Connected Morphological Attribute Filters on Distributed Memory Parallel Machines

Jan J. Kazemier[1], Georgios K. Ouzounis[2], and Michael H.F. Wilkinson[1]

[1] Johann Bernoulli Institute, University of Groningen,
P.O. Box 407, 9700 AK, Groningen, The Netherlands
[2] DigitalGlobe, Inc., 1300 W 120th Ave, Westminster, CO 80234, USA
jan.kazemier@tno.nl, gouzouni@digitalglobe.com
m.h.f.wilkinson@rug.nl

Abstract. We present a new algorithm for attribute filtering of extremely large images, using a forest of modified max-trees, suitable for distributed memory parallel machines. First, max-trees of tiles of the image are computed, after which messages are exchanged to modify the topology of the trees and update attribute data, such that filtering the modified trees on each tile gives exactly the same results as filtering a regular max-tree of the entire image. On a cluster, a speed-up of up to 53× is obtained on 64, and up to 100× on 128 single CPU nodes. On a shared memory machine a peak speed-up of 50× on 64 cores was obtained.

1 Introduction

Attribute filters are powerful tools for filtering, analysis and segmentation [1, 2]. In the binary case attribute filters [1] remove connected components if some property (or attribute) of the latter, such as area, fails some threshold. This can be generalised to grey scale by performing the following steps:

- Compute all threshold sets.
- Organize all threshold sets into sets of connected components.
- Compute attribute values for each connected component of each set.
- Apply the binary attribute filter to each connected component set.
- Compute the combined filtered results of all threshold sets.

Components of threshold sets at higher levels can only be subsets of, or equal to those at lower threshold sets. Using this nesting property along the grey-scale, the set of all components can be organised into a tree structure referred to as a max-tree (or min tree if we want to filter background components) [3], as shown in Figure 1. Max-trees and min-trees are collectively referred to as component trees [4]. Once a component tree has been constructed and the attributes for each node have been computed, the attribute filter computed by removing unwanted nodes. Component trees are also used for multi-scale analysis, e.g., in the form of pattern spectra and morphological profiles [5–7].

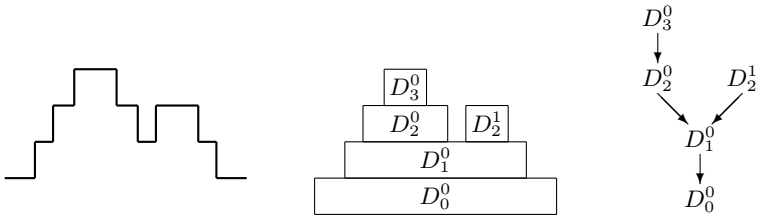


Fig. 1: A 1-D signal f (left), the corresponding peak components (middle) and the Max-Tree (right). Figure from [8].

The most time consuming part in attribute filtering and analysis using connected operators is building the max-tree. Many algorithms for computing these trees exist [3, 9, 10]. A shared-memory, parallel algorithm for creating max-trees has been proposed [8]. It relies in building separate max-trees for disjoint parts of the image, using a variant of the algorithm from [3]. These trees are subsequently merged into a single max-tree. This approach was extended in [11] to computing differential morphological profiles. The key problem limiting the algorithm to shared-memory machines, and limiting the size of images that can be processed is the fact that the entire max-tree needs to be stored in memory. Though efficient, it cannot handle images of tens of gigapixels, or even terapixels, such as those collected in remote sensing, or ultramicroscopy.

These very high-resolution images require a different approach of building the max-tree, which does not require the entire tree to be stored in memory. In this paper we develop an algorithm that computes a forest of max-trees for disjoint tiles of the input image, and then modifies these trees in such a way that filtering them results in the same output as filtering the whole tree. To be able to use distributed memory or hybrid memory model machines, an implementation of this new algorithm using MPI is given. A key part of the proposed solution is a structure called a *boundary tree*, which allows efficient communication between the processes; passing all required information, whilst keeping communication to a minimum.

The rest of this paper is organized as follows: Section 2 discusses our novel algorithm, and results are discussed in Section 3. The conclusion and an overview of future work are given in Section 4.

2 The proposed algorithm

In this section the design of the distributed memory parallel solution for filtering connected components based on their attribute(s) is described. In the following discussion we will focus on max-trees, but min-trees can be built equivalently.

