A Proposal for 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 a parallel implementation of a watershed algorithm is proposed. The algorithm is designed for a ring-architecture with distributed memory and a piece of shared memory using a single program multiple data (SPMD) approach. 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.

1 Introduction

In the field of image processing and more particularly in gray scale Mathematical Morphology [5, 6] the watershed transform [2, 3, 7] is frequently used as one of the stages in a chain of image processing algorithms. Unfortunately, the computation of the watershed transform of a gray scale image is a relatively time consuming task and therefore usually one of the slowest steps in this chain. The use of a parallel watershed algorithm can significantly improve the overall performance. In [4] a distributed algorithm for the watershed transform was developed by splitting the input image into equally sized blocks. Communication overhead turned out to be a major problem.

The watershed algorithm can easily be extended to graphs, as shown in [7]. This fact is used to derive an alternative algorithm which is suitable for parallel implementation. We first transform the image into a graph in which each vertex represents a connected component at a certain gray level h. Then we compute the watershed of this graph and transform the result back into an image. The computation of a skeleton of plateaus is performed as a post-processing step.

2 The Classical Algorithm

A digital algorithm for computing the watershed transform was developed by Vincent [7]. In this section we will give a short summary of this algorithm.
A digital gray 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 gray value of this pixel. Gray 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* of f at altitude h is a connected component of pixels p with $f(p) = h$ from which 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 is called the *watershed transform* of the image f.

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. Let $B \subseteq A$ be partitioned in k connected components B_i, i.e. $B = \bigcup_{i=1}^{k} B_i$. 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. $IZ_A(B) = \bigcup_{i=1}^{k} iz_A(B_i)$. The complement of the set $IZ_A(B)$ within A, i.e. $SKIZ_A(B) = A \backslash IZ_A(B)$, is called the *skeleton by influence zones* of A. The set $T_h(f) = \{p \in D \mid f(p) \leq h\}$ is called the *threshold set* of f at level h. Let h_{min} and h_{max} respectively be the minimum and maximum gray level of the digital image. Let Min_h denote the union of all regional minima at the height h.

Definition Watershed algorithm Define the following recurrence:

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

$$X_{h+1} = \mathrm{Min}_{h+1} \cup IZ_{T_{h+1}(f)}(X_h), \qquad h = h_{min}, \ldots, h_{max} - 1. \tag{1}$$

Intuitively, one could interpret $X_h(f)$ as the set of pixels p, satisfying $f(p) \leq h$, that lie in some basin. The *watershed transform* of the image f is the complement of $X_{h_{max}}(f)$ in D:

$$Wshed(f) = D \backslash X_{h_{max}}(f). \tag{2}$$

Most implementations of algorithms that compute the watershed of a digital gray scale function are translations of the recursive relation (1). The fact that X_h is needed to compute X_{h+1} expresses the sequential nature of this algorithm.

Computing influence zones is a costly operation, while it is not necessary to compute them for non-watershed plateaus. 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.

3 An Alternative Algorithm

In the algorithm described in the previous section influence zones are 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 $f^* = (F, E)$.
2. Compute the watershed of the directed graph.
3. Transform the labeled graph back into a binary image.

3.1 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, called *level components*, of pixels which have the same gray-values. The set of level components at level h is defined as

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

The set of vertices of the graph f^* is the collection of level components of f, i.e. $F = \bigcup_{h=h_{min}}^{h_{max}} L_h$. A pair of sets (v, w) is an element of E if and only if $\exists p \in v, q \in w : (p,q) \in G \wedge f(p) < f(q)$. With a little abuse of notation we denote the gray-value of a level component w by $f(w)$, which is the value $f(p)$ for some $p \in w$.

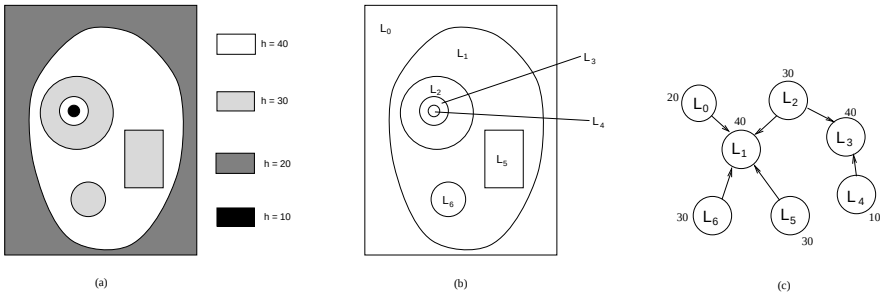

Fig. 1. (a) artificially generated image. (b) labeled level sets. (c) components graph.

3.2 Stage 2

The second stage of the algorithm computes the watershed of the directed graph. The procedure is very similar to the classical algorithm. The basic idea of the algorithm is to assign a colour (label) to each minimum node and its associated basin by iteratively flooding the graph using a breadth first algorithm. If to some node v there 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. 3.

Fig. 2. (a) graph after flooding. (b) binary output image. (c) skeleton of output image.

3.3 Stage 3

In the third 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 but other types of skeletons can be used as well.

