ICPRAI 2018 SI: Distributed Component Forests in 2-D: Hierarchical Image Representations Suitable for Tera-Scale Images.

Simon Gazagnes and Michael H.F. Wilkinson

Bernoulli Institute, University of Groningen

Groningen, The Netherlands

*

Abstract

The standard representations known as component trees, used in morphological connected attribute filtering and multi-scale analysis, are unsuitable for cases in which either the image itself, or the tree do not fit in the memory of a single compute node. Recently, a new structure has been developed which consists of a collection of modified component trees, one for each image tile. It has to date only been applied to fairly simple image filtering based on area. In this paper we explore other applications of these distributed component forests, in particular to multi-scale analysis such as pattern spectra, and morphological attribute profiles and multi-scale leveling segmentations.

Keywords: Mathematical morphology, attribute filter, multi-scale representation, distributed-memory computation

1 Introduction

As image sizes get larger, so too do hierarchical representations of these images. One class of hierarchical image representation is from connected filtering, referred to as min trees and max trees [20] component trees [9], or opening and closing trees [26]. These tree structures form a

*The position of S. Gazagnes was funded from a grant by the Centre for Data Science and Systems Complexity, University of Groningen. 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.

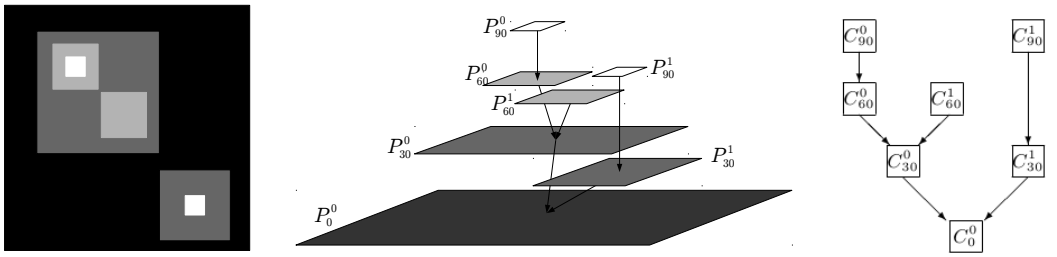
compact representation of all the connected components of all threshold sets in an image. They find use in various applications, such as attribute filtering [3, 13, 20, 23], computation of pattern spectra [24], or morphological profiles [1, 18, 31], and visualization [27].

The computational complexity of algorithms to construct these tree structures is also modest: either $O(GN)$, with G the number of grey levels in the regular 8-16 bit per pixel case or $(N \log N)$ in the 32-bit or more integer or floating point case [2, 15]. Furthermore, *shared-memory* parallel algorithms for computation of these trees [14, 29], and subsequent postprocessing have been developed [30, 31]. On fairly modest compute servers they can handle images up to a few gigapixel at most.

This is despite the attractive property of these tree structures, that they form a very memory-efficient multi-scale representation of an image, which can represent all scales present in the image, unlike classical image pyramids, or wavelets, which discretize scale in some way. However, their storage cost is proportional only to the image size, not the number of desired scales, such as in e.g. a Gaussian scale space. A typical component tree can grow to about 20-40 times the size of the image in storage. In practice, in parallel computation we pre-allocate this maximum, both because it can simplify the algorithm, and because it avoids the need for (locking) memory allocation during tree construction [14, 29].

Given that many imaging modalities routinely acquire images in the order of tens or hundreds of gigapixels, and tera-scale images occur in both remote sensing and astronomy, there is a need for a representation capable of handling these hierarchies in a distributed manner. Very recently, such a method has been developed in the form of *distributed component forests* (DCFs) [10]. The algorithm presented only allowed application of morphological area openings [5, 25], and could only handle a 8-bit-per-pixel grey-scale images. Here, we extend the approach and explore how to implement multi-scale analysis methods such as pattern spectra and morphological profiles using distributed component forests.