The key idea is the following: Assuming we have a very large image stored as a series of disjoint tiles, we can of course compute a max-tree for each tile using any suitable algorithm, and merge them using the merging algorithm in [8]. However,

rather than passing the entire max-tree to a neighbouring tile for merger, we only select the necessary nodes, i.e. those that correspond to connected components that touch the boundary of the tile. Any component that does not touch the boundary cannot possibly be changed as a result of a merger. Therefore we create a subtree of the max-tree called the boundary-tree, effectively pruning away all branches of the full max-tree that do no touch the tile boundary. We can then merge the two boundary-trees of neighbouring tiles, by passing the boundary-tree of one processor to its neighbour. As a boundary-tree is essentially a pruned max-tree, we can use the same merging algorithm as in [8], where we use an extra flag to indicate which nodes have changed during merger. Only these nodes in the two involved max-trees (not boundary-trees!) are then updated, in terms of changed attributes and parent relationships. For a two-tile situation, the entire process can be summarised as:

1. Load the grey scale image tiles from disk.
2. Compute local max-trees for both parts.
3. Extract boundary-trees from max-trees
4. Merge the boundary-trees
5. Use the changed nodes in the merged boundary-tree to correct each of the local max-trees.

For more than two tiles, we assume the image is cut into a $P = a \times b$ grid, with P the number of processes, and a and b powers of 2. As before, each processor performs the single-processor algorithm on its tile. We end up with P max-trees. The boundary-tree is sent to the appropriate adjacent processor, which merges the boundary-trees, whilst updating the attribute(s). After merging, the corrected attributes and parent-child relations are propagated back to the corresponding processors. Once this update is completed, a boundary-tree of the merged region is created. This process repeats until the boundary-tree of the entire images has been formed. After this, the same filtering step as used in the shared memory program can be used, per processor, on its own tile, using the updated attribute(s) and topology.

Note that the order of the message size is $\mathcal{O}(G\sqrt{N})$ in this scheme, since we only need to send the border plus ancestors, rather than $\mathcal{O}(N)$ if we send full max-trees. The number the message steps is $\mathcal{O}(\log P)$. For an 8 bit-per-pixel $40{,}000^2$ tile this is a savings of at least a factor of 78 in communication and memory overhead.

The following sub-sections describe each of the steps in detail.

2.1 Building the max-tree

For each tile the max-tree is built using the same algorithm as used in the shared memory parallel solution from [8], albeit some differences in how the data is structured. In our solution all data is confined in MaxNodes as much as possible.

The max-tree is represented as a one-dimensional array of MaxNodes, which is a mapping to the 2D image, i.e. every node represents a pixel in the image (plus

extra information) and vice versa. A node in the max-tree, called a MaxNode consist of the following data:

- `index`, its index in the array.
- `parent`, the index of its parent.
- `area`, the area of all child-nodes pointing to this node. This is the attribute we filter on.
- `gval`, the grey value, or intensity, of the pixel.
- `filter`, the grey value of the pixel after filtering.
- `flags`, boolean flags that are set during execution.
- `borderindex`, after being put in the boundary-tree, this is the index thereof.
- `process`, the rank of the processor the initial max-tree is built on. This is used when sending back the updated attributes.

Only MaxNodes with a parent at a lower grey level represent a node in the max-tree. Such nodes are referred to as *level-roots*. Because we store the tree in an array, pointers are replaced by indices. The root of a max-tree (or boundary tree) has a parent field with a special index denoted $\perp$.

The flags field is a series of bits, of which just three are used:

- `Reached`, whether the flood tree algorithm has already reached this node
- `Added`, whether this node is already in the border; to avoid duplicates
- `Changed`, whether this node is changed in a round of merging; to prevent overhead

The algorithm used to build the tree is exactly the same as that in [8], so for space considerations it is not reproduced here.

