university of groningen

University Medical Center Groningen

University of Groningen

The Implementation of a Parallel Watershed Algorithm

Meijster, A.; Roerdink, J.B.T.M.

Published in:
EPRINTS-BOOK-TITLE

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
1995

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Meijster, A., & Roerdink, J. B. T. M. (1995). The Implementation of a Parallel Watershed Algorithm. In *EPRINTS-BOOK-TITLE* University of Groningen, Johann Bernoulli Institute for Mathematics and Computer Science.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 28-12-2022

The Implementation of a Parallel Watershed Algorithm *

A. Meijster and J.B.T.M. Roerdink
University of Groningen,
Institute for Mathematics and Computing Science
P.O. Box 800, 9700 AV Groningen, The Netherlands
Email: arnold@cs.rug.nl roe@cs.rug.nl
Tel. +31-50-633931, Fax. +31-50-633800

Abstract

In this paper the implementation of a parallel watershed algorithm is described. The algorithm is implemented on a multiple instruction multiple data (MIMD) ring-architecture using a single program multiple data (SPMD) approach using an asynchronous message passing interface and simulated shared memory via the Linda tuple space. The watershed transform is generally considered to be inherently sequential. This paper shows that it is possible to exploit parallelism by splitting the computation of the watersheds of an image into three stages that can be executed in parallel. This paper is an extended version of [4].

Keywords: watershed algorithm, parallel implementation, image segmentation.

1 Introduction

In the field of image processing and more particularly in grey scale Mathematical Morphology [5, 6] the watershed transform [2, 3, 8] is frequently used as one of the stages in a chain of image processing algorithms. Unfortunately, the computation of the watershed transform of a grey scale image is a relatively time consuming task and therefore usually one of the slowest steps in this chain. A common solution for such computationally expensive algorithms is to search for implicit parallelism in the algorithm and use this to implement the algorithm on a parallel computer. Unfortunately this technique does not apply to the watershed algorithm, since the basic watershed algorithm is inherently sequential.

The watershed algorithm can easily be extended to graphs, as shown in [8]. This fact is used to derive an alternative algorithm which enables a parallel implementation. We first transform the image into a graph in which each vertex represents a connected component at a certain grey level h. Then we compute the watershed of this graph and transform the result back to the image domain. The computation of a skeleton of plateau regions is performed as a post-processing step.

2 The Watershed Transform

In [8] an algorithmic definition of the watershed of a digital grey scale image is given. In this section we will give a short summary of this definition.

A digital grey scale image is a function $f : D \longrightarrow \mathbb{N}$, where $D \subseteq \mathbb{Z}^2$ is the domain of the image (pixel coordinates) and for some $p \in D$ the value $f(p)$ denotes the grey value of this pixel. Grey scale images are looked upon as topographic reliefs where $f(p)$ denotes the altitude of the surface at location p. Let G denote the underlying grid, i.e. G is a subset of $\mathbb{Z}^2 \times \mathbb{Z}^2$. A *path* P of length l between two pixels p and q is an $l+1$-tuple $(p_0, p_1, ..., p_{l-1}, p_l)$ such that $p_0 = p$, $p_l = q$ and $\forall i \in [0, l) : (p_i, p_{i+1}) \in G$. For a set of pixels M the predicate $conn(M)$ holds if and only if for every pair of pixels $p, q \in M$ there exists a path between p and q which only passes through pixels of M. The set M is called connected if $conn(M)$ holds. A *connected component* is a nonempty maximal connected set of pixels. A *regional minimum* (minimum, for short) of f at altitude h is a connected component of pixels p with $f(p) = h$ from which

*In: Proc. Computing Science in the Netherlands, 27-28 november, Utrecht, 1995, pp. 134-142. Postscript version obtainable at http://www.cs.rug.nl/~roe/

it is impossible to reach a point of lower altitude without having to climb. Now, suppose that pinholes are pierced in each minimum of the topographic surface and the surface is slowly immersed into a lake. Water will fill up the valleys of the surface creating basins. At the pixels where two or more basins would merge we build a 'dam'. The set of dams obtained at the end of this immersion process, that is when the entire surface is flooded, is called the *watershed transform* of the image f.