Besides astronomy and remote sensing, these tools could be used in large variety of fields. In medical imaging, vessels extraction and analysis, which was already shown to be successful in [19, 27], could be extended to very high resolution X-ray or MRI imaging. The comparatively new field of virtual nanoscopy is another important area of application. Faas et al [6] report the capture of a 281 Gpixel image of a complete section of a zebra-fish embryo, at 1.6 nm resolution. Multi-scale analysis is essential for understanding the relationships between structures at vastly different scales, ranging from structures within organelles, to macroscopic features such as complete



Input image Thresholded sets(peaks) Max-tree

Figure 1: A simple grey-scale image, the foreground components of each threshold set, also know as *peak components*, and the resulting component tree, which is referred to as a *max-tree* in this case.

organs.

The paper is organized as follows: first we discuss connected attribute filters. We then turn to component trees and how they can be used to implement these methods efficiently. After this, we discuss parallelization strategies in shared and distributed memory. Finally, we discuss two multi-scale analysis tools, i.e., morphological profiles and pattern spectra, and how they can be implemented in this framework.

2 Connected Filters and Multi-Scale Tools

Connected filters [21] are a type of morphological filter based on manipulation of *flat zones*, which are connected zones of constant grey level or colour of maximal extent. In the binary and grey-scale cases, a simple, but powerful approach is that of *attribute filters* [3]. In the binary case, an attribute opening computes some increasing property, or *attribute* of all connected foreground components, and removes all those for which the attribute falls below some threshold λ. The simplest case is the area opening [5, 25]. In the grey-scale case, we can in principle compute all threshold sets, apply the binary attribute filter to each threshold set, and stack the results up to obtain a grey-scale result. In the case of the area opening, the result resembles filing off the top of every bright peak in the images until its area is at least λ in area. In practice faster methods exist, which will be discussed in Section 3.

These filters can be extended to non-increasing attributes, like elongation, and also vector attributes [23], which allow enhancement or detection of a range of structures.

3 Component Trees and Forests

The trivial implementation of attribute filtering consists in thresholding the image explicitly and performing connected component analysis at each threshold is very inefficient. However, it is readily seen that the connected foreground components of each threshold set except the lowest are nested in precisely one connected foreground component at a lower level. Therefore, they can be organized into a tree structure, as seen in Figure 1.

We can distinguish two types of component trees: *max-trees*, as depicted in Figure 1, in which each node represents a foreground component at some threshold set, and *min-trees* in which the nodes represent background components at each threshold set, and where the nesting relationship is reversed. Min-trees are often computed as max-trees of the inverted image. For the low dynamic range images in which we are interested here, the most efficient algorithm to compute these trees is by flooding from the lowest grey levels in a depth-first manner, using the algorithm of [20]. For a recent review of algorithms see [4].

There are various ways in which we can represent component trees. Here we use the representation from [29]. In this representation we allocate an array of nodes the size of the image, which is the worst-case number of nodes for any image. In practice, the number of component tree nodes is smaller, so multiple pixels will in general belong to the same node. All pixels in the node have parent pointers that point to a single pixel that represents the entire node. This is called the *canonical element*, or level root. The level root in turn points to a pixel in the parent node in the component tree. Level roots are readily distinguished from all others because they point to a node representing a pixel of a different grey level than their own. The overall root of the tree has a pointer with a special value: $\perp$. Any manipulation of the component tree can be performed by appropriate manipulation of the level roots in this representation.

3.1 Parallel Computation

To perform parallel computation for low-dynamic-range images, we can use the algorithm from [29]. The image is first divided into strips or tiles, one to each processor. After this, local component trees are computed, using a modification of the algorithm of [20], where the labeling of nodes is done in such a way that it allows merger of nodes using union-find [22]. For 16-bit images, we use the algorithm of [28] which performs better for this dynamic range. After this, neighbouring tiles or strips are merged hierarchically, and a single component tree of the image is constructed. During the final merge, carried out by the thread with rank zero, the entire component tree

and image must be accessible to that thread. Once the final merge has been performed, each thread can proceed to filter the image independently [29], or perform more complex operations like computation of differential attribute profiles [31], or pattern spectra [30].

