COMPUTATION OF WATERSHEDS BASED ON PARALLEL GRAPH ALGORITHMS *

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. +31-50-3633931, Fax. +31-50-3633800

Abstract. In this paper the implementation of a parallel watershed algorithm is described. The algorithm has been implemented on a Cray J932, which is a shared memory architecture with 32 processors. The watershed transform has generally been considered to be inherently sequential, but recently a few research groups, see [5, 9, 10], have designed parallel algorithms for computing watersheds. Most of these parallel algorithms are based on splitting the source image in blocks, computing the watersheds of these blocks and merging the resulting images into the desired result. A disadvantage of this approach is that a lot of communication is necessary at the boundaries of the blocks. It is possible to formulate the computation of the watershed transform as a shortest path searching problem that is commonly found in algorithmic graph theory. In this paper we use a parallel adapted version of Dijkstra's algorithm for computing shortest paths in undirected graphs.

Key words: watersheds, segmentation, shortest path algorithms, shared memory, parallelism

1. Introduction

Meyer gives in [7] a definition of the watershed of a digital gray scale image in terms of shortest paths. In this section we will give a short summary of this definition.

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 and $f(p)$ denotes the gray value of the pixel $p \in D$. Let E denote the underlying grid, i.e. E 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 $\forall i \in [0, l) : (p_i, p_{i+1}) \in E$. The length of a path P is denoted by $l(P)$. For a pixel $p \in D$ the set of neighboring pixels of p is defined as $N_E(p) = \{q \in D \mid (p, q) \in E\}$.

The *lower slope*, which is the maximal slope linking a pixel p to any of its neighbors of lower altitude, is defined as

$$LS(p) = \underset{q \in \{p\} \cup N_E(p)}{\text{MAX}} (f(p) - f(q))$$

The cost for walking from one position p to a neighboring position q is defined as

$$cost(p, q) = \begin{cases} LS(p) & \text{if } f(p) > f(q) \\ LS(q) & \text{if } f(p) < f(q) \\ \frac{LS(p)+LS(q)}{2} & \text{if } f(p) = f(q) \end{cases}$$

* In: *Mathematical Morphology and its Applications to Image and Signal Processing*, P. Maragos, R.W. Shafer, M.A. Butt (eds.), Kluwer, 1996, pp. 305-312. Postscript version obtainable at `http://www.cs.rug.nl/~roe/`

We denote the set of all paths from p to q by $p \rightsquigarrow q$. The *topographical distance* between two pixels p and q *along a path* $P = (p_0, ..., p_{l(P)})$ is defined as

$$T_f^P(p,q) = \sum_{i=0}^{l(P)-1} cost(p_i, p_{i+1})$$

The *topographical distance* between points p and q is defined as the minimum of the topographical distances along all paths between p and q:

$$T_f(p,q) = \underset{P \in p \rightsquigarrow q}{\mathrm{MIN}} \, T_f^P(p,q)$$

The topographical distance between a point $p \in D$ and a set $A \subseteq D$ is defined as:

$$T_f(p,A) = \underset{a \in A}{\mathrm{MIN}} \, T_f(p,a)$$

Note that $T_f(p,q) = 0$ if p and q are interior pixels of the same plateau. Now we construct a function f^* by replacing the values of f in all the local minima of f by 0, i.e. $f^*(p) = 0$ if p lies in a regional minimum, $f^*(p) = f(p)$ otherwise. Let $(m_i)_{i \in I}$ be the collection of minima of the function f^*. Note that these minima are sets, since a minimum can be a plateau instead of one single pixel. The catchment basin of a minimum m_i, denoted $CB(m_i)$, is defined as the set of points $p \in D$ that are topographically closer to m_i than to any other minimum m_j:

$$CB(m_i) = \{p \in D \mid \forall j \in I \backslash \{i\} : T_{f^*}(p, m_i) < T_{f^*}(p, m_j)\}$$

The watershed of a function f is the set of points of its domain which do-not belong to any catchment basin:

$$Wsh(f) = D \cap (\cup_{i \in I} CB(m_i))^c$$

2. Dijkstra's algorithm

In the previous section the definition of the watershed of a digital image is given. However, although this definition is mathematically sound, it is not immediately clear how to compute the watershed of a digital image, since the definition quantifies over all topographical paths between each pixel $p \in D$ and all the minima m_i.

