0%</td>
<td>95.4%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>54.5%</td>
<td>79.8%</td>
<td>90.1%</td>
<td>98.0%</td>
<td>97.0%</td>
<td>98.9%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>56.2%</td>
<td>79.0%</td>
<td>96.1%</td>
<td>98.0%</td>
<td>99.4%</td>
<td>99.3%</td>
</tr>
<tr>
<td>ct</td>
<td>2</td>
<td>77.3%</td>
<td>91.0%</td>
<td>98.1%</td>
<td>97.3%</td>
<td>97.7%</td>
<td>99.3%</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>51.5%</td>
<td>76.3%</td>
<td>91.9%</td>
<td>96.2%</td>
<td>94.7%</td>
<td>97.5%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>47.2%</td>
<td>74.0%</td>
<td>94.4%</td>
<td>97.5%</td>
<td>98.2%</td>
<td>99.5%</td>
</tr>
</tbody>
</table>

Table A.20: The efficiency calculated from table A.19
Table A.21: Performance test of the implementation of Tarjan's disjoint set algorithm. The backward version is used with a mpart of 256.

<table>
<thead>
<tr>
<th>image</th>
<th>#P</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>1</td>
<td>20</td>
<td>80</td>
<td>326</td>
<td>1444</td>
<td>5913</td>
<td>23673</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>11</td>
<td>43</td>
<td>167</td>
<td>703</td>
<td>3042</td>
<td>12188</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>30</td>
<td>113</td>
<td>487</td>
<td>2026</td>
<td>8086</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>23</td>
<td>85</td>
<td>362</td>
<td>1491</td>
<td>6291</td>
</tr>
<tr>
<td>vertical</td>
<td>1</td>
<td>18</td>
<td>70</td>
<td>285</td>
<td>1252</td>
<td>5251</td>
<td>20861</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>30</td>
<td>83</td>
<td>211</td>
<td>844</td>
<td>3733</td>
<td>14953</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>33</td>
<td>100</td>
<td>192</td>
<td>720</td>
<td>3157</td>
<td>13003</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>49</td>
<td>111</td>
<td>173</td>
<td>555</td>
<td>2321</td>
<td>9464</td>
</tr>
<tr>
<td>horizontal</td>
<td>1</td>
<td>16</td>
<td>67</td>
<td>274</td>
<td>1285</td>
<td>5304</td>
<td>20993</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>9</td>
<td>37</td>
<td>142</td>
<td>647</td>
<td>2648</td>
<td>10665</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
<td>26</td>
<td>95</td>
<td>430</td>
<td>1791</td>
<td>7157</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>20</td>
<td>72</td>
<td>324</td>
<td>1299</td>
<td>5368</td>
</tr>
<tr>
<td>comb</td>
<td>1</td>
<td>17</td>
<td>72</td>
<td>297</td>
<td>1301</td>
<td>5361</td>
<td>21624</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>35</td>
<td>82</td>
<td>239</td>
<td>970</td>
<td>4226</td>
<td>17720</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>39</td>
<td>108</td>
<td>243</td>
<td>811</td>
<td>3552</td>
<td>14446</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>89</td>
<td>130</td>
<td>220</td>
<td>683</td>
<td>2873</td>
<td>12282</td>
</tr>
<tr>
<td>random</td>
<td>1</td>
<td>18</td>
<td>78</td>
<td>324</td>
<td>1417</td>
<td>5823</td>
<td>23783</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>44</td>
<td>165</td>
<td>720</td>
<td>2995</td>
<td>12015</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>11</td>
<td>33</td>
<td>117</td>
<td>483</td>
<td>2022</td>
<td>8067</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>11</td>
<td>28</td>
<td>86</td>
<td>365</td>
<td>1490</td>
<td>6026</td>
</tr>
<tr>
<td>music</td>
<td>1</td>
<td>19</td>
<td>78</td>
<td>319</td>
<td>1435</td>
<td>5868</td>
<td>23493</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>43</td>
<td>164</td>
<td>723</td>
<td>3023</td>
<td>12042</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>12</td>
<td>36</td>
<td>114</td>
<td>484</td>
<td>2036</td>
<td>7933</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>27</td>
<td>85</td>
<td>360</td>
<td>1480</td>
<td>6051</td>
</tr>
<tr>
<td>ct</td>
<td>1</td>
<td>17</td>
<td>71</td>
<td>308</td>
<td>1384</td>
<td>5736</td>
<td>23508</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>44</td>
<td>157</td>
<td>699</td>
<td>2978</td>
<td>11851</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>14</td>
<td>42</td>
<td>118</td>
<td>479</td>
<td>1993</td>
<td>7899</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>12</td>
<td>27</td>
<td>85</td>
<td>361</td>
<td>1480</td>
<td>6076</td>
</tr>
</tbody>
</table>

Table A.22: The efficiency calculated from table A.21.
Table A.23: Performance test of the implementation of Tarjan’s disjoint set algorithm.
The backward version is used with a *m part of ∞.