2.2 The Boundary-Tree

Once all local max-trees have been computed, a *boundary-tree* is created. This structure consists of a one-dimensional array, containing all max-tree nodes in the border, and all of their ancestors, i.e. parents through to the root, with added metadata. The boundary-tree is structured as follows:

- array; the array of all max-tree nodes in the border
- offset; an array of five offsets: for north, east, south, west and ancestors respectively; north is at always 0, just for convenience
- size; the number of elements in the boundary
- borderparent; an array with the index of this node in the boundary-tree and a pointer to which boundary-tree of the parent of this node belongs
- merged; an array of booleans of size P, indicating which processors are merged into this boundary-tree

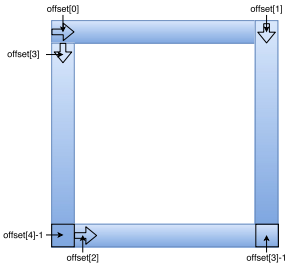


Fig. 2: Mapping of boundary to the 1D array

Building the boundary-tree The algorithm goes as follows: add each side: north, east, south and west. When the width or height of the tile is x, each side contains $x-1$ nodes, as the final node of that side is contained in the next side (in the first side for the last side). We administer the offsets of the sides in the array when building the tree. The mapping to the one-dimensional array is shown in Figure 2.

After the four sides are added, all ancestors of the nodes in the sides are added. The theoretical upper limit of the boundary including ancestors is: $(G-1)*(L/2)$ where G is the number of grey levels, and L is the length of the border. The worst case occurs only when half the boundary pixels are local maxima, and are at grey level $G-1$. We allocate the upper limit and shrink afterwards. For each node, we traverse the tree from boundary nodes to root. To make sure we add every node only once, we flag the nodes when adding them. For each ancestor we distinguish three cases:

1. The parent is $\perp$, continue with the next node
2. The parent is new, add it to the border, point from the current node to its parent, and continue with the parent
3. The parent is already in the border node, point from the current node to its parent, and continue with the next node

After all ancestors are added we shrink the array to its final size.

2.3 Merging tiles

The process for merging two tiles starts by sending the boundary-tree of a tile to its neighbour. Merging the nodes is done in a similar fashion to the sequential memory algorithm; but needs some more administration on the place of the nodes: is the node in the boundary-tree or in the tile itself. The process is as follows: for each node in the border, traverse the tree until you are at the bottom. The algorithm is shown in Algorithm 1, which is near identical to that in [8].

While merging two nodes in two trees, there basically are three options.

1. The parent of the current node to merge is $\perp$; in this case we are done with the traversal for this branch; continue with the next node in the border.

2. The parent of the current node to merge is not in the merged tree yet; add this node by accumulating the area of this node, and point to its parent; then continue with the parent.
3. The parent of the current node to merge is already in the merged tree. Add the accumulated area of this node to that node, and point to it. Then traverse its parents to add the accumulated area.

Algorithm 1 Merging attributes of two adjacent boundary-trees. Function LEVROOT(x) returns the index of the level-root of x, PAR(x) the level-root of the parent of x. Function FUSE is called once for each pair of adjacent nodes along the common border of the two boundary-trees.

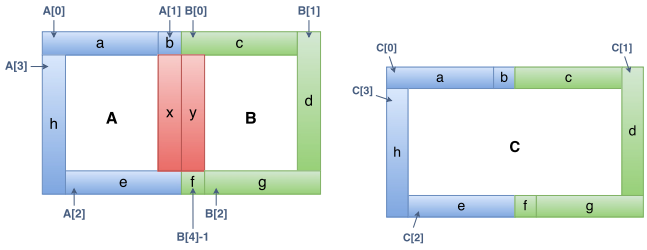
```
procedure FUSE(MaxNodeIndex x, MaxNodeIndex y )
    a ← 0
    b ← 0
    x ← LEVROOT(x)
    y ← LEVROOT(y)
    if GVAL(x) < GVAL(y) then
        SWAP(x,y)
    while x ≠ y ∧ y ≠⊥ do
        z ← PAR(x)
        if z ≠⊥ ∧ GVAL(z) >= GVAL(y) then
            Add a to x
            x ← z
        else
            b ← a+AREA(x)
            a ← AREA(x)
            AREA(x) ← b
            PARENT(x) ← y
            x ← y
            y ← z
    if y =⊥ then                                   ▷ Traverse down x
        while x ≠⊥ do
            AREA(x) ←AREA(x) +a
            x ← PAR(x)
```

We are done when all nodes in the border have been merged with the tree in the tile. We then need to propagate the retrieved information about the area of each merged node back to the tile that corresponds to the boundary-tree.

Alternating tree merges As a connected component can theoretically span multiple tiles, or even all tiles, we have to merge all tiles recursively to obtain the correct attributes. In order to keep the boundary-tree as small as possible, to avoid overhead, we merge the longest edge of the grid first. This process is repeated until all merges have been done.

