Author manuscript, published in "10th International Symposium on Mathematical Morphology (ISMM'11), France (2011)"
DOI : 10.1007/978-3-642-21569-8_22

hal-00622506, version 1 - 21 Oct 2012

Towards a Parallel Topological Watershed: First Results

Joël van Neerbos[1], Laurent Najman[2], and Michael H.F. Wilkinson[1]

[1]Johann Bernoulli Institute, University of Groningen
joelvneerbos@gmail.com, m.h.f.wilkinson@rug.nl
[2]Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge, A3SI, ESIEE
l.najman@esiee.fr

Abstract. In this paper we present a parallel algorithm for the topological watershed, suitable for a shared memory parallel architecture. On a 24-core machine an average speed-up of about 11 was obtained. The method opens up possibilities for segmentation of gigapixel images such as found in remote sensing routinely.

1 Introduction

The watershed transformation is a popular tool for segmenting grayscale images, introduced by S. Beucher and C. Lantuéjoul [2]. It can be used to segment an image into regions with similar gray values. Due to ever increasing image sizes, several parallel algorithms have become available for different watershed paradigms [9]. In contrast, no parallel algorithms for the *topological watershed* [1, 3] are available. In the topological watershed framework, some of the grayscale information from the original image is preserved, which may be useful for further processing, such as reconnection of corrupted contours. Also, this grayscale information can be used to determine the significance of watershed lines [7].

In this paper we present a parallel algorithm for the topological watershed on shared-memory parallel machines. The algorithm is based on parallellizing each of the stages of the sequential algorithm [4], and including a multi-pass stage such as used in most parallel algorithms for the regular watershed [9].

We will first describe the topological watershed briefly, after which we describe the parallel algorithm. Timing results are presented in Section 4, followed by the conclusions.

2 Topological Watersheds

The topological watershed [1, 3] was introduced to include grey level information in the end result, in such a way that the significance of each watershed line is preserved. Specifically, the topological watershed preserves the *pass values* [1, 8], i.e. the highest altitude of the lowest path between any two minima.

The pass value of two points in the image is related to a concept of *separation* of the points. The points p and q are said to be *k-separated* if the following conditions apply:

hal-00622506, version 1 - 21 Oct 2012

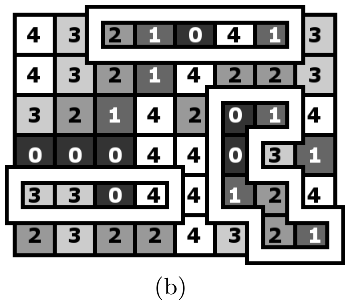

Fig. 1. Separation. (a) a digital grey-scale image; (b) lowest paths between three pairs of pixels.

– There exists a path from p to q with maximum value $k-1$
– There exists no path from p to q with a maximum value lower than $k-1$
– Both p and q have a value lower than $k-1$

A path that satisfies the first two conditions for some k is called a *lowest path* in this paper. If a lowest path from p to q contains no value that is higher than both p and q, then p and q are not separated, but are *linked*. Separation is illustrated in Fig. 1. The top path in Fig. 1(b) connects two 5-separated pixels. The left path connects pixels that are not separated (but linked). The right path connects two 3-separated pixels.

In contrast to most watershed algorithms, topological watershed does not produce a binary image, but a gray-scale image. Intuitively, grey values of the pixels are obtained as follows:

– All pixels in a basin have the same gray value, namely the value of the minimum from the input image that is contained within the basin.
– The values of the pixels on the watershed lines are as low as possible, without changing the separation relations between the basins. If two pixels from different basins were k-separated in the input image, they should still be k-separated in the topological watershed of the image.

The generic algorithm for computing a topological watershed of an image F proceeds by iteratively lowering some point satisfying a condition of destructibility until there is no more such points. More precisely, a point x is said to be W-destructible for F (where W stands for Watershed) if its altitude can be lowered by one without changing the number of connected components of the level set $\overline{F}[k] = \{p \in E; F(p) < k\}$, with $k = F(x)$. A map G is called a W-thinning of F if it may be obtained from F by iteratively selecting a W-destructible point and lowering it by one. A topological watershed of F is a W-thinning of F which contains no W-destructible point.

