Concurrent determination of connected components

Hesselink, Wim H.; Meijster, Arnold; Bron, Coenraad

Published in:
Science of computer programming

Science of Computer Programming 41 (2001) 173–194

Science of Computer Programming

www.elsevier.com/locate/scico

Concurrent determination of connected components

Wim H. Hesselink *,[1], Arnold Meijster [2], Coenraad Bron

Department of Mathematics and Computing Science, University of Groningen, P.O. Box 800, 9700 AV Groningen, Netherlands

Received 4 September 1998; received in revised form 5 August 2000; accepted 5 August 2000

Abstract

The design is described of a parallel version of Tarjan's algorithm for the determination of equivalence classes in graphs that represent images. Distribution of the vertices of the graph over a number of processes leads to a message passing algorithm. The algorithm is mapped to a shared-memory architecture by means of POSIX threads. It is applied to the determination of connected components in image processing. Experiments show a satisfactory speedup for sufficiently large images. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Connected components; Parallel algorithm; Pthreads; Mutex; Condition variable

1. Introduction

In many image processing applications, one of the first steps is to compute the connected components of the image. For this purpose one usually takes the simple breadth first scanning algorithm, which stems from the corresponding problem in graph theory. This algorithm has the disadvantages that it requires a FIFO-queue the size of which is a priori unknown, and that it is hard to parallelize. The number of pixels involved is often large, say more than a million, and for real-time applications often several images must be processed per second. It is therefore important to have an efficient parallel algorithm for this task. This is confirmed by the fact that there are many articles on parallel image component labelling. Most of these articles aim at distributed memory architectures, e.g., cf. [2,7,9].

Two classical sequential algorithms that explicitly use the fact that the graph is an image, are given in [11,13]. The main drawback of these algorithms is the use of a large equivalence table. Inspired by these two algorithms and Tarjan's disjoint set algorithm [15], we here present an algorithm that does not need such a large table, and

* Corresponding author.

E-mail addresses: wim@cs.rug.nl (W.H. Hesselink), a.meijster@rc.rug.nl (A. Meijster), cb@cs.rug.nl (C. Bron).

[1] http://www.cs.rug.nl/~wim

[2] http://www.rug.nl/hpc/people/arnold

PII: S0167-6423(01)00007-7

that can elegantly be parallelized. The sequential algorithm on itself is not new [8], but as far as we know there does not exist a parallel implementation of this algorithm, which is the main focus of this paper. The algorithm can be implemented on distributed as well as shared memory machines.

The algorithm determines a directed spanning forest for an undirected graph by placing links that are not necessarily along the edges of the graph. It is meant for large graphs, the nodes of which are distributed over a relatively small number of processes, preferably in such a way that most of the edges belong to only one process. In this respect, the situation differs from settings as investigated in [12,16], where the processes are in one-to-one correspondence with the nodes. Indeed, a typical setting for our algorithm could be a medical application used by medical specialists to analyse 3-D CT-images of a brain. In that case, the graph may have in the order of 10^8 points and the computation can be distributed over, say, four up to 16 processors.

Although we are especially interested in the application to images, we present the algorithm for general undirected graphs. The design goes through four stages. We first give a version of Tarjan's sequential algorithm, then distribute this over several processes with message passing. This design is then mapped to a shared memory architecture by means of mutual exclusion and synchronization. Finally, the mutual exclusion and synchronization are implemented by means of POSIX thread primitives.

The resulting algorithm is a concurrent one in which the amount of communication is decided at runtime. Such algorithms are very error prone. Our presentation may seem to focus on logic, but that is not the case. Since we want a working algorithm, our focus is on correctness, i.e., preservation of invariants, avoidance of deadlock, and guarantee of progress. Logical formulae are the only way to unambiguously express the properties needed.

Since we want to avoid unnecessary communication, we use no path compression beyond the parts of the graph under control of a single process. If the vertices of the graph are distributed randomly over the processes, this leads to bad worst case performance (i.e. quadratic in the lengths of the paths). In practice, however, there is often a natural way to distribute the nodes over the processes such that most edges adjacent to a node belong to only one process. In that case, the performance of the algorithm is quite good.

We finally describe the application to the determination of connected components in images. Since images are usually more or less constant locally, we sketch an optimization that can reduce the number of communications needed significantly. The results show that the algorithm makes distribution quite effective.

Overview: In Section 2 we give the abstract problem and develop a sequential solution. In Section 3, the algorithm is distributed over several processes in an asynchronous way. In Section 4, we specialize to a shared memory architecture in bounded space with atomicity brackets and **await** statements. In Section 5, these constructs are implemented by means of POSIX thread primitives. Section 6 describes the finalization of the algorithm. Section 7 contains the application to image processing. We draw conclusions in Section 8.

2. The problem and a sequential solution

In the image processing context, points of an image are regarded as directly connected if they are neighbours and have (nearly) the same colour or grey value. The problem is to determine the connected components of the image. Since images contain many points, and since one may want to process many subsequent images in real time, there is reason to consider distributed solutions. Graph theory is the proper abstract setting for any discussion of connected components.

We therefore let the image be represented by an undirected graph. The aim is to determine its connected components by means of a distributed algorithm. Our first step is the design of a sequential algorithm, which is a variation of Tarjan's algorithm, cf. [15, Chap. 2; 14, 12.3].

Let (V, E) be an undirected graph. We regard E as a (symmetric binary) relation. The connected components of the graph are the equivalence classes of the reflexive transitive closure E^* of E. The idea is to represent the components by rooted trees by means of an array variable

$$\texttt{par} : \textbf{array } V \textbf{ of } V \ ,$$

which stands for "parent". We define function $root : V \to V$ by

$$root(n) \quad = \quad \textbf{if } \texttt{par}[n] = n \textbf{ then } n \textbf{ else } root(\texttt{par}[n]) \textbf{ fi} \ .$$

Since V is finite, function $root$ is well defined if and only if the directed graph induced by the arrows of par has no cycles of length >1. We want to establish the postcondition that function $root$ is well defined and satisfies

$$Q: \quad (\forall m, n :: (m, n) \in E^* \quad \equiv \quad root(m) = root(n)) \ .$$

In order to establish Q, we introduce the equivalence relation Con given by

$$(m, n) \in Con \quad \equiv \quad root(m) = root(n) \ .$$

Now Q is equivalent to $E^* = Con$.