Before going to the algorithm for computing watersheds, we need a few more definitions.

Definition 1. Let A be a set, and a, b two points in A. The *geodesic distance* $d_A(a, b)$ within A is the infimum of the lengths of all paths from a to b in A. If B is a subset of A, we define $d_A(a, B) := \inf_{b \in B}(d_A(a, b))$. In the digital case one uses an appropriate distance, such as the city-block distance function.

Now we will give the definition of influence zones. Let A be some set of pixels. Let $B \subseteq A$ be partitioned in k connected components B_i, i.e. $B = \bigcup_{i=1}^{k} B_i$.

Definition 2. The *geodesic influence zone* of the set B_i within A is defined as $iz_A(B_i) = \{p \in A \mid \forall j \in [1..k] \backslash \{i\} : d_A(p, B_i) < d_A(p, B_j)\}$.

The set $IZ_A(B)$ is defined as the union of the influence zones of the connected components of B, i.e.

$$IZ_A(B) = \bigcup_{i=1}^{k} iz_A(B_i) \tag{1}$$

Definition 3. The complement of the set $IZ_A(B)$ within A is called the *skeleton by influence zones* of A:

$$SKIZ_A(B) = A \backslash IZ_A(B) \tag{2}$$

3 The Classical Algorithm

A digital algorithm for computing the watershed transform was developed in [7, 8].

Definition 4. Let f be a grey level function. The set

$$T_h = \{p \in D \mid f(p) \leq h\} \tag{3}$$

is called the *threshold set* of f at level h.

Let h_{min} and h_{max} respectively be the minimum and maximum grey level of the digital image. Let Min_h denote the union of all regional minima at the altitude h.

Definition 5. (Watershed algorithm) Define the following recurrence:

$$X_{h_{min}} = \{p \in D \mid f(p) = h_{min}\}$$

$$X_{h+1} = X_h \cup \mathrm{Min}_{h+1} \cup (IZ_{T_{h+1}}(X_h) \backslash T_h), \quad h \in [h_{min}, h_{max}) \tag{4}$$

The *watershed transform* of the image f is the complement of $X_{h_{max}}$ in D:

$$Wshed = D \backslash X_{h_{max}} \tag{5}$$

Intuitively, one could interpret X_h as the set of pixels p, satisfying $f(p) \leq h$, that lie in some basin.

The recursion above is based upon the following case analysis [8], which is explained here in some detail in preparation of the parallel algorithm to follow.

For the recursive relation between X_h and X_{h+1} the threshold set T_{h+1} is considered. It is obvious that $X_h \subseteq X_{h+1} \subseteq T_{h+1}$. Let Y be a connected component of T_{h+1}. There are three possible relations between Y and X_h:

1. $Y \cap X_h = \emptyset$. In this case Y is a new minimum at level $h + 1$ and thus (after piercing a hole in it) the starting set of a new basin. Clearly $Y \subseteq X_{h+1}$.

2. $Y \cap X_h \neq \emptyset$ and is connected. Clearly Y is an extension of the basin X_h, and thus $Y \subseteq X_{h+1}$.

3. $Y \cap X_h \neq \emptyset$ and is not connected. In this case Y contains two or more distinct minima of f. Let $Z_1, \ldots, Z_k$ be these minima. Then the basin X_h is expanded by computing the geodesic influence zone of Z_i within Y.

Most implementations of algorithms that compute the watershed of a digital grey scale function are direct translations of the recursive relation (4). The basic structure of these algorithms is a main loop in which h ranges from h_{min} to h_{max}. In every iteration the basins belonging to the minima are extended with their influence zones within the set T_{h+1}. The fact that X_h is needed to compute X_{h+1} clearly expresses the sequential nature of this algorithm.

Computing influence zones is a costly operation, while it is a waste of time in the first two cases of the above case analysis. Also, the SKIZ is not necessarily connected, and may also be a 'thick' one, meaning that a set of pixels equally distant from two connected components may be thicker than one pixel.

The watershed algorithm as given above can easily be extended to graphs, as shown in [8]. This fact is used in the next section where we propose an alternative algorithm which enables a parallel implementation of this algorithm. We first transform the image into a graph in which each vertex represents a connected component at a certain grey level h. Then we compute the watershed of this graph and transform the result back to the image domain. The computation of a skeleton of plateau regions is performed as a post-processing step.