Both the watershed and a topological watershed of the image from Fig. 1(a) are shown in Fig. 2. Fig. 1(a)(c) shows that the separated pixel pairs from Figure 1(b) are still respectively 5-separated and 3-separated. The third pair is not shown as it was not separated.

hal-00622506, version 1 - 21 Oct 2012

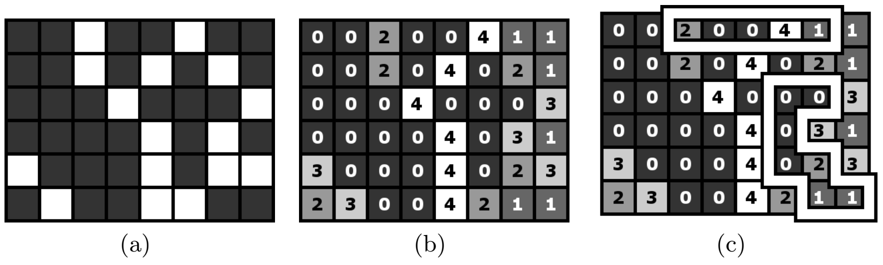

Fig. 2. Watershed and topological watershed: (a) watershed of Fig. 1(a); (b) topological watershed of Fig. 1(a); (c) lowest paths between the separated pairs of pixels from Fig. 1(b).

3 Parallel Implementation

To obtain a fast algorithm, we need to lower a W-destructible point not by 1 as in the generic algorithm, but as much as possible. Following the theoretical results in [4], the lowest value to which a point can be lowered can be computed from the Min-tree $\mathcal{C}(\overline{F})$. This Min-tree, or component tree, can be obtained using the parallel algorithm from [11]. Furthermore, the Min-tree allows one to easily check if two points are separated. More precisely, the separation is related to the altitude of the Lowest Common Ancestor (LCA) between two nodes of $\mathcal{C}(\overline{F})$. For efficiency, we need to perform preprocessing so that the LCA can be found in constant time. This is done using the algorithm in [10]. After this we need to lower all W-destructible pixels in the image to the value of their watershed basin. A pixel is W-destructible if its value can be lowered without linking two local minima into a single basin.

A key element of the algorithm is the function `W-Destructible`, which determines to which grey level a pixel can be lowered in the output. This uses the LCA value of neighbouring components of a the Min-tree. Because the function `W-Destructible` is a local function, we can parallelize the topological watershed algorithm for n threads simply by dividing the image into n tiles and assigning one tile to each thread. An example division for a 2D image is illustrated in Figure 3(a) and a division for a 3D volume is shown in Figure 3(d).

Problems arise when examining border pixels, because their neighbours in other tiles can be changed at any time by their assigned threads, producing incorrect results. We can solve these problems by letting each thread process its tile in different stages, and synchronizing all threads after each stage. Each tile is divided into sub-tiles, and a different sub-tile is processed in each stage. The tiles are divided in such a way that no two adjacent sub-tiles need to be processed at the same time.

Whether or not two sub-tiles are adjacent depends on the connectivity, as shown in Fig. 3. Fig. 3(a) shows an image divided into 12 tiles for 12 threads. With 4-connectivity, each tile should be divided into 2 sub-tiles, as shown in Fig. 3(b). The dark sub-tiles represent the sub-tiles that are processed in the first