We assume that the initialization establishes $\texttt{par}[n] = n$ for all $n \in V$. Then function $root$ is well defined and relation Con is equal to the identity. We shall modify array par in such a way that function $root$ remains well defined and that relation Con is only extended. We therefore only modify par by assignments of the form $\texttt{par}[x] := y$ under one of the preconditions

$$P0(x, y): \quad (\exists k : k \geqslant 1 : \texttt{par}^k[x] = y) \ ,$$
$$P1(x, y): \quad \texttt{par}[x] = x \wedge root(y) \neq x \ .$$

Here, $\texttt{par}^k[x]$ is obtained by k subsequent applications of par on index x. In the case of $P0(x, y)$, node y is an ancestor of x and the assignment $\texttt{par}[x] := y$ does not modify relation Con. Such an assignment is called *path compression*, cf. [1]. In case of $P1(x,y)$, node x is a root and not an ancestor of y. Since y becomes the parent of x, relation *Con* is strictly extended.

We now come to the edge relation E of the graph. Since we do not want to store every unordered pair twice, we assume that relation E is represented by a set *edlis* of pairs of nodes via the initial relation $E = sym(edlis)$ where function *sym* is defined by

$$(m,n) \in sym(R) \quad \equiv \quad (m,n) \in R \vee (n,m) \in R \ .$$

We take *edlis* to be a program variable and introduce the loop invariant

$$J0: E^* = (Con \cup sym\ (edlis))^* \ .$$

Predicate $J0$ holds initially, since then $sym(edlis) = E$ and *Con* is the identity. If *edlis* is empty, $J0$ implies predicate Q since *Con* is an equivalence relation. We therefore take $edlis \neq empty$ as the guard of the loop.

Now the abstract sequential algorithm is

A: **while** $edlis \neq empty$ **do**
  fetch (u,v) *from edlis* ;
  Extend
od ,

where command *Extend* has to restore predicate $J0$ if it is falsified by the removal of (u,v) from *edlis*. Restoration can be done by placing a `par` pointer between the components of u and v. We therefore search for elements x, y, connected to u and v, that satisfy $P1(x,y)$. We thus introduce the predicate

$$JE: (u,x),(v,y) \in \ Con \quad \vee \quad (u,y),(v,x) \in Con$$

and describe *Extend* by

Extend: **if** $(u,v) \notin Con$ **then**
  choose x,y *with* $P1(x,y) \wedge JE$;
  $\mathtt{par}[x] := y$
fi .

It is easy to see that in this way $J0$ is preserved and that, consequently, algorithm A is correct. So it remains to implement *Extend*. Since relation *Con* is not directly available, we implement *Extend* by means of a loop with *JE* as invariant. Since *Con* is an equivalence relation, $JE \wedge x = y$ implies $(u,v) \in Con$. We can therefore refine *Extend* as follows.

Extend: $x := u$; $y := v$ $\{JE\}$;
while $x \neq y \wedge \neg P1(x,y)$ **do**
  modify x,y *while preserving JE*
od ;
if $x \neq y$ **then** $\mathtt{par}[x] := y$ **fi** .

For the ease of distributed verification of the inequality $root(y) \neq x$ in $P1(x, y)$, we assume that the set V has a total order $\leqslant$ and introduce the additional invariant (cf. [14, p. 261]):

$$J1: \quad \mathtt{par}[n] \leqslant n \ .$$

Here and henceforth, we use the convention that all invariants are universally quantified over the free variables they contain (here n).

We now decide that the loop in *Extend* preserves the invariant $JE \wedge x \geqslant y$. We therefore assume that the pairs in *edlis* are ordered by

$$J2: \quad (m,n) \in edlis \Rightarrow m > n \ .$$

In the body, we replace x by $\mathtt{par}[x]$ and, if necessary, restore $x \geqslant y$ by swapping. Now the guard of the loop can be simplified since $J1 \wedge x \geqslant y$ implies

$$P1(x, y) \quad \equiv \quad \mathtt{par}[x] = x \wedge x \neq y \ .$$

It follows that

$$x \neq y \wedge \neg P1(x, y) \quad \equiv \quad x \neq y \wedge \mathtt{par}[x] \neq x$$

and thus we obtain

Extend: $x := u$; $y := v$;
  while $x \neq y \wedge \mathtt{par}[x] \neq x$ **do**
    $x := \mathtt{par}[x]$;
    if $x < y$ **then** $x, y := y, x$ **fi**
  od ;
  if $x \neq y$ **then** $\mathtt{par}[x] := y$ **fi** .

Since V is finite, it is easy to see that the loop terminates.

The efficiency of algorithm A can be improved considerably by path compression, i.e., by extending the final **then** branch of *Extend* with assignments $\mathtt{par}[z] := y$ for all nodes z on the $\mathtt{par}$ paths of u and v. This optimization preserves all invariants. A simple version of it only adds $\mathtt{par}[u] := y$ and $\mathtt{par}[v] := y$. In our application this seems to be just as effective.

3. Distribution

In this section, we distribute algorithm A over a system of sequential processes that communicate by message passing. We use the following convention with respect to private variables. If x is a private variable of process p, we refer to it as x in the code and as $x.p$ if p is not obvious from the context. Let *Process* be the set of processes. We assume that the set V is distributed over the processes by means of a function $owner: V \to Process$. We assume that process p is allowed to inspect and modify $\mathtt{par}[x]$ if and only if $p = owner(x)$.