4 An Alternative Algorithm

In the algorithm described in the previous section influence zones were computed during every iteration of the algorithm. There is the problem of plateaus which may result in thick watersheds. Now, suppose that the image f does not contain plateaus, i.e. $\forall(p, q \in D : (p, q) \in G \Rightarrow f(p) \neq f(q))$. In this case every 'plateau' consists of exactly one pixel. This observation leads us to an alternative watershed algorithm, which consist of 3 stages:

1. Transform the image f into a directed valued graph $f^* = (F, E)$, called the *components graph of* f.
2. Compute the watershed of the directed components graph.
3. Transform the labeled graph into a binary image, and compute a skeleton of the watershed plateaus to get thin watersheds.

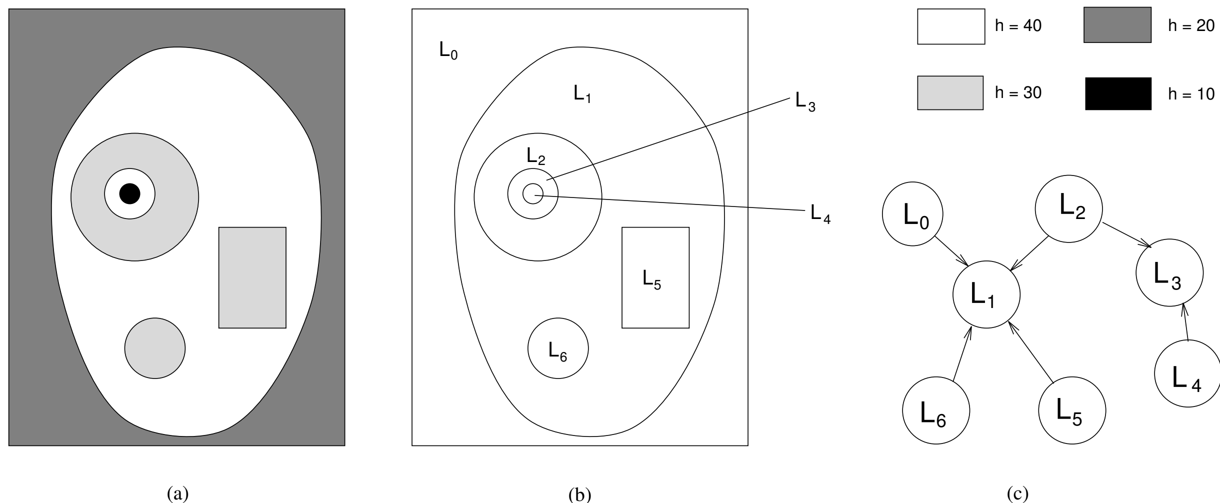

Figure 1: *(a) artificially generated image. (b) labeled level sets. (c) components graph.*

4.1 Stage 1

The first stage of this algorithm transforms the image f into a directed valued graph $f^* = (F, E)$, called the *components graph of* f. Here F denotes the set of vertices of the graph and E the set of edges.

The vertices of this graph are maximal connected sets of pixels which have the same grey values. In the remainder of this paper these sets are called *level components*. The set of level components at level h is defined as

$$L_h = \{C \subseteq T_h \backslash T_{h-1} \ : \ C \textit{ is a connected component of } T_h \backslash T_{h-1}\} \tag{6}$$

The set of vertices of the graph f^* is the collection of level components of f, i.e.

$$F = \bigcup_{h=h_{min}}^{h_{max}} L_h \tag{7}$$

A pair of level components (v, w) is an element of the edge set E if and only if $\exists(p \in v, q \in w : (p, q) \in G \wedge f(p) < f(q))$. By definition every directed path through this graph increases in altitude. With a little abuse of notation we denote the grey-value of a level component w by $f(w)$, which is the value $f(p)$ for some $p \in w$.