hal-00622506, version 1 - 21 Oct 2012

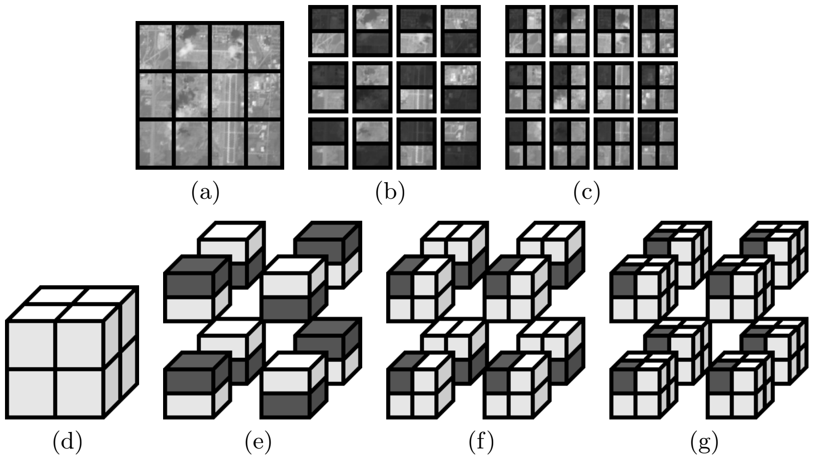

Fig. 3. Tiling: (a) 2D image; (b) tiling for 4-connectivity; (c) same for 8-connectivity; (d) 3D volume; (e) tiling for 6-connectivity; (f) same for 18-connectivity and (g) 26-connectivity;

stage. Note that no two dark sub-tiles are neighbours of each other. However, they would be neighbours with 8-connectivity, so 4 sub-tiles are used for 8-connectivity, as shown in Fig. 3(c). Fig. 3(d) shows a 3D volume divided into 8 'tiles' for 8 threads. For 6-, 18- and 26-connectivity, 2, 4 and 8 sub-tiles are needed, respectively. This is illustrated in Figures Fig. 3(e), (f) and (g). Again, the dark sub-tiles represent the sub-tiles to be processed in the first stage.

A single application of the multi-stage algorithm suggested above is however insufficient. For example, a certain pixel x may need to be lowered to the value of some local minimum y to obtain a topological watershed of the input image. However, if pixels x and y are part of different sub-tiles, it is possible that pixel x will not get the value of y the first time its sub-tile is processed. Multiple iterations may be needed to obtain the desired result, as in the case of regular watersheds [9].

The sequential topological watershed algorithm [4] processes all pixels in an order that is determined by a priority queue, built as preprocessing stage. In the parallel implementation, this is done when the pixels are processed for the first time, but visiting each pixel in every later iteration is not necessary. Instead, we keep track of the pixels that have been changed, and only add those pixels to the priority queue that are adjacent to pixels in other sub-tiles that changed recently. For this purpose we will use the binary map `pxChanged`, that will store for each pixel whether or not its value has changed recently. The use of this map is illustrated in Fig. 4.

The parallel algorithm is distributed over two procedures: the procedure `InitializeQueue` , which corresponds to the first part of the sequential algorithm, and the procedure `TopologicalWatershedTile`. , which corresponds to the second part. Both procedures are given below.

hal-00622506, version 1 - 21 Oct 2012

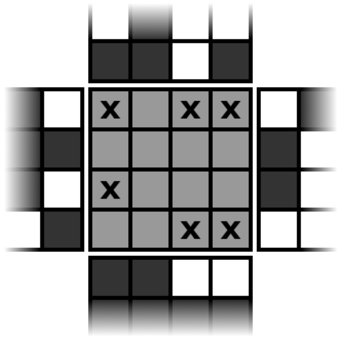

Fig. 4. An example of a `pxChanged` map in an iteration after the first. The white pixels in the four adjacent sub-tiles are pixels that are marked as 'changed' in the `pxChanged` map. The pixels in the current sub tile that are adjacent to such a changed pixel, marked with x in this figure, are added to the priority queue of the sequential algorithm.

As its input, the `InitializeQueue` procedure needs the image F, the min-tree $\mathcal{C}(\overline{F})$ and the corresponding component map Ψ that associates to each point the node it belongs to in $\mathcal{C}(\overline{F})$. Additionally, it also needs the sub-tile T in which it should operate, and the current state of the `pxChanged` map. If the pixel turns out to be W-destructible, it is added to the priority queue with its priority set to the level to which the pixel may be lowered. Also, this new level is stored in the map K and a pointer to the component to which the pixel may be added is stored in the map H. The pixel itself is not lowered yet. Thus, the output consists of the priority queue L, the maps K and H, and the updated `pxChanged` map. The queue L and the maps K and H are all local, but the map `pxChanged` is global, and may be read and modified by other threads while this procedure is being executed. However, each thread will only write in the part of the `pxChanged` map that corresponds to its current sub-tile, and will only read in adjacent sub-tiles that are processed in a different stage, so no conflicts emerge.