MASK := -1; *WSHED* := 0; *lab* := 1;
for h := h_{min} **to** h_{max} **do**
begin forall $v \in F$ **with** $f(v) = h$ **do** $wsh[v]$:= *MASK*; $(*$ mask nodes at level h $*)$
forall $v \in F$ **with** $f(v) = h$ **do begin** $(*$ extend basins $*)$
$iswshed$:= **false**;
forall $w \in F$ **with** $(w, v) \in E \wedge \neg iswshed$ **do**
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 $wsh[v] \neq wsh[w]$ **then begin** $wsh[v]$:= *WSHED*; $iswshed$:= **true end**
end;
$(*$ process newly discovered minima $*)$
forall $v \in F$ **with** $wsh[v] = MASK$ **do**
begin $wsh[v]$:= lab; lab := lab + 1 **end**
end;

Fig. 3. Watershed algorithm on a graph.

4 Parallelization of the Graph Algorithm

It turns out that the average performance of our algorithm is approximately the same as that of the classical 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. Because of this fact, in contrast with the classical 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 has an unique identifier called $myproc$ and can communicate with both its neighbouring processors. Each processor has its own local memory for storing data, and a simulated piece of shared memory called the Linda tuple space [1]. Three atomic operations can be performed on this tuple space. A tuple (a, b) is stored using the command **out** (a, b). A tuple is read and deleted using the command **in** (a, b), while a tuple can be read without deleting it using the command **read** (a, b). When the read operation is performed the runtime system tries to find a tuple which matches the value of a. If

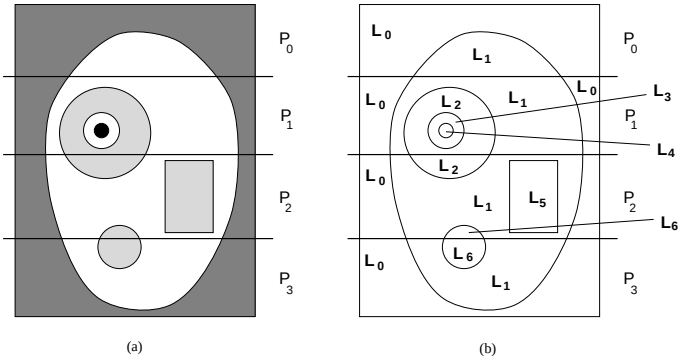

Fig. 4. (a) data distribution for four processors. (b) labeling of the distributed image.

such a tuple exists, let us say (a, c), the value c is assigned to b, otherwise the operation is blocked until some matching tuple is stored in the tuple space.[1]

4.1 Data Distribution and Level Components Labeling

The parallel algorithm consists of the same three stages as the sequential one. The labeling of the level components is performed by only one processor, since this is a very fast operation which is hardly worth the burden of parallelization.

After labeling of the components the input image and the labeled image are distributed. Each processor is assigned an equally sized slice of consecutive image rows, and consecutive slices are assigned to neighbouring processors. During distribution of the slices one processor builds up a table, called $shared$. The value $shared[i]$ denotes the number of processors that share component L_i. After distribution each processor receives a copy of this table. This table is extensively used in the second stage of the algorithm.

4.2 Parallel Watershed Transform of a Graph

After the labeling stage each processor builds a local components graph. Since some level components are shared these graphs are not disjoint. Each processor performs a modified version of the flooding algorithm on its own graph. A new minimum which is shared between two or more processors must be given the same label. This is done by introducing an array $owner$. If $owner[v] = i$ for some minimum v then processor P_i assigns a new label to this minimum, and stores this value in the tuple space, such that other processors sharing this vertex can read this label and assign it to its local vertex v. A similar method is used for expansion of basins. After flooding of level h each processor puts the local colour of every shared vertex in the tuple space. After that, every processor retrieves these values and compares them. If all these values are the same label number, the corresponding local copy of the vertex is coloured with this number, otherwise it is coloured *WSHED*.

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

```
LAB := -2; MASK := -1; WSHED := 0; if myproc = 0 then out (LAB, 1);
for h := h_min to h_max do
begin forall v ∈ F with f(v) = h do wsh[v] := MASK;    (* mask nodes at level h *)
      forall v ∈ F with f(v) = h do   (* extend basins *)
      begin iswshed := false;
            forall w ∈ F with (w, v) ∈ E ∧ ¬iswshed do
              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;
      (* now we have to take care of shared level components *)
      forall v ∈ F with f(v) = h ∧ shared[v] > 1 do out (v, wsh[v]);
      forall v ∈ F with f(v) = h ∧ shared[v] > 1 do begin
        i := 0; while i ≠ shared[v] ∧ wshed[v] ≠ WSHED do
        while i ≠ shared[v] ∧ wshed[v] ≠ WSHED do
        begin i := i + 1; read (v, tmp);
              if wsh[v] = MASK then wsh[v] := tmp
              else if tmp ≠ MASK ∧ wsh[v] ≠ tmp then wsh[v] := WSHED
        end
      end;
      forall v ∈ F with wsh[v] = MASK do   (* process newly discovered minima *)
      if owner[v] = myproc then
      begin in (LAB,lab); wsh[v] := lab;
            for i := 1 to shared[v] − 1 do out (LAB,lab + 1)
      end else read(v, wsh[v])
end;
```

Fig. 5. Parallel (SPMD) watershed algorithm on a graph.

At the end of the flooding process the local component graphs are transformed back into image slices. Since the watersheds in these slices can be thick plateaus we could decide to perform a skeletonization, which we regard as a postprocessing stage.

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.N. Moga, T. Viero, B.P. Dobrin, M. Gabbouj. Implementation of a distributed watershed algorithm. In J. Serra and P. Soille (Eds.), *Mathematical Morphology and Its Applications to Image Processing*, Kluwer, 1994, pp. 281-288.
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 Academic Press, 1982.
7. 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.