We assume that *edlis* is distributed over the processes as well. So, each process p has its own set $edlis(p)$ and we regard *edlis* as an alias for the union of the sets $edlis(p)$. We introduce the invariant

$$J3: \quad (m,n) \in edlis(q) \;\Rightarrow\; owner(m) = q \ .$$

Since the loop in *Extend* can only be executed by process p as long as $owner(x) = p$, we introduce the local search command

$$
\begin{array}{ll}
Search: & x := u \ ; \quad y := v \ ; \\
 & \textbf{while } owner(x) = self \wedge x \neq y \wedge \mathtt{par}[x] \neq x \textbf{ do} \\
 & \quad x := \mathtt{par}[x] \ ; \\
 & \quad \textbf{if } x < y \textbf{ then } x, y := y, x \textbf{ fi} \\
 & \textbf{od} \ ,
\end{array}
$$

where *self* stands for the executing process. Since the guards are evaluated from left to right, $\mathtt{par}[x]$ is not inspected if $owner(x) \neq self$. Execution of *Search* establishes the postcondition

$$owner(x) \neq self \vee x = y \vee \mathtt{par}[x] = x.$$

It is now clear that each process should repeatedly execute

$$
\begin{array}{l}
fetch \quad (u,v) \ \ from \quad edlis(self) \ ; \\
Search \ ; \\
\textbf{if } x \neq y \textbf{ then} \\
\quad \textbf{if } owner(x) = self \textbf{ then } \mathtt{par}[x] := y \\
\quad \textbf{else } put(x, y) \ into \ edlis(owner(x)) \textbf{ fi} \\
\textbf{fi} \ .
\end{array}
$$

This program fragment preserves $J0 \wedge J1 \wedge J2 \wedge J3$, i.e., indeed, $J0$, $J1$, $J2$, $J3$ are invariants. It terminates for the same reason as in the case of the sequential algorithm.

In this way, the sets $edlis(p)$ become buffers with one consumer and many producers. Process p fetches elements from $edlis(p)$ and other processes may put elements into it. These actions therefore require communication: the last line of this fragment can be read as “send (x, y) to the process that owns x”.

Since communication is expensive in performance, we partition the set $edlis(p)$ into two parts $edlis0(p)$ and $edlis1(p)$, and assume the invariant $edlis(p) = edlis0(p) \cup edlis1(p)$ with initially

$$
\begin{array}{l}
edlis0(p) = \{(u,v) \in edlis(p) \mid owner(v) = p\} \ , \\
edlis1(p) = \{(u,v) \in edlis(p) \mid owner(v) \neq p\} \ .
\end{array}
$$

We can therefore treat *edlis0*(p) in an initial program fragment A0, obtained from A by substituting *edlis0*(p) for *edlis*.

A0: **while** *edlis0*(*self*) $\neq$ *empty* **do**
  fetch (u,v) *from edlis0*(*self*) ;
  Extend
od .

Since initially $\mathtt{par}[z]=z$ for all nodes z, fragment A0 preserves the invariant that $owner(\mathtt{par}[z])=p$ for all z with $owner(z)=p$.

During the treatment of *edlis1*(p), process p must be able to put elements into *edlis1*(q) where q is some process with $q\neq p$. As a consequence, process p must not stop when its set *edlis1*(p) is empty since other processes may insert new elements in *edlis1*(p). We declare for each process a private variable *continue* to indicate that new pairs yet may arrive.

A1: **while** *continue* **do**
  fetch (u,v) *from edlis1*(*self*) ;
  Search ;
    if $x\neq y$ **then**
      if $owner(x)=self$ **then** $\mathtt{par}[x]:=y$
      else $put(x,y)$ *into edlis1*$(owner(x))$ **fi**
    fi
od .

The program for process p now becomes the composition of the two parts A0 and A1. Part A0 needs no further refinement. Part A1 primarily requires termination detection: how to give the boolean variables *continue* the adequate values?

We assume that process p starts up with initial values for *edlis0*(p) and *edlis1*(p). The size of the union of the sets *edlis1*(p) only shrinks. Every process can terminate when all sets *edlis*(p) are empty and each process has finished its local computation, but not earlier. To keep track of the edges that have yet to be treated, we attach a unique token t to each edge (u,v) in *edlis1*(p). This token serves to indicate the originator of the pair (u,v) for the sake of termination detection. It is sent unmodified with the changing edge (u,v) as a message $edge(u,v,t)$. When no triple is sent, the token t is destroyed.

Each token shall belong to the process that creates it. We represent the assignment of tokens to processes by a function $origin: Token\rightarrow Process$. Each process gets a private integer variable *ctok* to count its number of outstanding tokens. Whenever a token is destroyed, a message *down* is sent to its origin. A process decrements *ctok* when it receives a message *down*. We thus have the invariant that *ctok* of process q is the number of messages $edge(u,v,t)$ in transit with $origin(t)=q$ plus the number of *down* messages in transit to q. This can be expressed by

$$J4: \quad ctok.q = \#\{edge(u,v,t) \mid origin(t) = q\} + transit(down,q) \ ,$$

where we use $transit(m,q)$ to denote the number of messages m in transit to q, and $\#A$ to denote the number of elements of the set A.

We introduce a message *stop* to signal termination. Indeed, when all tokens of all processes have been destroyed, all buffers are empty and every process may terminate.

In order to decide that all tokens of all processes have been destroyed, we introduce a global counter gc for the number of processes that are initializing or have $ctok > 0$. We give one process, say *adm*, the additional task to administrate the value of gc, which initially equals $\#Process$. A process that reaches $ctok = 0$, sends a *gcdown* message to *adm*. We postulate the invariant that gc equals the number of processes q with $ctok.q > 0$ plus the number of *gcdown* messages in transit, i.e.

$$J5: \quad \mathtt{gc} = \#\{q \mid ctok.q > 0\} + transit(gcdown, adm) \ .$$

When process *adm* receives the message *gcdown* it decrements gc and, if gc becomes 0, it sends messages *stop* to all processes, as expressed in command *GcDown*:

GcDown: $\mathtt{gc} := \mathtt{gc} - 1$;
  if $\mathtt{gc} = 0$ **then**
    for all $q \in Process$ **do** **send** *stop* **to** q **od**
  fi .