The algorithm starts by initializing the priority queue L in line `01`. It then proceeds by setting the `pxChanged` map to `false` for every pixel in sub-tile T. If the procedure is run during the first iteration, then lines `04` to `08` are executed. In the first iteration, L is initialized exactly like in the original sequential algorithm, apart from the fact that only the pixels within T are processed instead of all pixels in F.
If the procedure is called after the first iteration, lines `09` to `18` are executed. In these lines the algorithm checks all border pixels for changed neighbours, and tests the pixels for W-destructibility if any changed neighbours are found. If a pixel turns out to be W-destructible, it is added to the priority queue, and the maps K and H are updated as before.

The second procedure, `TopologicalWatershedTile`, needs the same input as the procedure `InitializeQueue`. The output consists of the updated image F, the update `pxChanged` map, and the binary variable `anyChanges`. The variable

hal-00622506, version 1 - 21 Oct 2012

Algorithm 1 Queue Initialization

```
Procedure InitializeQueue (Input $F, \mathcal{C}(\overline{F}), \Psi$, $T$, $pxChanged$;
Output $L$, $K$, $H$, $pxChanged$)
01. For $k$ From $k_{\min}$ To $k_{\max} - 1$ Do $L_k \leftarrow \emptyset$
02. For All pixels $p \in T$ Do $pxChanged[p] \leftarrow$ false
03. If first iteration Then
04.     For All pixels $p \in T$ Do
05.         $c \leftarrow$ W-Destructible($F, p, \mathcal{C}(\overline{F}), \Psi$)
06.         If $c \neq \emptyset$ Then
07.             $i \leftarrow$ level of $c$; $L_i \leftarrow L_i \cup \{p\}$
08.             $K(p) \leftarrow i; H(p) \leftarrow$ pointer to $c$
09. Else
10.     For All border pixels $p$ of $T$ Do
11.         addP $\leftarrow$ false
12.         For All neighbours $q$ of $p$ Do
13.             If $pxChanged[q]$ = true Then addP $\leftarrow$ true
14.         If addP = true Then
15.             $c \leftarrow$ W-Destructible($F, p, \mathcal{C}(\overline{F}), \Psi$)
16.             If $c \neq \emptyset$ Then
17.                 $i \leftarrow$ level of $c$; $L_i \leftarrow L_i \cup \{p\}$
18.                 $K(p) \leftarrow i; H(p) \leftarrow$ pointer to $c$
```

anyChanges is used to quickly determine if any changes have occurred in the sub-tile during the execution of this procedure.

The procedure TopologicalWatershedTile starts by initializing the value of anyChanges. The function call on line 02 produces an initialized priority queue L, as well as initialized maps K and H. The rest of the function is mostly the same as the second part of the sequential algorithm. Line 08 is added, where the pxChanged map and the anyChanges variable are updated. Also, a new restriction is added to line 09, saying that only neighbours of p that lie within the sub-tile T should be added to the priority queue.

With the procedure TopologicalWatershedTile implemented, we can now define the main parallel algorithm: the procedure ParallelTW. As its input, it needs the pixel mapping F, the Min tree $\mathcal{C}(\overline{F})$ and the corresponding component map Ψ. Because each thread will run this procedure independently, the global map pxChanged needs to be provided to each thread as well. However, no initial values need to be stored in it. Additionally, each thread is provided its identifier id. The first thread gets id value 0, the second gets value 1 and so on. The output of the procedure is the updated map F, that now contains the topological watershed of the input image.

The algorithm keeps looping until a topological watershed of the input image is found. In each iteration, the loop starting on line 02 is executed once by each thread. Line 04 starts a loop that visits all stages. The number of stages is equal to the number of sub-tiles assigned to each thread, as shown in Fig. 3. Line 05 then determines the location and dimensions of the sub-tile to be processed by the current thread in the current stage. Some examples of the sub-tiles that