In graph theory shortest path searching problems have been studied extensively. In the rest of this section we will review the problem of computing the lengths of the shortest paths in a given graph from a source node s to all the other nodes in this graph. We assume we have an undirected graph $G = (V, E)$, and a weight function $w : E \rightarrow \mathbb{N}$, that assigns a length to each edge of the graph. The goal is to find for each $v \in V$ the length of the shortest path from the source node s to v.

A well known algorithm for solving this problem[1] was found by E.W. Dijkstra in 1959 (see [1]). The algorithm is based on the fact that if $P = (p_0, p_1, .., p_n)$, with

[1] Actually, the general problem is to find the shortest path, instead of its length, but we are only interested in the length of this path.

```
procedure Dijkstra ($G=(V,E)$; $s \in V$; $w : E \to \mathbb{N}$; var $d : V \to \mathbb{N}$);
var $u : V$;
begin forall $v \in V$ do $d[v] := \infty$;
      $d[s] := 0$;
      while $V \neq \emptyset$ do
      begin $u := GetMinDist(V)$; (* find $u \in V$ with smallest $d$-value *)
            $V := V \backslash \{u\}$;
            forall $v \in V$ with $(u, v) \in E$ do
               if $d[u] + w[u, v] < d[v]$
               then $d[v] := d[u] + w[u, v]$
      end
end;
```

Fig. 1. Dijkstra's algorithm for an undirected graph $G = (V, E)$

$(p_i, p_{i+1}) \in E$, is the shortest path from a node p_0 to another node p_n, then the shortest path from p_0 to p_i, with $0 \leq i \leq n$, is given by $(p_0, ..., p_i)$. This trivial observation leads to a very elegant algorithm for solving the shortest path problem. The basic idea is to initialize for each node $v \in V \backslash \{s\}$ the distance between v and s to infinity, while the distance between s and itself is set to zero. After initialization, a wavefront starting in s is propagated through the graph along the edges of the graph. During the propagation we keep track of the distance the wavefront has traveled so far. When a node is reached by the wavefront and the distance traveled is smaller than the current value stored in this node, the value of this node is updated. This propagation process stops when all nodes of the graph have been reached by the wavefront. The pseudo-code of this algorithm is given in Fig. 1.

From the code of the algorithm it is clear that, assuming that the time complexity of the function *GetMinDist* is $O(1)$, the time complexity of the entire algorithm is $O(|E|)$, since each edge of the graph is traversed only twice[2]. Since $E \subseteq V \times V$, the time complexity can also be written as $O(|V|^2)$.

3. Computation of the Watershed based on Dijkstra's algorithm

If we compute the function *cost* of a digital gray scale image f, and use it as the weight function associated with the edges of the grid E, then Dijkstra's algorithm can be used to compute the topographical distance between each pixel and a local minimum m_i. In the rest of this paper all distances are topographical distances unless explicitly stated otherwise. Dijkstra's algorithm appears to be a very time consuming operation, since the number of nodes of the graph is the number of pixels in the image. However, because the graph is a digital image there are only 4, 6 or 8 edges leaving each node, in the cases of 4, 6, or 8-connectivity, respectively. Thus $|E| = \frac{k}{2} \cdot |V|$, where k denotes the connectivity we use. So, the time complexity of

[2] In a directed graph each edge is traversed only once.

```
procedure $SeqWshed$ ($E : D \times D$; $cost : E \rightarrow \mathbb{N}$; var $d : D \rightarrow (I \cup \{Wsh\}) \times \mathbb{N})$);
var $u : D$;
begin forall $v \in D$ do $d[v] := (0, \infty)$;
      forall $i \in I$ do
         forall $v \in m_i$ do $d[v] := (i, 0)$;
      while $D \neq \emptyset$ do
      begin $u := GetMinDist(D)$;
            $D := D \backslash \{u\}$;
            forall $v \in D$ with $(u, v) \in E$ do
               if $snd(d[u]) + cost[u, v] < snd(d[v])$
               then $d[v] := (fst(d[u]), snd(d[u]) + cost[u, v])$;
               else if $snd(d[u]) + cost[u, v] = snd(d[v])$
                    then $d[v] := (Wsh, snd(d[v]))$;
      end
end;
```

Fig. 2. Sequential Watershed Algorithm

Dijkstra's algorithm for this specific case is not quadratic in the number of pixels, but linear.