A process that receives *stop*, sets *continue* to false. This leads to the invariants

$$J6: \quad continue.q \quad \equiv \quad \mathtt{gc} > 0 \vee transit(stop,q) > 0 \ ;$$

$$J7: \quad transit(stop,q) > 0 \Rightarrow \mathtt{gc} = 0 \ .$$

Fragment A1 is now replaced by

```
A1:   Init1 ;
      while continue do
      in  edge(u,v,t) →
              Search ;
              if x ≠ y  ∧  owner(x) ≠ self then
                 send edge(x,y,t) to owner(x)
              else
                 if x ≠ y then par[x] := y fi ;
                 send down to origin(t) ;
              fi
      []    down  →
              ctok := ctok −1 ;
```

```
            if ctok = 0 then send gcdown to adm fi
    []   gcdown → GcDown
    []   stop → continue := false
    ni
  od .
```

The auxiliary command *Init1* is treated below. The rest of A1 is a repetition that consists of reception and treatment of messages. For this purpose, we use a variation of the **in...ni** construct of the language SR of [4]. It involves waiting for the next message to arrive, the choice according to the arriving message, and it introduces formal parameters for the arguments of the message, if any. Note that this code implies that a process may send asynchronous messages to itself. Such messages can easily be eliminated. We have not done so for the sake of uniformity.

After the treatment of $edge(u,v,t)$, the process may perform path compression along the two paths it has investigated in its own part of the graph. In view of the communication overhead, we decided not to consider more extensive forms of path compression.

For the sake of uniformity, the initialization of A1 translates the edges in *edlis1* into *edge* messages from the process to itself. A1 is thus initialized by

```
Init1:   ctok := 0 ;
         continue := true ;
         for all (x, y) ∈ edlis1(self) do
             create a token t with origin(t) = self ;
             ctok := ctok + 1 ;
             send edge(x, y, t) to self
         od ;
         if ctok = 0 then send gcdown to adm fi .
```

In order to verify the invariants, we first need to describe the execution model. Processes are concurrently allowed to receive and execute messages. Since the effect of execution of a message only depends on the message and the precondition of the accepting process, we may (for the sake of the correctness proof) assume that the messages are accepted by the processes in some linear order and that a message is accepted only when the command associated to the previous message has been executed completely by the previous accepting process. In other words, in our model, the acceptance of a message includes atomic execution of the associated command. The invariants are predicates that are supposed to hold before and after each complete acceptance of a message.

It is now straightforward to verify the invariants $J4$, $J5$, $J6$, and $J7$. Indeed, each of these predicates holds when all processes have completed *Init1*. Acceptance of a message *edge* leads to re-sending of a message *edge* or *down*. Therefore, $J4$ is preserved. Acceptance of *down* by process p preserves $J4$ since $ctok.p$ is decremented. It also preserves $J5$, since *gcdown* is sent if $ctok.p$ reaches 0. Acceptance of *gcdown* by process *adm* preserves $J5$ since gc is decremented. It also preserves $J6$ and $J7$ since *stop* is sent if and only if gc reaches 0. Acceptance of *stop* by process p preserves $J6$ since *continue.p* is set to false and $J7$ implies $\mathtt{gc} = 0$.

It follows from $J4 \wedge J5$ that, while there are edges to be processed, we have $\mathtt{gc} > 0$, so that $J6$ implies that all processes have not yet terminated. On the other hand, when there are no messages in transit, then $J4 \wedge J5$ implies that $\mathtt{gc} = 0$, so that $J6$ implies $\neg\, continue.q$ for all processes q. So, then, all processes have terminated.

4. Bounded shared memory

We now assume that the processes communicate by means of shared memory, and that the size of this memory is bounded. We use the convention that shared variables are in typewriter font. In this section we specify the requirement on atomicity and synchronization by means of atomicity brackets and **await** statements. The next section is devoted to the implementation of these constructs by means of the POSIX thread primitives.

We eliminate the messages *edge*, *down* and *stop*, and replace them by procedures *PutEdge*, *Down*, and *Stop*. The edge triples that are to be communicated between the processes will be placed somewhere in the shared memory. Each process is equipped with a private list of such triples and has a private variable *head0* that serves as the head of this list. The private list is empty iff $head0 = nil$. Procedure *GetEdge* fetches a triple from the private list.

We introduce a procedure *AwaitEdge*, the postcondition of which implies that $head0 \neq nil$ or *Stop* has been called. Then program fragment A1 is replaced by

```
A2:   Init2 () ;
      loop
         AwaitEdge () ;
         if head0 = nil then exitloop fi ;
         GetEdge (u, v, t) ;
         Search ;
         if x ≠ y  ∧  owner(x) ≠ self then
            PutEdge (owner(x), x, y, t)
         else
            if x ≠ y then par[x] := y fi ;
            Down (origin(t))
         fi
      endloop .
```

In order to replace the messages *down* by a procedure *Down*, we replace the private variables *ctok* by a shared variable

ctok: **array** *Process* **of** *Integer* , and we define

procedure *Down* $(q : Process) =$
  var $b : Boolean$;
  $\langle$ $\texttt{ctok}[q] := \texttt{ctok}[q] - 1$; $b := (\texttt{ctok}[q] = 0)$ $\rangle$;
  if b **then** *GcDown* () **fi**
end .

Here, atomicity brackets $\langle\ \rangle$ are used to specify that the command enclosed by them shall be executed without interference. Now *GcDown* is a procedure given by

procedure *GcDown* ()=
  var $b : Boolean$;
  $\langle$ $\texttt{gc} := \texttt{gc} - 1$; $b := (\texttt{gc} = 0)$ $\rangle$;
  if b **then** *Stop* () **fi**
end .

We replace the private variables *continue* by a shared array

`cntu`: **array** *Process* **of** *Boolean* ;

with initially $\texttt{cntu}[q] = true$ for all processes q. We then define procedure *Stop* by

procedure *Stop* () =
  for all $q \in Process$ **do** $\langle$ $\texttt{cntu}[q] := false$ $\rangle$ **od**
end .

Note that, in this way, the special process *adm* is eliminated.

Remark. One could of course replace the array `cntu` by a single boolean variable. This would cause memory contention, however, when many processes try to access it concurrently. We therefore prefer to use an array.