hal-00622506, version 1 - 21 Oct 2012

Algorithm 2 The algorithm for a single tile

```
Procedure TopologicalWatershedTile (Input $F, \mathcal{C}(\overline{F}), \Psi$, $T$, $pxChanged$;
Output $F$, $pxChanged$, $anyChanges$)
01. $anyChanges \leftarrow$ false
02. InitializeQueue($F, \mathcal{C}(\overline{F}), \Psi$, $T$, $pxChanged$)
03. For $k$ From $k_{\min}$ To $k_{\max} - 1$ Do
04.     While $\exists p \in L_k$ Do
05.         $L_k = L_k \backslash \{p\}$
06.         If $K(p) = k$ Then
07.             $F(p) \leftarrow k; \Psi(p) \leftarrow H(p)$
08.             $pxChanged[p] \leftarrow$ true; $anyChanges \leftarrow$ true
09.             For All neighbours $q$ of $p$ within $T$, with $k < F(q)$ Do
10.                 $c \leftarrow$ W-Destructible($F, q, \mathcal{C}(\overline{F}), \Psi$)
11.                 If $c = \emptyset$ Then $K(q) \leftarrow \infty$
12.                 Else
13.                     $i \leftarrow$ level of $c$
14.                     If $K(q) \neq i$ Then
15.                         $L_i \leftarrow L_i \cup \{q\}; K(q) \leftarrow i$
16.                         $H(q) \leftarrow$ pointer to $c$
```

should be processed by each thread in the first stage are displayed in Fig. 3. In the other stages the sub-tiles that are processed should have a similar pattern, always assuring that no two adjacent sub-tiles are processed at the same time.
The topological watershed of the tile is then computed on line `06`. If the algorithm returns that there have been some changes, then this is stored for the current thread in the (global) `anyChangesThr` array. After this, a standard barrier function is called, that just waits until all threads have reached this barrier and then lets all thread continue. This is done to ensure that no thread will start with the next stage until all threads are done with the current one.
When all threads have finished processing all their sub-tiles, the first thread will check if there have been any changes in any of the threads. If there have not been any changes at all, a topological watershed has been found and all threads will terminate. Otherwise, each thread will go to the next iteration by starting again with the main loop from line `02`.

3.1 Min-Tree Compression

Wilkinson et al. [11] described how to parallelize the computation of a min-tree. Because the algorithm from [11] also deals with features of the min-tree that we won't use, its implementation is simplified somewhat. Basically, the sequential algorithm is parallelized by letting multiple threads each compute the min-tree of a different part of the input image, and merging the min-trees of the parts afterwards. This algorithm uses a representation of the min-tree which is as large as the image or volume itself, i.e. each pixel or voxel is a node, containing a pointer to its parent. Only those nodes which have a parent with grey level greater than its own are relevant to either filtering or the watershed computation.

hal-00622506, version 1 - 21 Oct 2012

Algorithm 3 The parallel topological watershed algorithm

```
Procedure ParallelTW (Input $F, \mathcal{C}(\overline{F}), \Psi$, $pxChanged$, $id$; Output $F$)
01. done $\leftarrow$ false
02. While not done Do
03.     $anyChangesThr[id] \leftarrow$ false
04.     For All stages $s$ Do
05.         $T \leftarrow$ current sub-tile, based on $id$ and $s$
06.         TopologicalWatershedTile($F, \mathcal{C}(\overline{F}), \Psi$, $T$, $pxChanged$)
07.         If $anyChanges =$ true Then $anyChangesThr[id] \leftarrow$ true
08.         Barrier()
09.     If $id = 0$ Then
10.         $anyChangesAtAll \leftarrow$ false
11.         For All threads $t$ Do
12.             If $anyChangesThr[t] =$ true Then
13.                 $anyChangesAtAll \leftarrow$ true
14.     Barrier()
15.     If $anyChangesAtAll =$ false Then done $\leftarrow$ true
```

These nodes are called *level roots*. When the min-tree is built in parallel, not all pixels of a given min-tree node directly point to the level root. This means finding a level root is costly. Because the LCA algorithm inspecs the level roots often, we need to compress the tree in the sense that all parent pointers always point to a level root, yielding what is refered to as a *canonical representation* of the tree [6]. This reduces the computation time of the LCA algorithm. The algorithm is shown in Alg. 4.

The procedure `CompressTree` needs a map F, a component tree $\mathcal{C}(\overline{F})$ and a corresponding component map Ψ as its input, as well as the identifier `id` of the thread that executes it. The output of the procedure consists of the compressed component tree and map. Function `LevelRoot2`is used by `CompressTree` to obtain the level roots, without ever writing in memory sections not assigned to the current thread.