<table>
<thead>
<tr>
<th>image</th>
<th>#P</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>1</td>
<td>19</td>
<td>80</td>
<td>330</td>
<td>1450</td>
<td>6013</td>
<td>24010</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>44</td>
<td>170</td>
<td>730</td>
<td>3080</td>
<td>12035</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>9</td>
<td>31</td>
<td>113</td>
<td>488</td>
<td>2042</td>
<td>8165</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>7</td>
<td>24</td>
<td>86</td>
<td>367</td>
<td>1681</td>
<td>6640</td>
</tr>
<tr>
<td>vertical</td>
<td>1</td>
<td>17</td>
<td>78</td>
<td>286</td>
<td>1318</td>
<td>5254</td>
<td>21012</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>29</td>
<td>79</td>
<td>258</td>
<td>958</td>
<td>3984</td>
<td>15630</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>43</td>
<td>101</td>
<td>276</td>
<td>913</td>
<td>3681</td>
<td>14075</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>47</td>
<td>105</td>
<td>283</td>
<td>799</td>
<td>2862</td>
<td>11283</td>
</tr>
<tr>
<td>horizontal</td>
<td>1</td>
<td>16</td>
<td>67</td>
<td>272</td>
<td>1246</td>
<td>5104</td>
<td>20751</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>10</td>
<td>36</td>
<td>153</td>
<td>656</td>
<td>2663</td>
<td>10553</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>8</td>
<td>26</td>
<td>96</td>
<td>436</td>
<td>1786</td>
<td>7074</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>6</td>
<td>21</td>
<td>73</td>
<td>316</td>
<td>1333</td>
<td>5219</td>
</tr>
<tr>
<td>comb</td>
<td>1</td>
<td>18</td>
<td>75</td>
<td>306</td>
<td>1340</td>
<td>5356</td>
<td>21968</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>32</td>
<td>82</td>
<td>280</td>
<td>1053</td>
<td>4570</td>
<td>18886</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>40</td>
<td>99</td>
<td>294</td>
<td>1005</td>
<td>4081</td>
<td>15980</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>51</td>
<td>124</td>
<td>315</td>
<td>883</td>
<td>3508</td>
<td>13982</td>
</tr>
<tr>
<td>random</td>
<td>1</td>
<td>19</td>
<td>78</td>
<td>320</td>
<td>1417</td>
<td>5917</td>
<td>23865</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>43</td>
<td>168</td>
<td>723</td>
<td>3047</td>
<td>12241</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>11</td>
<td>33</td>
<td>123</td>
<td>486</td>
<td>2055</td>
<td>8251</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>10</td>
<td>29</td>
<td>90</td>
<td>369</td>
<td>1507</td>
<td>6072</td>
</tr>
<tr>
<td>music</td>
<td>1</td>
<td>19</td>
<td>80</td>
<td>331</td>
<td>1449</td>
<td>5962</td>
<td>23825</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>13</td>
<td>43</td>
<td>168</td>
<td>725</td>
<td>3029</td>
<td>12070</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>12</td>
<td>34</td>
<td>117</td>
<td>487</td>
<td>2058</td>
<td>8100</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>8</td>
<td>29</td>
<td>88</td>
<td>369</td>
<td>1500</td>
<td>6063</td>
</tr>
<tr>
<td>ct</td>
<td>1</td>
<td>17</td>
<td>73</td>
<td>305</td>
<td>1403</td>
<td>5861</td>
<td>23830</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>12</td>
<td>42</td>
<td>163</td>
<td>724</td>
<td>3023</td>
<td>12001</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>15</td>
<td>42</td>
<td>132</td>
<td>513</td>
<td>2035</td>
<td>8161</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>11</td>
<td>32</td>
<td>91</td>
<td>443</td>
<td>1456</td>
<td>6041</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Efficiency per processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>image</td>
</tr>
<tr>
<td>empty</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>vertical</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>horizontal</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>comb</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>random</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>music</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>ct</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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