For the computation of the watershed of f we need to know the distance of each pixel $p \in D$ to each minimum $(m_i)_{i \in I}$, so we could apply the algorithm $| I |$ times, to compute the distances between each pixel p and each minimum in the image. However, we will modify the function d in Dijkstra's algorithm as follows. We store for each $p \in D$ in the first coordinate of $d[v]$ the index of the nearest minimum, and in the second coordinate the distance to this minimum. The resulting algorithm is given in Fig. 2. A wavefront is initiated in each minimum of the image. Each wave is labeled with the index of the minimum it started in. If wavefront i reaches a node p after it has propagated over a distance l, and l is less then the value of the second coordinate of $d[p]$, the value l is placed in the second coordinate of $d[p]$, while the first coordinate is set to i. If a node p is reached by another wavefront that has propagated over the same distance, the first coordinate of p is set to the artificial value *Wsh*, designating that p is a watershed pixel.

If, for the time being, we assume that $GetMinDist$ has time complexity $O(1)$, the sequential watershed has time complexity $O(| E |)$, which is the same as time complexity $O(| D |)$. Thus, if we are able to implement the function $GetMinDist$ such that it runs in constant time, we can compute the watershed of an image in an amount of time which is linear in the number of pixels of the image.

4. Implementation of *GetMinDist* using queues

In this section we will show that it it possible to implement the function $GetMinDist$ such that it has time complexity $O(1)$. The function should return the pixel p, which has not been reached by the wavefront yet, with the shortest distance to any of the

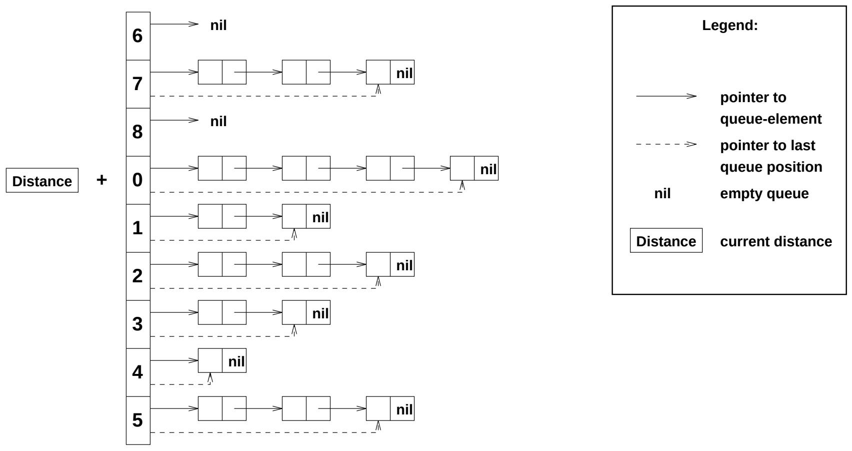

Fig. 3. A sample queue data structure

minima.

This can be realized with a priority queue of fifo-queues. It is implemented as a simple circular array. With each fifo-queue a distance is associated. This distance is the distance that a wavefront still has to travel before it will reach the pixels in this queue. The distances associated with the fifo-queues are used as the priority values in the priority queue – a smaller distance means a higher priority. In the fifo-queue with distance d associated with it, we store the pixels that will be reached by some wavefront after it travels a distance d further than where it is now. The order in which pixels of different plateaus are stored in these queues is irrelevant. The queues are fifo-queues, such that pixels which are located in the interior of a plateau, are ordered in this queue according to another distance function d^*, which measures how far pixels are away from the boundary of the plateau. For this function d^* one may take any of the standard metrics for binary images, such as the city-block distance in the case of 4-connectivity. In this way the algorithm automatically computes a skeleton by influence zones of such a plateau, if the plateau is reached by two or more waves at the same time. The priority queue is initialized with a fifo-queue (at index 0) containing all pixels that are located in the regional minima of the image. It is clear that, using this data structure, *GetMinDist* runs in $O(1)$ time, since it simply returns (and removes) the pixel at the front of the fifo-queue which is the first queue in the priority queue (queue with index 0 in fig. 3). This queue, and that pixel, are directly accessible. Insertion in the queues can also be done in $O(1)$ time, if we keep track of the last position in each fifo-queue, as well as the first position.