Algorithm 4 The Min-tree compression algorithm.

```
Function LevelRoot2 (Input $c$, $F$, $\mathcal{C}(\overline{F})$)
01. If $c = \text{root}(\mathcal{C}(\overline{F})) \vee F(c) \neq F(\text{parent}(c))$ Then
02.     Return $c$
03. Else
04.     Return LevelRoot2(parent($c$), $\mathcal{C}(\overline{F})$, $F$)

Procedure CompressTree (Input $F$, $\mathcal{C}(\overline{F})$, $\Psi$, $id$; Output $\mathcal{C}(\overline{F})$, $\Psi$)
01. For All components $c \in$ segment $id$ of $\mathcal{C}(\overline{F})$ Do
02.     parent($c$) $\leftarrow$ LevelRoot2(parent($c$), $F$, $\mathcal{C}(\overline{F})$)
03. For All pixels $p \in$ segment $id$ of $F$ Do
04.     $\Psi(p) \leftarrow$ LevelRoot2($\Psi(p)$, $F$, $\mathcal{C}(\overline{F})$)
```

hal-00622506, version 1 - 21 Oct 2012

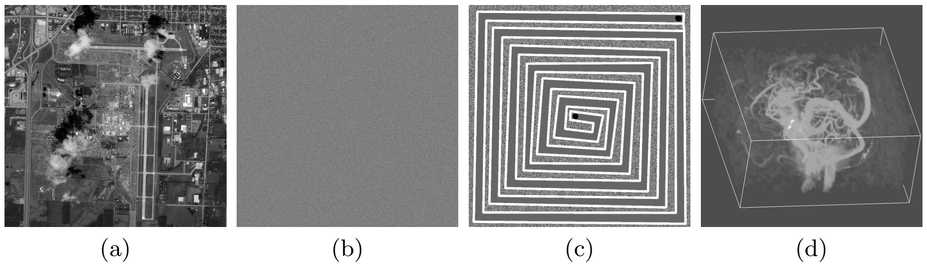
(a) (b) (c) (d)

Fig. 5. The four test input images. (a) shows a satellite image, (b) an image with random pixel values, (c) an image with a spiral-shaped plateau and (d) an angiogram, which is a 3D volume.

4 Results

The implementation was tested on the following four different input images: (i) a 4000 × 4000 satellite image of an airfield, (ii) a 4000 × 4000 image in which each pixel has a random gray value (iii) a 1000 × 1000 image with a spiral shaped plateau, and (iv) an angiogram with dimensions 256 × 256 × 128. The spiral image was chosen as a (near) worst-case scenario, in the same way as in [9]. The random noise image was another extreme case. The images are displayed in Fig. 5.

All data sets were tested on a 4-socket, 6-core per sockets, opteron-based machine with 128 GB of memory, using 1 to 24 threads, and various connectivities (4 and 8 in 2D, 6 and 26 in 3D). The memory is divided into banks assigned equally to each processor sockets, but is accessible as shared memory amongst all cores. Accessing memory of another porcessor socket does incur a slight speed penalty. In all cases, each thread was initially assigned an equal slice of the image or volume during the min-tree construction phase, as in [11], and an equal tile as described in section in the remainder of the algorithm as detailed above.

The previously existing sequential algorithm by Couprie et al. [4] and the newly implemented parallel algorithm were run on the same machine with the same input (the satellite image, with 4-connectivity using 1 thread) to compare the two versions on a single core.

On the satellite image, we obtained a wall-clock time of 86.36 s at 4 connectivity, for a single thread. This dropped to 17.14 s at 8 threads, 9.71 s at 16 threads, and 7.17 s at 24 threads. At 8 connectivity, all times rose, and the timings were 135.17 s at 1 thread, decreasing in a similar fashion to 12.04 s at 24 threads. The noise image (at 4 connectivity) showed a very similar pattern, decreasing from 112.83 s on 1 thread to 11.95 s on 24 threads. The influence of connectivity was most profound in the 3D case. At 6 connectivity, timings run from 73.18 s at 1 thread to 5.7 s at 24 threads. At 18 connectivity these figures rise to 214.71 s and 16.62 s respectivily. increasing further to 359.22 s and 33.37 s at 26 connectivity.