4.2 Stage 2

The second stage of the algorithm computes the watershed of the directed graph. The structure of this stage is very similar to the original watershed algorithm described in the previous section. The basic idea is to assign a colour (label) to each minimum and its associated basin by iteratively flooding the graph using a breadth first algorithm. If some node v can be assigned two or more different labels, i.e. the node can be reached from two different basins along an increasing path, the node is marked to be a watershed node. If the node can only be reached from nodes which have the same label the node is assigned this same label, i.e. the node is merged with the corresponding basin. A pseudo-code of this algorithm is given in Fig. 2.

4.3 Stage 3

In the third, and last, stage of the algorithm the labeled graph is transformed back into an image. The pixels belonging to a watershed node are coloured white while pixels belonging to non-watershed nodes are coloured black. After this transformation we end up with a binary image, in which the watersheds are plateaus. If we want thin watersheds we need to compute a skeleton of this image, for example the skeleton by influence zones as described in section 2. But also different types of skeletons can be used, which gives us more freedom than in the original watershed algorithm. If node v is a watershed node, we compute the skeleton of the set v by computing the influence zones of the non-watershed components.

5 Parallelization of the Second Algorithm

The runtime performance of the sequential algorithm proposed in the previous section turns out to be approximately the same as the performance of the algorithm described in [8]. For images containing many small level components the graph algorithm performs less well, since the components graph of such images is very large and thus it takes a relatively large time to build the graph. On the other hand, if the image contains larger level components the size of the graph decreases, taking less time to build the graph. Now the algorithm starts to outperform the classical algorithm, since we only have to compute the skeletons of watershed nodes. So, at first sight it appears we hardly gained anything using the graph algorithm.

However, since we clustered all the pixels which are in the same level component in one single node of the components graph, we can decide whether a node is a watershed node based on local arguments, i.e. we only have to look at the lower neighbours of the node in the graph. In the traditional watershed algorithm it is not possible to make this decision based on the altitude of neighbouring pixels since these pixels can be part of (very large) plateaus. Because of this fact it is hard to make a parallel version of the traditional watershed algorithm, since there will be substantial communication between the processors. In contrast to the traditional algorithm, the graph algorithm can be parallelized. In the rest of this paper we assume that we have a ring network of N processors. Each processor is identified by an identifier called *myproc*, which represents the number of the processor in the network. Each processor can communicate directly with both its neighbouring processors, using an asynchronous message passing interface. Each

processor has its own local memory for storing the program it executes and for storage of data. Also, there is a simulated piece of shared memory called the Linda tuple space [1].

```
MASK := -1; WSHED := 0; lab := 1;
for h := h_min to h_max do
begin (* mask all nodes at level h *)
      forall v ∈ F with f(v) = h do
         wsh[v] := MASK;
      (* extend basins *)
      forall v ∈ F with f(v) = h do
      begin iswshed := false;
            forall w ∈ F with (w, v) ∈ E do
               if ¬iswshed
               then if wsh[v] = MASK
                    then wsh[v] := wsh[w]
                    else if wsh[w] > 0
                         then if wsh[v] = WSHED
                              then wsh[v] := wsh[w]
                              else if wsh[v] ≠ wsh[w]
                                   then begin wsh[v] := WSHED;
                                              iswshed := true
                                        end
      end;
      (* process newly discovered minima *)
      forall v ∈ F with wsh[v] = MASK do
      begin wsh[v] := lab;
            lab := lab + 1
      end
end;
```