Synchronizing attributes After merging the attributes of the two trees, only the process that has performed the merging has the merged information. We therefore have to synchronise the information to the processes that the information belongs to. We chose to store the process number of each node of the boundary in an array in the boundary-tree. This way, after merging the attributes of two trees, but before combining the two trees to one, we can update all tiles by sending them the nodes. To keep the overhead small, we only send the nodes that have changed.

Combining two trees In order to minimize the overhead when sending the boundary-tree, we grow the boundary-tree to a larger one by combining the nodes from both trees. Note that the nodes in the *middle* of the new tree, as shown in Figure 3, labeled x and y can be omitted. We combine the trees after merging and syncing their attributes. There is a difference in combining the trees horizontally vs. vertically; as the offsets of the nodes are different.



(a) Labeled parts and their offsets

(b) The resulting tree, with new offsets

Fig. 3: Combining two boundary-trees horizontally

When combining two boundary-trees horizontally, we distinguish different parts. The labels are shown in Figure 3a. An overview of how the nodes are placed in memory, using the labels from Figure 3a is shown in Figure 4. The vertical merge is done in an equivalent manner.

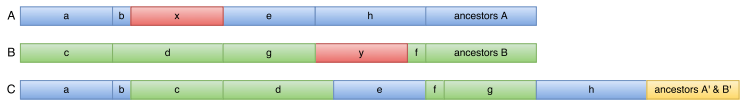


Fig. 4: Memory layout of the two boundary-trees A and B, combined in C

Adding ancestors After the nodes on the border of the area spanned by two boundary-trees has been extracted and put into a new boundary-tree C, we need

to add all relevant ancestors from the two original boundary-trees A and B. This process is very similar to adding ancestors when creating a boundary-tree out of a local max-tree, but now we do not retrieve the nodes from the local max-tree, but from the boundary-trees A and B. The algorithm is shown in Algorithm 2.

For each node in the border of the new boundary-tree we check the original trees A and B for ancestors. We start by checking the origin of each node (that is: the node in A or B, which has been copied to C) for parents. If there is no parent of the current node, we can continue with the next node in the border of C. If there is a parent, which is already in C, we can point there from our current node and we can continue with the next node in the border of C. Otherwise the parent is new, we add it to C and point there from the current node. Then we continue with this parent, and repeat the process.

Algorithm 2 Adding ancestors. ORIGIN(n) indicates for each boundary node in C with which node in input boundary-trees A or B it corresponds; PARENT(n), denotes the parent of n; INDEX(n) returns the index of n.

```
procedure AddAncestors(BoundaryTree A, BoundaryTree B, BoundaryTree C)
    s ← size of boundary of C                    ▷ Store the initial size of the boundary
    offset[4] ← s                                ▷ Store the starting point of the ancestors
    Reserve upper limit of memory
    for all node n in border of C do             ▷ Process all nodes
        p ← parent( origin(n) )
        curnode ← index(n)
        while true do
            if p =⊥ then
                BorderParent(curnode) ←⊥
                Continue with next
            if inBorder(p) then                  ▷ Of C
                BorderParent(curnode) ← index(p)
                Continue with next
            else                                 ▷ Add parent
                BorderParent(curnode) ← index(p)
                n ← p
    Shrink array to required size
```

Obtaining attributes of the full tree Combining the algorithms described in this chapter we can obtain the attributes for each local max-tree as if it was built in one large max-tree:

- Build a local max-tree for each process
- Build a boundary-tree for each local max-tree
- Recursively follow a pattern where vertical and horizontal merges/combinations are done based on the size of the grid of tiles

  - Send/receive the boundary-tree following that pattern
  - The receiving process merges the attributes and updates parent relations of two trees
  - Then sends back the nodes with changed attributes and parent pointers
  - Both the sending and receiving process apply the changes on their local max-tree
  - The receiving process combines the two boundary-trees
- Continue until all boundary-trees (and therefore their attributes) are merged

After merging and combining all boundary-trees, all local max-trees contain nodes with their updated attributes. Therefore we can use the same filtering process as in the shared memory solution [8] and apply it to each local max-tree. We end up with the filtered tiles and we can write them to files. Merging the images is considered a trivial post-processing step.

3 Results

In this section we show the results of executing the application. As a first essential test we verified that the output of the algorithm remained the same, regardless of the number of MPI processes used (data not shown).