hal-00622506, version 1 - 21 Oct 2012

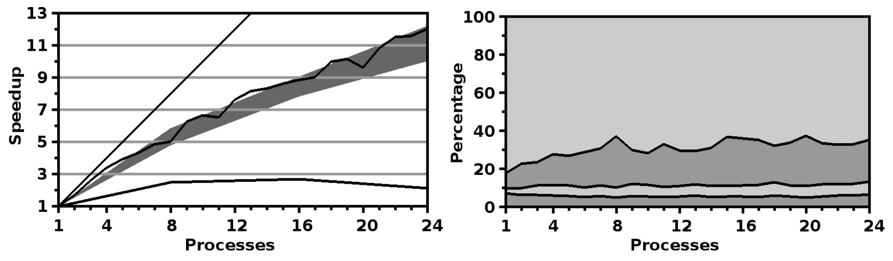

Fig. 6. The overall behaviour of the algorithm: (a) The total speedups in the performed tests, relative to the wall-clock time for a single thread with the same input; (b) the contributions of the four stages to the total wall-clock time of the algorithm, when computing the topological watershed for the satellite image with 4-connectivity.

As expected, the (smaller) spiral image behaved very differently, with timings of 4.29 s on 1 thread, 1.71 s on 8, 1.58 s on 16, and increasing again at 24 threads to 1.99 s. This is due to the increase in the number of iterations of the topological watershed algorithm.

The overall behaviour is shown in Fig. 6. In Fig. 6(a) the diagonal line shows the ideal speedup, the line at the bottom shows the speedup for the spiral image. The line through the gray area shows the speedup for the satellite image with 4-connectivity, which is computed for each number of threads individually. The grey area represents the results of the all tests, excluding for the spiral image. The top and bottom of the grey area are defined as the average speedup of their stage plus and minus the standard deviation, respectively. The diagonal lines show the ideal speedup, where the speedup is equal to the number of threads.

In Fig. 6(b) we show the contributions of the four stages to the total wall-clock time of the algorithm. The bottom layer represents the time consumed by the construction of the min-tree, the second layer from the bottom represents the time taken to compress the min-tree, the third layer shows the time it takes to perform the preprocessing for the LCA, and finally the remaining layer on top represents the time consumed by the final stage which produces the topological watershed.

The individual speedups are shown in Figure 7. The speedup of each stage is represented by a black line and a grey area, where the line shows the speedup of that stage when computing the satellite image with 4-connectivity. The corresponding grey area represents the results of the remaining tests as in Fig. 6, again excluding the spiral image.

The construction of the component tree and the final stage that computes the topological watershed parallelize quite well, while the tree compression and the LCA preprocessing parallelize rather poorly. Fortunately, the tree compression takes up only a small percentage of the total wall-clock time, even when 24 threads are used. However, this percentage will probably increase when more threads are used. The LCA has a larger impact on the total wall-clock time of

hal-00622506, version 1 - 21 Oct 2012

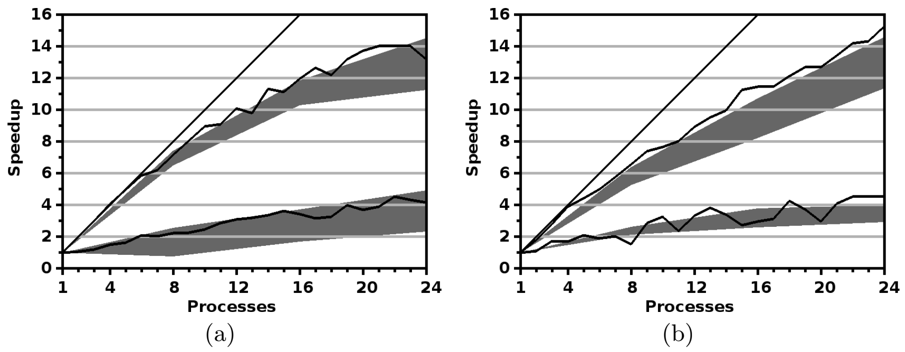

Fig. 7. The speedups of the four stages of the algorithm: (a) Top: min-tree construction, bottom: tree compression; (b) Top: topological watershed computation, bottom: LCA preprocessing.

the algorithm, and will cause the speedup of the total algorithm to decrease more severely when using a larger number of threads.