Figure 2: *Watershed algorithm on a components graph.*

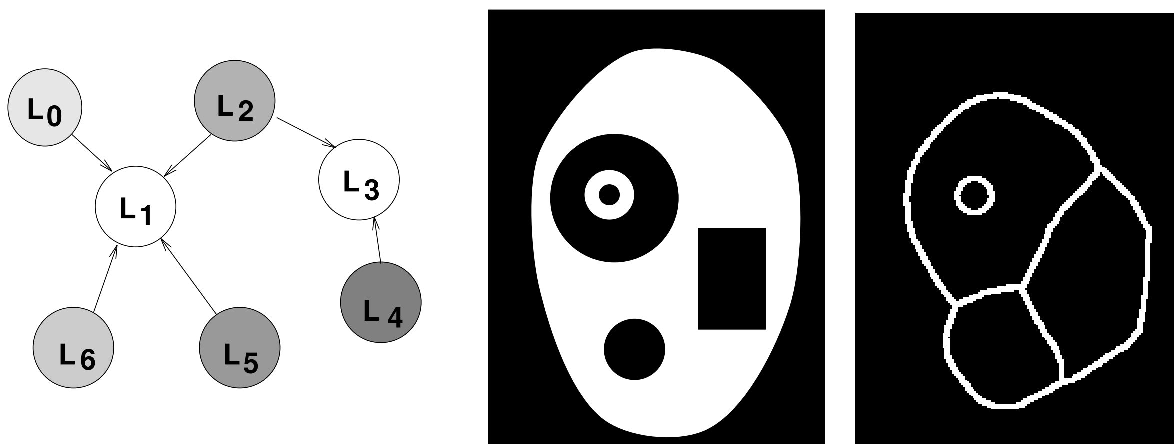

Figure 3: *(a) graph after flooding. (b) binary output image. (c) skeleton of output image.*

Every processor can perform three atomic operations on this tuple space. A processor can store a

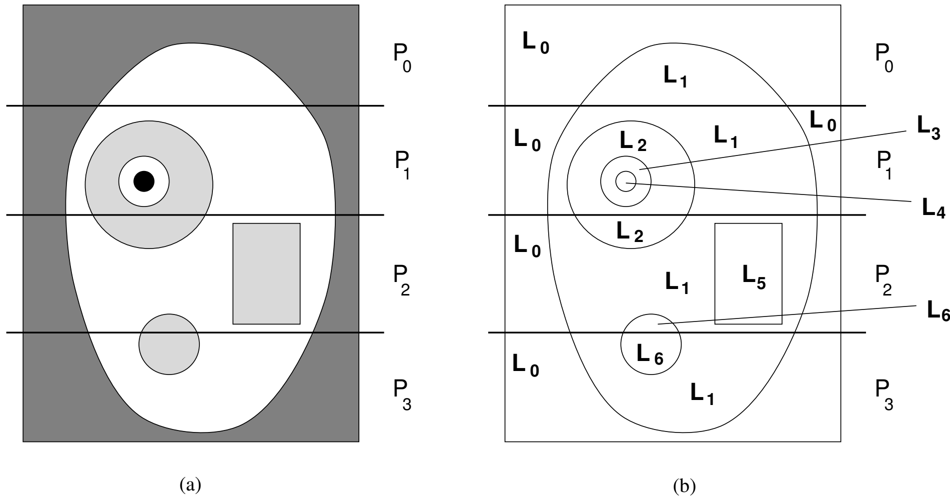

Figure 4: *(a) data distribution for four processors. (b) labeling of the distributed image.*

tuple (a, b) in the tuple space using the command **out** (a, b). It can read and delete a tuple from the tuple space using the command **in** (a, b). A tuple can be read from the tuple space without deleting it using the command **read** (a, b). For either of both reading operations, a must have an initialized value before the read operation. When the read operation is performed the runtime system tries to find a tuple in the tuple space which matches this value of a. If it finds such a tuple, let us say (a, c), the value c is assigned to b. If a processor performs two consecutive reading operations trying to find the same matching tuple, two distinct tuples result. If the runtime system cannot find a matching tuple the processor that called the read operation is blocked until some processor places a matching tuple in the tuple space.[1] The programming style we use is called SPMD (single program multiple data), which means that every processor runs exactly the same program, performing operations on its own data space.

5.1 Data Distribution and Level Components Labeling

The parallel implementation of the watershed algorithm consists of the same three stages as described in the previous section. The first stage concerns the labeling of the level components. This stage is performed by only one processor, let us say processor 0, on the entire image. After labeling this processor distributes the input image and the labeled image over the processors using the ring network. Let H and W respectively be the height and width of the input image. We assume that H is a multiple of N. If this is not the case the image is augmented with a few extra scanlines. The value of the pixels of these extra scanlines is set to h_{max} in order to avoid that new basins are introduced. Every processor is assigned a slice of H/N consecutive scanlines, while consecutive slices are assigned to neighbouring processors. Each processor also has one scanline overlap with its neighbouring processors, so that it can decide whether level components are shared with neighbouring processors. During the distribution of the image slices processor 0 builds up an integer valued table which is indexed by label numbers. This table, called *shared* in the program, denotes the number of processors that have at least one pixel of the corresponding component in its image slice. After distribution every processor receives a local copy of this table. This table is extensively used in the second stage of the algorithm.