We finally come to the central problem of a shared data structure where the processes can deposit the edges destined for other processes. For this purpose, we assume that there is a constant M such that $\#edlis1(p) \leqslant M$ for all processes p. Let N be the number of processes. It then follows that we need at most $N * M$ tokens. We thus define the type $Token = [0\ ..\ N * M - 1]$ and use this type as the index set for the messages. We decide to store the triple (x, y, t) always at index t by means of the shared variable

`pair`: **array** *Token* **of** $V \times V$.

The message buffers are constructed as lists of pairs. For this purpose, we introduce a value $nil \notin Token$ to designate the empty list and declare the shared variables

`next`: **array** *Token* **of** $Token \cup \{nil\}$;

`head`: **array** *Process* **of** $Token \cup \{nil\}$;

with initially $\texttt{head}[q] = nil$ for all q. We use $\texttt{head}[q]$ as the head of the list for process q where other processes can write. Now procedure *PutEdge* is given by

```
procedure PutEdge (q : Process; x, y : V; t : Token) =
  pair[t] := (x, y) ;
  ⟨ next[t] := head[q] ;
    head[q] := t   ⟩
end .
```

Reading and writing of $\texttt{head}[q]$ must be done under mutual exclusion. The assignment to $\texttt{pair}[t]$ is not threatened by interference, however, since we preserve the invariant that there is always at most one process that holds token t.

Recall that every process also has a private variable *head0* as the head of a private list of tokens. A process fetches an element from its private list by the simple procedure

```
procedure GetEdge (var x, y : V; var t : Token) =
  t := head0 ;
  head0 := next[t] ;
  (x, y) := pair[t]
end .
```

In procedure *AwaitEdge*, the two lists of a process are swapped whenever the private list is empty and the public one is not:

```
procedure AwaitEdge () =
  if head0 = nil then
    ⟨ await head[self] ≠ nil ∨ ¬ cntu[self] then
        head0 := head[self] ;
        head[self] := nil   ⟩
  fi
end .
```

Here we use an atomic **await** statement as described in (e.g.) [3,5]. Note that, indeed, *AwaitEdge* has the postcondition that $head0 \neq nil$ if $\texttt{cntu}[self]$ holds.

We assume that processes are numbered from $Process = [\,0\ ..\ N-1\,]$. We distribute the tokens according to the rule that process p gets the tokens t with $p*M \leqslant t < (p+1)*M$. It follows that function *origin* is given by $origin(t) = t\ \mathbf{div}\ M$. Then the initialization is given by

```
procedure Init2 () =
  var t := self * M ;
  head0 := nil ;
  for all (x, y) ∈ edlis1(self) do
     next[t] := head0 ;
     head0 := t;
     pair[t] := (x, y) ;
```

$t := t + 1$
od ;
ctok[*self*] $:= t - self * M$;
if ctok[*self*] $= 0$ **then** *GcDown* () **fi**
end .

Here the assignments to ctok are not threatened by interference with *Down*, since the tokens from process q are not yet available to other processes. The use of two lists for every process enables us to treat the initialization of the processes as a private activity.

5. Using mutexes and condition variables

In this section, we implement the atomicity brackets and the **await** statement introduced in the previous section by means of mutexes and condition variables as specified in the POSIX thread standard, cf. [6,10].

Mutexes serve to implement the atomicity brackets $\langle\rangle$. A mutex can be regarded as a record with a single field *owner* of type *Process*; $m.owner = \perp$ means that the mutex is free. The commands to lock and unlock a mutex m are given by

lock(m): $\langle$**await** $m.owner = \perp$ **then** $m.owner := self$ $\rangle$;
unlock(m): $\langle$**await** $m.owner = self$ **then** $m.owner := \perp$ $\rangle$.

The description of **unlock** is maybe slightly surprising: it enforces that only the owner of the lock is able to unlock it. A thread that tries to unlock a mutex it does not own, has to wait indefinitely. For every mutex m, we use the initialization $m.owner = \perp$. The commands **lock** and **unlock** are abbreviations of the POSIX primitives pthread_mutex_lock and pthread_mutex_unlock.

After this preparation we come back to the synchronization of the program of the previous section. In order to allow maximal concurrency, we introduce several mutexes and arrays of mutexes for the protection of specific atomic regions. We introduce a mutex mtok[q] to protect ctok[q] and a mutex mgc to protect gc. We thus declare

mtok: **array** *Process* **of** *Mutex* ;
mgc: *Mutex* .

The procedures *Down* and *GcDown* become

procedure *Down* (q : *Process*) =
var b : *Boolean* ;
lock (mtok[q]) ;
ctok[q] := ctok[q] $- 1$; $b :=$ (ctok[q] $= 0$) ;
unlock (mtok[q]) ;
if b **then** *GcDown* () **fi**
end .

procedure *GcDown* () =
  var b : *Boolean* ;
  lock (mgc) ;
  gc := gc − 1 ; b := (gc = 0) ;
  unlock (mgc) ;
  if b **then** *Stop* () **fi**
end .

We use condition variables for the implementation of the **await** construct in *AwaitEdge*. A variable v of type *Condition* is the name of a list $Q(v)$ of threads that are waiting for a signal. We only use the POSIX primitives `pthread_cond_wait` and `pthread_cond_signal`, abbreviated by **wait** and **signal**. These primitives are expressed by

wait (v, m) :
  $\langle$ **unlock** (m); *insert self in* $Q(v)$ $\rangle$;
  lock (m) .

Command **wait** consists of two atomic commands: to start waiting and to lock when released. Note that a thread must own the mutex to execute **wait**.

Command **signal** (v) is equivalent to *skip* if $Q(v)$ is empty. Otherwise, it releases at least one thread waiting at $Q(v)$. This is expressed in

signal (v):
  $\langle$**if not** *isEmpty* $(Q(v))$ **then** *release some threads from* $Q(v)$ **fi**$\rangle$.

Notice that, when some thread signals v and thus releases a waiting thread, the latter need not be able to (immediately) lock the mutex. Some other thread may obtain the mutex first.