Note however, that all speedups reported are relative to the wall-clock times of the parallel algorithm using one thread. In practice, the previously existing sequential implementation performs about 1.5 times faster than parallel algorithm when only one thread is used. On two threads, the current algorithm has the edge, ableit by a small margin. Further optimizations in the parallel algorithm, for example by using a better tree compression method, may reduce this difference.

5 Conclusions

This paper described a way to parallelize the computation of the topological watershed. An implementation that was created according to this description showed that a reasonably good speedup could be achieved while using up to 24 threads, and the trend in the results suggests that even better speedups may be achieved when more than 24 threads are used.

The parallel implementation may be improved further by creating a more compact min-tree in the min-tree construction or compressing it more in the tree compression stage. For example, each node in the min-tree that has only one child can be merged with that child, setting all pixels belonging to its component to the gray level of its child, and setting the parent of its child to be the parent of the node itself. This simple change could significantly reduce the LCA preprocessing time, and may also improve the wall-clock time of the last stage where the topological watershed is computed.

Another improvement could be to use a more sophisticated method of parallel list ranking. Better performing algorithms already exist, the algorithm in this paper was used for simplicity reasons only. A list ranking algorithm that

hal-00622506, version 1 - 21 Oct 2012

parallelizes better and has higher speedups, may significantly reduce the total wall-clock time of the algorithm, especially when using larger numbers of threads.

Furthermore, the image is now divided into tiles which are each processed by their own thread. This division is only based on the image dimensions and the number of threads, not on the contents of the image. Taking into account the contents of the image, maybe in combination with the min-tree and component map, while dividing the image among the threads may reduce the communications needed between the different threads, which could lead to a faster algorithm. We can also try other ways of tiling the image, following [5].

In short, there is still room for improvement in the parallel algorithm proposed in this paper, but in its current form it can already be used to greatly speed up the computation of the topological watershed, compared to the previously existing sequential algorithm.

References

1. Bertrand, G.: On topological watersheds. Journal of Mathematical Imaging and Vision 22(2), 217–230 (2005)
2. Beucher, S., Lantuéjoul, C.: Use of watersheds in contour detection. In: International Workshop on image processing, real-time edge and motion detection/estimation. pp. 17–21 (1979)
3. Couprie, M., Bertrand, G.: Topological grayscale watershed transformation. In: SPIE Vision Geometry V Proceedings. vol. 3168, pp. 136–146. Citeseer (1997)
4. Couprie, M., Najman, L., Bertrand, G.: Quasi-linear algorithms for the topological watershed. J. Math. Imag. Vis. 22, 231–249 (2005)
5. Matas, P., Dokládalova, E., Akil, M., Grandpierre, T., Najman, L., Poupa, M., Georgiev, V.: Parallel Algorithm for Concurrent Computation of Connected Component Tree. In: Advanced Concepts for Intelligent Vision Systems (ACIVS'08). Lecture Notes in Computer Science, vol. 5259/2008, pp. 230–241. Springer-Verlag (Oct 2008)
6. Najman, L., Couprie, M.: Building the component tree in quasi-linear time. IEEE Trans. Image Proc. 15, 3531–3539 (2006)
7. Najman, L.: On the equivalence between hierarchical segmentations and ultrametric watersheds. J. Math. Imag. Vis. (2010), `http://www.laurentnajman.org`, to appear
8. Najman, L., Couprie, M., Bertrand, G.: Watersheds, mosaics and the emergence paradigm. Discrete Applied Mathematics 147(2-3), 301–324 (Apr 2005)
9. Roerdink, J., Meijster, A.: The Watershed Transform: Definitions, Algorithms and Parallelization Strategies. Fundamenta Informaticae 41, 187–228 (2001)
10. Schieber, B., Vishkin, U.: On finding lowest common ancestors: simplification and parallelization. SIAM Journal on Computing 17(6), 1253–1262 (1988)
11. Wilkinson, M.H.F., Gao, H., Hesselink, W.H., Jonker, J.E., Meijster, A.: Concurrent computation of attribute filters using shared memory parallel machines. IEEE Trans. Pattern Anal. Mach. Intell. 30(10), 1800–1813 (2008)