The fact that the labeling is performed by only one processor instead of N processors is just a matter of implementation. It is possible to label the image in parallel, however labeling is a fast operation which is hardly worth the burden of parallelization.

[1] Full Linda implementations are more general than described here, but this subset of the semantics of the Linda tuple space suffices for our needs.

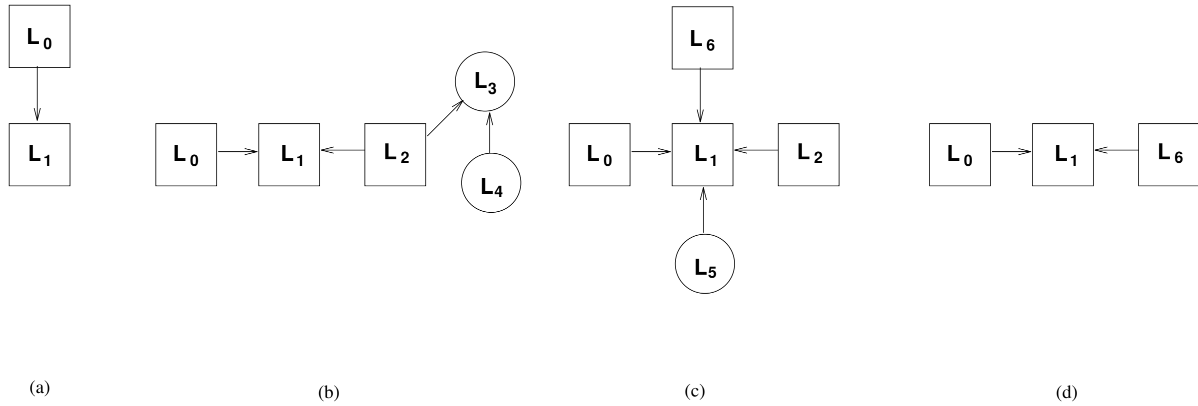

Figure 5: *(a)-(d) local components graph on processors P_0, P_1, P_2 and P_3*

5.2 Parallel Watershed Transform of a Graph

After the labeling stage every processor builds a local components graph for its own slice of the image. Since some level components are shared between several processors the graphs on the processors are not disjoint. In Fig. 5 the local graphs for the example image are shown. Shared vertices are drawn as a rectangular node while non-shared vertices are drawn as circular nodes. Note that a processor can easily determine whether a node v is shared, since in that case $shared[v] > 1$.

After building the local components graphs every processor performs an adapted version of the flooding algorithm. One of the problems to be solved is that a new minimum which is shared between two or more processors must be given the same new label. This is solved by introducing an integer array *owner* which is indexed by label numbers, just like the table *shared*. If $owner[v] = i$ for some minimum v then processor P_i, which has at least one pixel belonging to vertex v in its image slice, assigns a new label to this minimum, and stores this value in the tuple space. After putting this tuple in the tuple space every other processor sharing this vertex can read this newly created label and assign it to its local vertex v.

A similar method is used for expansion of basins. After performing local flooding for level h each processor puts the local colour of every shared vertex, which can be *MASK*, *WSHED* or some positive label, in the tuple space. After that, every processor retrieves these values from the tuple space and compares these values. If all these values are the same positive label number, the corresponding local copy of the vertex is coloured this label number. If not, the corresponding vertex is a watershed node.

At the end of the flooding process each processor transforms its local components graph back into an image slice, like in the sequential case. The result is a slice of the watershed transform of the input image. Since the watersheds in these slices can be thick plateaus we could decide to perform a skeletonization, like the skeleton by influence zones. This skeleton can be computed using a parallel or a sequential algorithm. In both cases, if we want to compute the skeleton of some level component v, we only need the pixels of the component v and its lower neighbours, which are easily accessible from the graph representation.