3.2 Distributed Component Forests

In the distributed case, we cannot directly use the above strategy, if the size of the component tree is such that it will not fit into the memory of the processor of rank zero. Quite apart from that limitation, communicating entire component trees becomes prohibitive. In a recent work Götz et al [8] implemented a method to perform a parallel computation of component trees on distributed memory machines. They first compute the local component trees of each tile, and then use a different approach based on a specific data structure, the *tupples*, to perform a parallel correction of the parents in each local component tree. This method does not perform any attributes computation, and is therefore not suited for computation of pattern spectra or morphological profiles, without significant extension. Therefore a new approach is needed.

We first split the image up into tiles, and built local component trees for each tile. We now have a hierarchical representation of h-connected components of each tile. Pixels p and q at grey level $\geq h$ are h-connected if there exists a path *within the tile* from p to q through pixels of grey level $\geq h$. We now need to correct these by modifying the connectivity along the boundary, such that the pixels p and q are considered h-connected if there exists a path *within the entire image* from p to q through pixels of grey level $\geq h$. Each node in the modified max-trees then represent h-connected components intersected with the domain of the tile. Furthermore, the attribute assigned to each node must equal that of the corresponding h-connected component of the entire image.

To do this, *boundary trees* consisting of the subsets of nodes that touch the boundary of a tile are communicated to the neighbours. If the number of grey levels G is modest, the boundary tree will be much smaller than the component tree of the entire tile. An upper bound of the size is simply $G\#(\delta B)/2$, where $\#(\delta B)$ indicates the cardinality of the boundary δB of the tile B. For a tile of $20,000^2$ and $G = 256$, this works out as $1.024 \cdot 10^7$ nodes maximally, compared to a worst case component tree size of $\#(B) = 4.0 \cdot 10^8$. Assuming square tiles, the boundary tree size scales as the square root of the tile size, whereas the component tree itself scales linearly with tile size. Contrary to the approach defined in [10], the current implementation never allocates this maximal size in the memory. Indeed, in practical cases, the final size of the boundary tree never exceeds a few percents of the upper bound limit. Consequently, we implemented a dynamic