Back to the program. We introduce a mutex gate[q] to protect head[q] and cntu[q] in the procedures *AwaitEdge*, *PutEdge*, and *Stop*. We introduce a condition variable cv[q] to signal process q that the condition it may be waiting for has been established. We thus declare

gate: **array** *Process* **of** *Mutex* ;
cv: **array** *Process* **of** *Condition* .

The procedures *PutEdge* and *Stop* are translations of their counterparts in Section 4 extended with signals to the possible waiting processes.

procedure *PutEdge* (q : *Process*; x, y : V; t : *Token*) =
  pair[t] := (x, y) ;
  lock (gate[q]) ;
  next[t] := head[q] ;
  head[q] := t ;

```
    signal (cv[q]) ;
    unlock (gate[q])
  end .

  procedure Stop () =
    for all q ∈ Process do
        lock (gate[q]) ;
        cntu[q] := false ;
        signal (cv[q]) ;
        unlock (gate[q])
    od
  end .
```

AwaitEdge is implemented in

```
procedure AwaitEdge () =
  if head0 = nil then
    lock (gate[self]) ;
    if head[self] = nil ∧ cntu[self] then
      wait(cv[self], gate[self])
    fi ;
    head0 := head[self] ;
    head[self] := nil ;
    unlock (gate[self])
  fi
end .
```

Note that, here, at most one process can be waiting at any condition variable. So, there is no danger that a signal releases more than one thread. On the other hand, since the waiting process is the only process that can invalidate it, the wait condition need not be tested again.

Remark. If one removes the **lock** and **unlock** in *Stop*, the program becomes incorrect, since then a process, say q, may observe that the guard in *AwaitEdge* holds true and another process may falsify `cntu`$[q]$ and signal `cv`$[q]$ before q starts waiting.

It is also possible to implement the **await** construct in *AwaitEdge* by means of a split binary semaphore, see e.g. [3].

6. Harvest

After execution of algorithm A or its shared memory version, the connected components of the graph are determined by the function *root*. We collect this result in a separate array

$\texttt{root}$: **array** V **of** V.

In view of invariant $J1$, a sequential algorithm to do this is

B: **for all** $n \in V$ **do in increasing order**
  if $\texttt{par}[n] = n$ **then**
    $\texttt{root}[n] := n$;
  else $\texttt{root}[n] := \texttt{root}[\texttt{par}[n]]$ **fi**
od .

In this way, the connected components of the graph are characterized by a unique representing element, the root of the par tree. Loop B is very efficient, of order $\mathcal{O}(\#V)$.

When the graph is very large, distributed harvesting may be indicated. To enable this, we decide that in harvest time all processes are allowed to inspect array par, but inspections and updates of $\texttt{root}[n]$ are only allowed for the owner of node n. We therefore write $V(p)$ to denote the set of nodes $n \in V$ with $owner(n) = p$ and we let the processes perform

C: **for all** $n \in V(self)$ **do in increasing order**
  if $\texttt{par}[n] \in V(self)$ **then**
    if $\texttt{par}[n] = n$ **then** $\texttt{root}[n] := n$
    else $\texttt{root}[n] := \texttt{root}[\texttt{par}[n]]$ **fi**
  else
    $r := \texttt{par}[n]$;
    while $r \neq \texttt{par}[r]$ **do** $r := \texttt{par}[r]$ **od** ;
    $\texttt{root}[n] := r$
  fi
od .

Fragment C has the inefficiency that root paths that leave $V(p)$ may be traversed repeatedly. We therefore introduce the following optimization. For each process, we declare a private variable *outList* of the type list of nodes with the invariant

$$J8: \quad x \in V(p) \ \land \ \texttt{par}[x] \notin V(p) \quad \Rightarrow \quad x \in outList.p \ .$$

We take *outList.p* to be empty initially. Predicate $J8$ is preserved by program fragment A0. In order to preserve $J8$ during A1 and A2, we now let the assignments $\texttt{par}[x] := y$ in A1 and A2 be accompanied by the instruction to add x to the private *outList*.

We now first set all values of root to some reserved value $\perp$ and then determine the roots of the elements of *outList*.

D: **for all** $n \in V(self)$ **do** $\texttt{root}[n] := \perp$ **od** ;
  for all $n \in outList$ **do**
    $r := n$;

```
        while r ≠ par[r] do r := par[r] od ;
        root[n] := r ;
      od .
```

After this loop, all points $x \in V(p)$ with $\mathtt{par}[x] \notin V(p)$ have the correct value for $\mathtt{root}[x]$, while the other points $x \in V(p)$ have $\mathtt{root}[x] := \perp$. These remaining points of $V(p)$ are treated in the following loop:

```
E:    for all n ∈ V(self) do in increasing order
        if  root[n] = ⊥ then
          if par[n] = n then root[n] := n
          else root[n] := root[par[n]] fi
        fi
      od .
```

Here, we use the invariant $J1$. Since the points are treated in increasing order, we have the invariant that **root**$[x]$ has the final value for all $x < n$. This property is preserved by the body of loop E because of $J1$. The composition D ; E is our final version of the harvest. This version is more efficient than version C, since only the root paths of points in *outList* are followed completely.

The list *outList* can be implemented most easily as a stack with maximal size equal to the number of boundary points of the set $V(p)$. Every element of $outList.p$ is an ancestor of a point of the boundary of $V(p)$, with all intermediate points within $V(p)$. This implies that $\#outList.p$ is bounded by the number of elements of the boundary of $V(p)$.

7. Application to image processing

In this section we focus on the application to image processing. We first consider a grey-scale image represented by a two-dimensional integer-valued array $im[H, W]$ (later we will consider three-dimensional 'images' as well), where H and W are the height and the width of the image, respectively. The first coordinate (x) denotes the row number (scan line), while the second coordinate (y) denotes the number of the column. Since grey-scale images are discretizations of real black-and-white photographs there is an implicit underlying grid. We consider the case of 4-connectivity, meaning that pixels (except for boundary pixels) have four neighbours (north, east, south, west). Two neighbouring pixels that have the same image value, are considered to be in the same connected component. So, the graph considered has the pixels as nodes, and two pixels are connected iff they are neighbours *and* have the same image value.