(a) Detailed input (b) Output with $\lambda = 25$

Fig. 5: Example of execution with real life data

Initial tests were performed on the Zeus compute server: a 64-core AMD Opteron Processor 6276 with 512 GB of RAM. The main tests were on up to 128 standard nodes of the Peregrine cluster of the Center for Information Technology

of the University of Groningen. The standard nodes have 24 Intel Xeon 2.5 GHz cores and 128 GB RAM, and are connected by a 56 Gb/s Infiniband network. We used only a single core per node, to simulate worst case-message passing conditions. We used a 30000×40000 pixel grey value VHR aerial image of the Haiti earthquake area. A small cutout is shown in Figure 5.

We performed two experiments with this image. For Experiment 1, we split up the image into up to 128 tiles. Because the image is small enough that the max-tree to fits into memory of a single node of the cluster, this allowed us to measure speed-up $S(P)$ in the usual way as $t(1)/t(P)$, with P the number of processes, and $t(1)$ is the wall-clock time for one process, and $t(P)$ the wall-clock time for P processes. The disadvantage of this approach is that we can only work on comparatively small images. We therefore used multiple copies of the 30000×40000 image to generate a series of synthetic images of sizes 2, 4, 8, 16, and 32 times bigger than the original. Here we estimate speed-up as

$$S(P) = Pt(1)/t(P)$$

because the images scale in size with the number of processes in Experiment 2. In all measurements we took the minimum wall-clock time from ten runs, to minimise the effect of other programs running on the cluster. Given that the Peregrine is a "production" HPC cluster, multiple jobs may share a node, so timing may vary.

Table 1: Timings for the Haiti image

	Experiment 1			Experiment 2			
Tiles	Time (s)	$S(P)$	Speed (Mp/s)	Size (GB)	Time (s)	$S(P)$	Speed (Mp/s)
1×1	331.09	1.0	3.62	1.2	331.09	1.0	3.62
1×2	163.08	2.03	7.36	2.4	357.65	1.85	6.71
2×2	80.25	4.13	14.95	4.8	362.64	3.65	13.24
2×4	45.06	7.35	26.63	9.6	369.30	7.17	26.00
4×4	23.54	14.06	50.98	19.2	381.16	13.90	45.33
4×8	15.04	22.01	79.79	38.4	393.99	26.89	97.46
8×8	10.39	31.87	115.50	76.8	395.27	53.67	194.3
8×16	8.39	39.65	143.71	153.6	409.87	103.52	374.75

Looking at the speedup in Table 1 we can see that the algorithm performs rather well in terms of scaling, achieving roughly 75% efficiency on 32 processes. For Experiment 1, we see a near linear speed-up up to 8 processes, after which some slowdown is observed, dropping to about 50% at 64 cores, and some 25% at 128 cores. Note that at these large values of P the size of the max-trees of each tile is quite small compared to the boundary-tree size. This increases the message-passing overhead. As a variant, we ran the 32-process runs of Experiment 1 on

16 cores of 2 nodes to reduce message-passing overhead, but we could not notice any significantly different timings. In Experiment 2, speed-up is clearly better for large P on the cluster: at $P = 128$ we get no less than 103× speed-up. The better results in Experiment 2 may be due to the fact that for large tiles the communication load is small compared to the compute load. In a final experiment on the Peregrine cluster, we combined experiment 1 and 2, using a 4×4 mosaic of the 1.2GB image, and subdividing the resulting 19.2GB image into 256 tiles. Using 16 cores on 16 nodes each, a performance of 302 Mpixel/s was obtained, or about 83 × speed-up.

The timings on the Zeus server, which could only run Experiment 1, were similar, and achieved up to 50.6× speed-up on 64 cores on a 3.88 Gpixel satellite image. Several short tests where run on a single, 48-core, 2.2 GHz Opteron node of a cluster at DigitalGlobe, using three panchromatic WorldView 2 and WorldView 3 satellite images: a 31888 × 31860 pixel image of the city of New York, USA at 0.5m spatial resolution; a 35840 × 35840 pixel image of Madrid, Spain, at 0.5m spatial resolution, and a 37711 × 54327 pixels image of Cairo, Egypt at 0.3m spatial resolution. Wall-clock times on these images were 36.8 s, 51.6 s, and 87.0 s respectively, including I/O, very much comparable to the results obtained on the 64-core machine with similarly sized images. This indicates that the results were not due to some special configuration of the image used.