6 Conclusions

In this paper we have shown that it is possible to compute the watershed transform of a grey scale image in parallel by splitting the computation in three consecutive stages. In theory all these stages can be implemented in parallel, but in practice it is only worthy to implement the second stage in parallel.
In the first stage of the algorithm the input image is transformed into a directed components graph. In the second stage of the algorithm the watershed of this graph is computed by a breadth first coloring algorithm. The decision which colour to assign to a certain node can be made by examining the colors assigned to its neighbouring nodes. This locality property makes it possible to perform this stage in parallel, in contrast with the classical watershed algorithm. In the final stage of the algorithm the flooded graph is transformed back into the image domain. Pixels belonging to watershed nodes of the graph are coloured white, while pixels belonging to non-watershed nodes are coloured black. The resulting watersheds are 'thick'. 'Thin' watersheds can be obtained by performing some skeletonization algorithm

```
LAB := -2; MASK := -1; WSHED := 0;
if myproc = 0 then out (LAB, 1);
for h := h_min to h_max do
begin (* mask all nodes at level h *)
      forall v ∈ F with f(v) = h do
        wsh[v] := MASK;
      (* extend basins *)
      forall v ∈ F with f(v) = h do
      begin isw shed := false;
            forall w ∈ F with (w, v) ∈ E do
              if ¬iswshed
              then if wsh[v] = MASK
                   then wsh[v] := wsh[w]
                   else if wsh[w] > 0
                        then if wsh[v] = WSHED
                             then wsh[v] := wsh[w]
                             else if wsh[v] ≠ wsh[w]
                                  then begin wsh[v] := WSHED;
                                             iswshed := true
                                       end
      end;
      (* put labels of shared level components in tuple space *)
      forall v ∈ F with f(v) = h ∧ shared[v] > 1 do
            out (v, wsh[v]);
      (* read tuples from tuple space, determining whether v is watershed node *)
      forall v ∈ F with f(v) = h ∧ shared[v] > 1 do
      begin i := 0;
            while i ≠ shared[v] ∧ wshed[v] ≠ WSHED do
            begin read (v, tmp);
                  if wsh[v] = MASK
                  then wsh[v] := tmp
                  else if tmp ≠ MASK ∧ wsh[v] ≠ tmp
                       then wsh[v] := WSHED;
                  i := i + 1
            end;
      end;
      (* process newly discovered minima *)
      forall v ∈ F with wsh[v] = MASK do
      if owner[v] = myproc
      then begin in (LAB,lab);
                 wsh[v] := lab;
                 lab := lab + 1;
                 out (LAB,lab)
           end
      else read(v, wsh[v])
end;
```

Figure 6: *Each processor performs the above code in the parallel version of the watershed algorithm.*

on the output image. The choice which skeletonization algorithm to use is arbitrary.

References

[1] H. Bal, *Programming Distributed Systems.* Prentice Hall, 1990.

[2] S. Beucher and F. Meyer. The morphological approach to segmentation: The watershed transformation. In E.R. Dougherty, editor, *Mathematical Morphology in Image Processing.* Marcel Dekker, New York, 1993. Chapter 12, pp. 433–481.

[3] F. Meyer and S. Beucher. Morphological segmentation. Journal of Visual Communications and Image Representation, 1(1):21–45, 1990.

[4] A. Meijster and J.B.T.M. Roerdink. A Proposal for the Implementation of a Parallel Watershed Algorithm. In: Proceedings of Computer Analysis of Images and Patterns (CAIP'95), Springer Verlag, 1995.

[5] J. Serra, *Image Analysis and Mathematical Morphology.* Academic Press, 1982.

[6] S.R. Sternberg, Grayscale Morphology. Computer Vision, Graphics, Image Processing, **35**, pp. 333-355, 1986.

[7] L. Vincent. *Algorithmes Morphologiques a Base de Files d'Attente et de Lacets. Extension aux Graphes.* PhD thesis, Ecole Nationale Superieure des Mines de Paris, Fontainebleau, 1990.

[8] L. Vincent and P. Soille, Watersheds in Digital Spaces: An Efficient Algorithm Based on Immersion Simulations. IEEE Transactions on Pattern Analysis and Machine Intelligence, **13**, no. 6, pp. 583-598, june 1991.