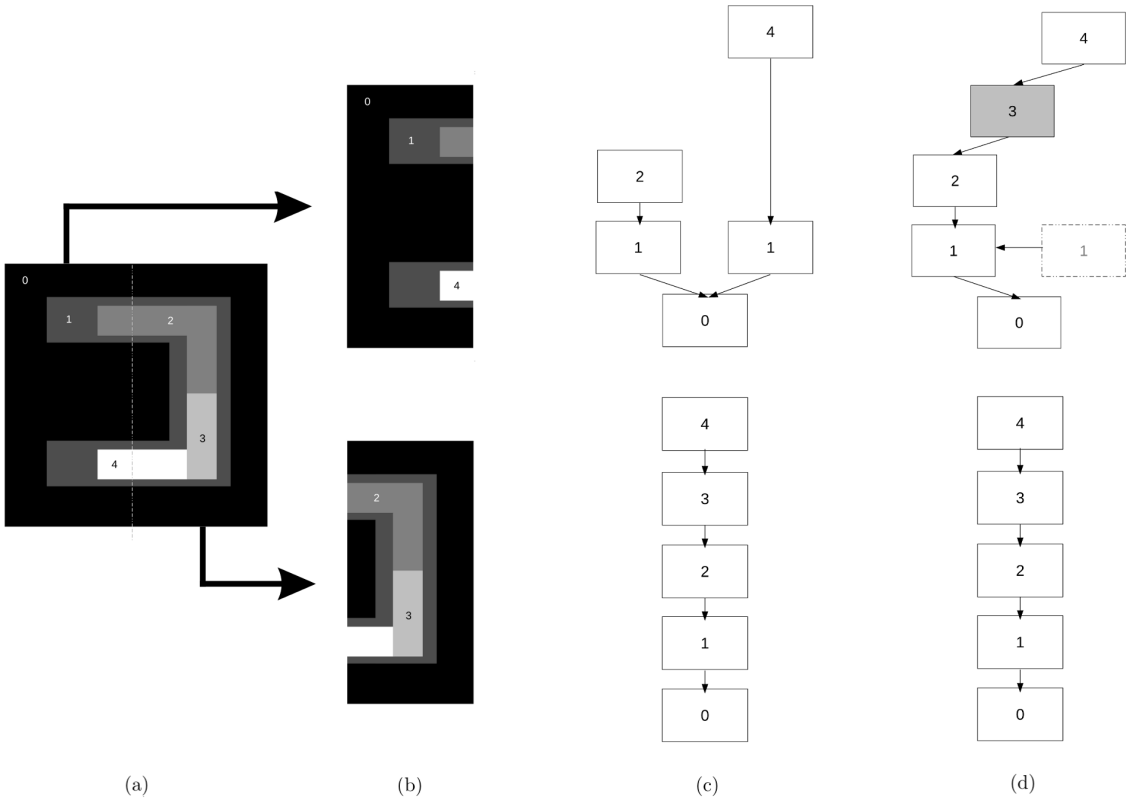


Figure 2: An example of the construction of a distributed component forest: (a) a simple grey scale image, with the dashed line indicating the border between tiles; (b) the two separate tiles, processed independently; (c) the max-trees corresponding to each tile; (d) the final updated trees forming the distributed component forest. The grey node on the top tree is a new node, whereas the node with the dashed boundary is no longer a level root, and therefore no longer represents a max-tree node.

memory allocation procedure; we first allocate a lower initial size, called `size_alloc`, typically chosen as the size of the tile boundary (given by $2 \times (W-1) + 2 \times (H-1)$, where W and H are respectively the width and the height of the 2-D tile). When the current number of nodes in the boundary tree reaches this limit, the memory allocated to the tree is expended to a larger size, such that the new allocated size is $1.5 \times$`size_alloc`. This increased the memory efficiency of the method, and also had positive repercussions on the computational time.

In [10], boundary trees were created locally, merged and the modifications directly sent back into the local component trees. However, this approach was found incomplete in certain cases, and the new procedure `correct_borders` is presented in Algorithm 1. First step consists of creating the local boundary tree from the local component tree of the tile. After this, the boundary trees from neighbours tiles are merged and combined into a single structure corresponding to the boundary tree of the merged tiles. The complete description of the merging pattern can be found in [10],

but we recall the important steps here. For each process, the variable n_merges stores the number of merges it will perform. For example, in a 2 by 2 grid (image divided in 4 tiles), the process 0 has $n\_merges = 2$, process 2 has $n\_merges = 1$, and the processes 1 and 3 have $n\_merges = 0$. During the first step, process 0 will combine the boundary tree of the tiles 0 and 1, and process 2 the tiles 2 and 3. In a second time, process 0 will combine the boundary tree of the tiles $\{01\}$ and $\{23\}$. As shown in Algorithm 1, each merge consists of three steps for the process performing it: receiving the boundary tree from the neighbour process, merging the nodes of the two boundary trees and then combining them into a new tree that only includes the level roots of the merged structure. This means the combined tree stores a smaller number of nodes than the worst-case upper limit which contains the total number of nodes in the two merged trees. The merging process is then repeated with the new combined trees until all merges have been done. Once the merging phase is done, all the changes in the nodes and their attributes need to be propagated to the local max-tree trees in such a way that filtering individual trees in the forest yields exactly the same result as filtering the entire component tree. We implemented a procedure that starts from the last tree combined and updates successively all the structures from the top to the bottom until we reach the local boundary trees. The function `update` propagate the most recent node attributes from the combined tree to the merged trees and reassign their parents such that both trees are independent again. This reconnection is done using the following simplified methodology: for each node x in the merged structure, we check if x and its most recent parent are from the same boundary tree. If yes, x is connected to this parent, otherwise the parent node is added to the boundary tree of x. Once the two trees have been updated, one is sent back to the process that it was received from. Both newly updated trees will in turn be used to update the former boundary trees in their respective processes until all the boundary trees in all processes have been corrected. To perform this procedure, each process need to store the boundary trees that he has received, merged and combined during the first phase. Each process then stores $2 \times n\_merges + 1$ boundary trees, where n_merges is the number of merging steps performed by the process and $+ 1$ stands for the local boundary tree in each tile. The local component tree is then corrected using the updated boundary tree of the local tile. Figure 2 shows an example of the evolution of the distributed component trees before (b) and after (c) the *correct_borders* procedure detailed in Algorithm 1. This approach may seem memory costly as we need to store the successive boundary trees, and this number increases with the number of processes. However, we show in Section 4 that this technique has a limited memory cost, due to the relative small size of the boundary trees