Since the graph under consideration is rectangular, we can distribute it over the N processes by splitting it in equally sized slices. We have decided to distribute the image on the last coordinate of a pixel. It follows that we split the image in (almost) equally sized vertical slices. The test $(x, y) \in V(p)$ becomes $lwb(p) \leqslant y < lwb(p+1)$, where *lwb* is given by

$$lwb(p) = (p \cdot W) \ \mathbf{div} \ N \ .$$

It follows that the corresponding function *owner* satisfies

$$owner(y) = ((N \cdot (y+1)) - 1) \ \mathbf{div} \ W \ .$$

The parallel algorithm consists of three phases. In the first phase, algorithm A0 is applied on the image slices. This is performed in a scan-line fashion, in which for pixel (x, y) only the pixels $(x-1, y)$ (north) and $(x, y-1)$ (west) need to be inspected.

In the middle phase, algorithm A2 is applied to edges of the graph which cross the boundaries of the distribution. In view of invariant $J2$, the list $edlis1(p)$ must contain the pixel pairs (x, y), $(x, y-1)$ with $y = lwb(p)$ for which $im[x, y] = im[x, y-1]$. It follows that H is an upper-bound for the length of the edge lists $edlis1(p)$. We can therefore take M of Section 4 to be equal to H.

Since we deal with images, a very effective optimization can be used to reduce the sizes of the buffers $edlis1(p)$. Indeed, we need not insert the pair (x, y), $(x, y-1)$ into $edlis1(p)$, if this list already contains the pair $(x-1, y)$, $(x-1, y-1)$ while also $im[x, y] = im[x-1, y]$. Indeed, if this is the case, the pair consists of pixels connected already, and we can therefore disregard the new edge. Experiments have shown that for camera made images this optimization often reduces the size of the buffers significantly. The optimization is used in the initialization *Init2*, while the remainder of A2 is left unmodified. In the final phase, we use the harvesting routine (D;E) to compute the output image.

The algorithm is easily adapted to 'images' of higher dimensionality. Apart from choosing another distribution, indexing, and the corresponding functions *lwb*, *owner* and *origin*, no modifications are necessary. We applied the algorithm to a three-dimensional CT-scan data set $im[D, H, W]$, where D is the number of 2-D image slices (depth) of the data set. We used the same functions *lwb* and *owner*. In this case, the bound M of the sizes of the lists *edlis1* is $D \cdot H$.

7.1. Practical results

We applied the shared memory version of the algorithm on a set of seven 2-D test images. We had the availability of two shared memory architectures, namely a Cray J90 vector computer consisting of 32 processors and 4 Gb shared memory, and a Compaq ES40 with 4 Alpha-processors and 1 Gb shared memory. The processors of the Cray J90 computer are shared with other users, and the scheduling of these is done by the operating system, without any control to the user. It is therefore almost impossible to acquire 32 processors without interference by other users. For this reason, we decided to do time measurements up to 16 processors, which turned out to be reasonably available. Each measurement was performed a 100 times, of which the 25 best and the 25 worst measurements were discarded. The remaining 50 measurements were averaged. This

Table 1
Absolute timings in milliseconds on a single CPU for different images sizes

Image	ES40			CRAY J90		
	256	512	1024	256	512	1024
empty	10	42	172	381	1558	6260
vline	7	30	123	283	1145	4610
hline	6	28	114	287	1163	4801
comb	7	30	123	279	1139	4615
squares	10	42	173	374	1539	6228
music	10	42	172	361	1488	6080
CT	9	42	181	297	1370	5808

Fig. 1. Test images: (a) squares (b) music (c) CT.

way we hope to get a reasonable measurement. On the Compaq ES40, measurements were performed simply 50 times and averaged immediately, since we were the only user on the system. The absolute timings on a single CPU are shown in Table 1. Note that the ES40 performs much better than the Cray. One may realize that the design of the Cray is more than 5 years older than that of the ES40, and that the Cray is a typical vector processor, which is not of any use in our algorithm. Besides, the Compaq has a cache memory on each processor of 512 kB, while the Cray has no cache whatsoever.

The image named *empty* is a trivial image of which all pixels have the same grey value. The image *vline* is an image for which pixels $im[x,y]=1$ if y is even, and $im[x,y]=0$ if y is odd. The image *hline* is the image *vline* rotated over 90 degrees. The image *comb* is similar to the image *vline* except that the pixels on the last scan-line have grey value 1, i.e. $im[H-1,y]=1$. Clearly, these images are artificial images. We also used some more realistic images, which are shown in Fig. 1. The first image consists of 50 squares of random sizes, located at random positions. Each square has a unique grey value. The second image is a camera-made image of handwritten music. The third image is slice 50 of a $93\times256\times256$ CT-scan of a head. The number of grey values is reduced from 256 to 32 to reduce the influence of noise.

Table 2
Speedups for the test set on the ES40

Image	256 × 256			512 × 512			1024 × 1024		
	S_2	S_3	S_4	S_2	S_3	S_4	S_2	S_3	S_4
empty	1.7	2.0	2.3	1.8	2.3	2.7	1.9	2.6	3.2
vline	1.7	1.8	2.4	1.8	2.2	3.0	1.8	2.5	3.4
hline	1.4	1.4	1.3	1.7	1.9	2.0	1.8	2.4	2.7
comb	1.7	1.8	1.9	1.8	2.2	2.6	1.8	2.5	3.0
squares	1.7	2.0	2.2	1.8	2.3	2.7	1.9	2.6	3.2
music	1.5	1.7	1.8	1.7	2.2	2.4	1.9	2.6	3.2
CT	1.6	1.8	2.0	1.8	2.3	2.8	1.9	2.7	3.3