5. Parallelization of the Sequential Watershed Algorithm

It is easy to compute the lower slope and the cost function of an image in parallel, since the computation of the function value of a pixel is completely independent of the computation of this value for some other pixel. On the Cray J932, a shared memory computer, the speedup for computing these routines is almost linear with the number of processors.

The detection of minima is not entirely trivial, since local minima can be huge plateaus, and as a result we cannot decide whether a pixel is located in a regional minimum by just inspecting its value and those of its neighbors. To solve this problem, we use the algorithm for detecting local minima as given in [9][3]. The speedup of this algorithm is approximately linear in the number of processors, although the influence of concurrent references to the same memory locations starts to play a major role if we use many processors[4].

The computation of the watershed on the graph can also easily be parallelized. Given a shared memory computer with as many processors as there are minima, each processor computes the catchment basin belonging to a single minimum. Each processor has a private version of the queue data structures. The algorithm executed by a single processor is almost the same as the sequential code. The only difference is that the priority queue is initialized differently. Instead of placing all minima pixels in the queue only the minima pixels corresponding to the processor's minimum are placed in the queue.

In practice we do not have as many processors as the number of minima. If this number is M and the number of processors is P we assign to each processor the task to compute the catchment basins of $\lceil M/P \rceil$ minima. Of course the number M is in general not divisible by P, so one processor will be assigned a slightly smaller task, which may result in a slight load imbalance. Since we use shared memory, concurrent references to the same memory locations are to be expected. Since this can result in unpredictable behavior we have to synchronize these memory references using critical sections. Critical sections are sections of the program that can be executed by only one processor at the same time. These critical sections are implemented using binary semaphores (see [2]).

6. Performance Results

In general it is impossible to predict the exact speed-up of the parallel algorithm, since it is unknown a priori how many minima there are, and we do not know the size of the corresponding catchment basins. If the number of minima is smaller than the number of processors, we should not expect to gain speed by using more processors since each extra processor will be idle. In practice however, most images contain many more minima than the number of processors. Load imbalance as a result of different sizes of the catchment basins is a much more serious cause of decrease in speedup. In theory it is even possible that an image has catchment basins of only a few pixels, while some other catchment basin contains most of the pixels. In this case, the runtime performance of the parallel algorithm will be close to, or even worse than, the sequential algorithm running on a single processor, since the task to compute the large catchment basin is (almost) as expensive as computing the watershed of the entire image. However, if all the catchment basins are of approximately the same size, then the load balancing should be relatively even.

We tested the algorithm on a series of 6 images of 512×512 pixels. While running

[3] In [9] a MIMD algorithm is given, but it can easily be adapted for a shared memory system.
[4] For most images, we usually see a decrease in performance if we use more than 16 cpu's.

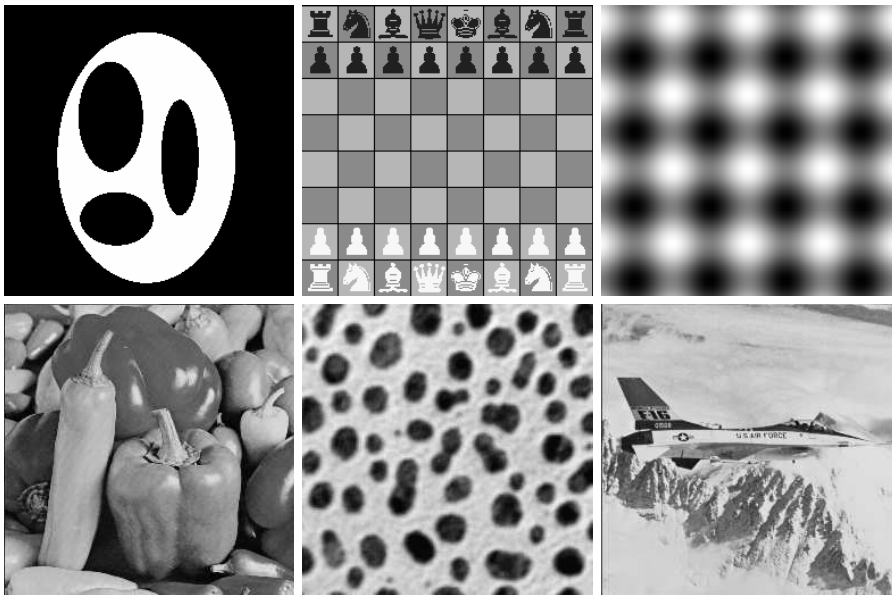

Fig. 4. (a) blobs (b) chess board (c) harmonic waves (d) peppers (e) gold particles (f) aircraft