4 Conclusion

In this paper we showed how to approach a distributed memory algorithm for connected filters, building on the shared-memory version. The key notion is to build a forest of modified max-trees, rather than a single max-tree. This is possible through the use of the boundary-tree, which allows selective, recursive exchange of just the necessary data to modify the max-trees of each tile to yield exactly the same result as would be obtained by filtering the max-tree of the entire image. With minor adaptation, this approach also allows processing images with max-trees larger than local memory. We have shown that the algorithm scales well on different architectures, certainly up to 128 processes for large data sets.

Several improvements or extensions of the current work can be made. First of all the algorithm could be extended to 3-D volumes or videos, where you would have to redefine the neighbour function. This does have consequences for the complexity, as the size of the boundary-tree of a 3-D volume is $\mathcal{O}(GN^{\frac{2}{3}})$.

A key issue is the reliance on a limited grey-level range to keep the communication load manageable. Matas et al [10] suggest using a fast linear max-tree algorithm that is efficient regardless of the bit depth of the image, but it uses the same merging algorithm as [8]. The problem is that the mergers of different trees scale linearly with the number of grey levels in the worst case, so bit depths beyond 16 bits require a different approach. Indeed, first merging lines in an image rather than using tiles or larger blocks maximizes the number of merges needed, exacerbating the problem.

Even for the current implementation, all the conclusions must of course be more thoroughly validated on a much more extensive, real-world data set. To that end, we are currently improving the I/O structure to accommodate more file formats, and easier integration within remote-sensing tool chains. Extensions for pattern spectra, and differential morphological and area profiles, and the use of other attributes will also be developed. The source code will be made available.

Acknowledgment. We would like to thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Peregrine high performance computing cluster. The 64 core Opteron machine was obtained by funding for the HyperGAMMA project from the Netherlands Organisation for Scientific Research (NWO) under project number 612.001.110.

References

1. Breen, E.J., Jones, R.: Attribute openings, thinnings, and granulometries. Computer Vision and Image Understanding **64**(3) (1996) 377–389
2. Salembier, P., Wilkinson, M.H.F.: Connected operators. Signal Processing Magazine, IEEE **26**(6) (2009) 136–157
3. Salembier, P., Oliveras, A., Garrido, L.: Antiextensive connected operators for image and sequence processing. Image Processing, IEEE Transactions on **7**(4) (1998) 555–570
4. Najman, L., Couprie, M.: Building the component tree in quasi-linear time. Image Processing, IEEE Transactions on **15**(11) (2006) 3531–3539
5. Pesaresi, M., Benediktsson, J.: Image segmentation based on the derivative of the morphological profile. Mathematical morphology and its applications to image and signal processing (2002) 179–188
6. Urbach, E.R., Roerdink, J.B.T.M., Wilkinson, M.H.F.: Connected shape-size pattern spectra for rotation and scale-invariant classification of gray-scale images. Pattern Analysis and Machine Intelligence, IEEE Transactions on **29**(2) (2007) 272–285
7. Ouzounis, G.K., Soille, P.: Differential area profiles. In: Proceedings of the 2010 20th International Conference on Pattern Recognition, IEEE Computer Society (2010) 4085–4088
8. Wilkinson, M.H.F., Gao, H., Hesselink, W.H., Jonker, J.E., Meijster, A.: Concurrent computation of attribute filters on shared memory parallel machines. Pattern Analysis and Machine Intelligence, IEEE Transactions on **30**(10) (2008) 1800–1813
9. Carlinet, E., Géraud, T.: A comparative review of component tree computation algorithms. IEEE Transactions on Image Processing **23**(9) (2014) 3885–3895
10. Matas, P., Dokladalova, E., Akil, M., Grandpierre, T., Najman, L., Poupa, M., Georgiev, V.: Parallel algorithm for concurrent computation of connected component tree. In: International Conference on Advanced Concepts for Intelligent Vision Systems, Springer (2008) 230–241
11. Wilkinson, M.H.F., Soille, P., Pesaresi, M., Ouzounis, G.K.: Concurrent computation of differential morphological profiles on giga-pixel remote sensing images. Mathematical Morphology and Its Applications to Image and Signal Processing (2011) 331–342