Table 3
Speedups for the test set on the Cray J90

Image	256 × 256				512 × 512				1024 × 1024			
	S_2	S_4	S_8	S_{16}	S_2	S_4	S_8	S_{16}	S_2	S_4	S_8	S_{16}
empty	1.9	3.5	5.7	6.4	2.0	3.7	7.1	11.0	2.0	3.9	7.8	14.1
vline	2.0	4.0	6.9	9.6	2.0	3.9	7.4	12.2	2.0	4.0	7.7	15.4
hline	1.8	3.3	4.5	3.6	1.9	3.6	6.1	7.6	2.0	3.8	7.3	11.4
comb	1.9	3.4	5.2	5.4	2.0	3.8	7.0	10.9	2.0	4.0	7.8	13.5
squares	2.0	4.0	6.4	6.6	2.0	3.9	7.4	11.4	2.0	4.0	7.9	14.0
music	2.0	3.8	5.9	6.2	1.9	3.8	7.3	10.5	2.0	3.9	7.9	14.1
CT	1.9	2.8	4.4	4.5	2.0	3.6	6.8	10.7	2.0	4.0	7.9	14.7

For the artificial images, path compression is extremely effective. For the more realistic images path compression is worthwhile, but is less effective. For these images, it turns out that the running time of the algorithm is hardly dependent on the content of the image. For most camera made images, the algorithm runs in approximately the same time.

In Tables 2 and 3, we see the speedup using more than one processor. The measurements are performed on the test set for different image sizes. The number S_N is the speedup of the algorithm running on N processors relative to execution on one processor, defined by $S_N = T_1/T_N$, where T_N is the running time on N processors. We clearly see, that the speedup gets better if the computational task size increases. This is to be expected, since the ratio between computation and communication gets in favour of the computational side. This effect is especially severe on the ES40, since its processors are much faster than those of the Cray, while the memory speed (and thus communication speed) is about the same.

On both machines we see that the image *vline* performs best. Again, this is to be expected, since there is no communication needed at all. The image *hline* on the other hand performs worst, since here the amount of communication is maximal among the images considered. Even for this case, however, the speedups are satisfactory. The

Table 4
Speedups for the 3-D CT data set

ES40		CRAY J90									
N	S_N	N	S_N	N	S_N	N	S_N	N	S_N	N	S_N
2	**1.8**	2	**2.0**	5	**4.6**	8	**7.2**	11	**9.2**	14	**10.9**
3	**2.4**	3	**2.9**	6	**5.6**	9	**7.9**	12	**9.9**	15	**11.5**
4	**3.1**	4	**3.8**	7	**6.4**	10	**8.6**	13	**10.5**	16	**12.7**

image *empty* gains most from the optimization mentioned above. For this image, the lists *edlis1* initially contain each only a single pair.

For the more realistic images *square*, *music* and *CT*, we see very nice results. This is of course the main goal of the algorithm. For large enough images, up to about 8 processors we see an almost linear speedup. If we add more processors, we see a slight drop in the efficiency as a result of relative increase of communication with respect to the computational task. However, an efficiency of generally more than 75% is very satisfactory.

We also applied the 3-D version of the algorithm to a CT data set with sizes $93 \times 256 \times 256$. In Fig. 1(c), we see slice 50 of this set. The amount of grey values was reduced from 256 to 32 grey values. In Table 4, we present the results for both architectures. The left-hand frame contains the results on the ES40, with $T_1 = 3.1$ s. The right-hand frame contains the results on the Cray J90, with $T_1 = 149$ s. The results show the same tendencies as the two-dimensional results.

8. Conclusion

The computation of the connected components of an image (2-D or 3-D) can effectively be distributed over a number of processors. The amount of communication needed can only be determined at runtime, but is for most natural images quite modest. We used a variation of Tarjan's connected components algorithm. The communication is based on message passing, but implemented in shared variables by means of POSIX thread primitives. The experiments show a speedup that is often almost linear in the number of processors.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, Data Structures and Algorithms, Addison-Wesley, Reading, MA, 1983.
[2] H.M. Alnuweiri, V.K. Prasanna, Parallel architectures and algorithms for image component labelling, IEEE Trans. Pattern Anal. Mach. Intell. 14 (1992) 1014–1034.
[3] G.R. Andrews, Concurrent Programming, Principles and Practice, Addison-Wesley, Reading, MA, 1991.
[4] G.R. Andrews, R.A. Olsson, The SR Programming Language, Concurrency in Practice, Benjamin/Cummings, Mento Park, CA, 1993.
[5] K.R. Apt, E.-R. Olderog, Verification of Sequential and Concurrent Programs, Springer, Berlin, 1991.

[6] J. Bacon, Concurrent Systems, Addison-Wesley Longman Ltd., 1998, ISBN 0-201-17767-6.
[7] N. Copty, S. Ranka, G. Fox, R.V. Shankar, A data parallel algorithm for solving the region growing problem on the connection machine, J. Parallel Distributed Comput. 21 (1994) 160–168.
[8] C. Fioro, J. Gustedt, Two linear time union-find strategies for image processing, Theoret. Comput. Sci. 154 (1996) 165–181; 21 (1994) 160–168.
[9] S. Hambrusch, X. He, R. Miller, Parallel algorithms for gray-scale digitized picture component labelling on a mesh-connected computer, J. Parallel Distributed Comput. 20 (1994) 56–68.
[10] S. Kleiman, D. Shah, B. Smaalders, Programming with Threads, SunSoft Press, Prentice-Hall, Englewood Cliffs, NJ, 1996, ISBN 0-13-172389-8.
[11] R. Lumia, L.G. Shapiro, O. Zuniga, A new connected components algorithm for virtual memory computers, Comput. Vision Graphics Image Process. 22 (1983) 287–300.
[12] N.A. Lynch, Distributed Algorithms, Morgan Kaufman, San Francisco, 1996.
[13] A. Rosenfeld, J.L. Pfaltz, Sequential operations in digital picture processing, J. ACM 13 (1966) 471–494.
[14] J.L.A. van de Snepscheut, What Computing is all About, Springer, Berlin, 1993.
[15] R.E. Tarjan, Data Structures and Network Algorithms, Regional Conference Series in Applied Mathematics, SIAM, Philadelphia, 1983.
[16] G. Tel, Distributed Algorithms., Cambridge University Press, Cambridge, 1994.