TABLE I
Timings and speedups for the 6 test images

image	#minima	T_1	S_2	S_4	S_8	S_{16}
blobs	4	88	1.7	2.5	3.0	3.0
chess	67	101	1.6	2.3	3.6	4.0
waves	20	115	1.7	2.4	3.6	8.5
peppers	44426	111	1.7	2.1	3.0	5.0
gold	359	115	1.7	2.5	3.7	10.4
aircraft	19053	114	1.6	2.1	2.9	4.8

these tests we soon discovered that we do not gain significant speedup if we use more than 16 processors, since the tasks which are assigned to one processor are too small if we use more than 16 processors. For larger images it might very well be profitable to use more processors. For our test images we have decided to use not more than 16 processors. The results are given in table I. The column T_1 is the time (in seconds) for the computation of 100 watersheds on a single processor. In the column S_p the speedup is given if we use p processors.

We see that the speedup in the case of the *blobs* image remains the same if we keep adding more processors. The image contains only 4 regional minima, and thus each extra processor will remain idle. The poor speedup in the case of the *chess* *board* image is caused by the fact that it contains a widespread regional minimum – the boundaries of the squares. This minimum reaches over the entire image, causing a big load imbalance. The *peppers* image and the *aircraft* image contain many regional minima, most of them are noise resulting in many very small tasks causing a lot of overhead. The *waves* image and the *gold* image contain a reasonable number of uniformly distributed regional minima, resulting in a fairly good speedup.

7. Conclusions and further research

Computing watersheds in parallel is difficult. The original watershed algorithm proposed by Vincent and Soille (see [11]) is very hard to parallelize since this definition is an inherently sequential recursion. The definition given by Meyer (see [7]) used in this paper, offers some possibilities to compute watersheds in parallel using Dijkstra's shortest path algorithm.

Since we do not know a priori the size of a catchment basin associated with each minimum, load imbalance may occur. This will be the subject of study for future implementations. One solution is to reduce the number of minima using standard techniques to reduce over-segmentation. In practice we see that a lot of computing time is wasted on noise minima.

Another possible solution for the load imbalance is a better allocation of minima to the processors. If we allocate minima which are close to each other to the same processor wavefronts will get pruned earlier.

References

1. E.W. Dijkstra. A Note on Two Problems in Connexion with Graphs, In *Numerische Mathematik 1*, pp.269-271, 1959
2. E.W. Dijkstra. Co-operating Sequential Processes. In F. Genuys (ed.), *Programming Languages*, Academic Press, London, 1968, pp.43-112
3. 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.
4. J.A. McHugh. *Algorithmic Graph Theory*, Prentice-Hall, 1990.
5. A. Meijster and J.B.T.M. Roerdink. A Proposal for the Implementation of a Parallel Watershed Algorithm. In Proceedings Computer Analysis of Images and Patterns (CAIP'95), Springer Verlag, 1995, pp. 790-795.
6. F. Meyer and S. Beucher. Morphological segmentation. Journal of Visual Communications and Image Representation, 1(1):21–45, 1990.
7. F. Meyer. Integrals, gradients and watershed lines. In J. Serra and P. Salembier (Eds.), Proc. Workshop on *Mathematical Morphology and its Applications to Signal Processing*, Barcelona, 1993, pp. 70–75.
8. F. Meyer. Minimum spanning forests for morphological segmentation. In *Mathematical Morphology and its Applications to Image Processing*, J. Serra, P. Soille (eds.), Kluwer, 1994, pp. 77-84.
9. 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.
10. A.N. Moga, T. Viero, M. Gabbouj. Parallel Watershed Algorithm Based on Sequential Scanning. In I. Pitas (Ed.), *1995 IEEE Workshop on Nonlinear Signal and Image Processing*, June 20-22, Neos Marmaras, Halkidiki, Greece, pp. 991-994.

11. 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.