with respect to the local component tree of the tile.

Algorithm 1 Successive merging and correction of boundary trees

```
procedure correct_borders (rank, n_merges : integer; maxtree : Max-Tree;
                           var boundtree : Boundary-Tree[0..2n_merges])
    boundtree[0] ← create_boundary(maxtree);
    neighrank ← 1;
    i ← 0;
    j ← n_merges;
    while j > 0 do
        boundtree[i + 1] ← receive_boundary(rank + neighrank);
        merge(boundtree[i], boundtree[i + 1]);
        boundtree[i + 2] ← combine(boundtree[i], boundtree[i + 1]);
        neighrank ← 2neighrank;
        i ← i + 2;
        j ← j − 1;
    end while
    if rank ≠ 0 then
        send_boundary(rank − neighrank, boundtree[i]);
        boundtree[i] ← receive_updated_boundary(rank − neighrank);
    end if
    while j < n_merges do
        boundtree[i − 2] ← update(boundtree[i − 2], boundtree[i]);
        boundtree[i − 1] ← update(boundtree[i − 1], boundtree[i]);
        send_updated_boundary(rank + neighrank, boundtree[i − 1]);
        free_boundary(boundtree[i])
        free_boundary(boundtree[i − 1])
        neighrank ← neighrank/2;
        i ← i − 2;
        j ← j + 1;
    end while
    maxtree ← correct_maxtree(maxtree, boundtree[0]);
end procedure
```

The algorithm presented in [10] only allowed application of area openings [25]. Here, we explore how to implement multi-scale analysis methods such as pattern spectra and morphological profiles using distributed component forests.

3.3 Granulometries and Pattern Spectra

Apart from simply filtering an image, multi-scale analysis using component trees is an efficient way of extracting information from images. After all, given that the component tree contains information of all connected components at all grey levels, and their attributes, *any* filtered version based on those attributes can be computed from it, along with any statistics on the sizes and shapes of components in the image.

A classical morphological tool for multi-scale analysis are pattern spectra [12], which are based on granulometries. Granulometries are totally ordered sets of openings $\{\gamma_r\}$, with r from some

index set I, such that

$$\gamma_s(\gamma_r(f)) = \gamma_{\max(r,s)}(f) \tag{1}$$

A pattern spectrum is obtained by computing how much of the image content is removed by each consecutive filter in a granulometry. Let $\gamma_0(f)$ be the original image, a discrete pattern spectrum $S(f)$ can then be calculated as

$$S(f)_r = \sum_A \Big(\gamma_{r-1}(f) - \gamma_r(f) \Big), \quad \text{r=1,2,..N} \tag{2}$$

where $S(f)_r$ is the r-th bin of the patter spectrum, and the sum is taken over the image domain A. In principle, this requires N openings to be computed, followed by N differences, and summations over the images.

Computing pattern spectra based on attribute filters can be done far more efficiently, simply by analysing the component tree. We need to visit each node, represented by its level root, in the component tree, compute which bin in the pattern spectrum it belongs to, and add the product of the area of the node and the grey-level difference to its parent to that bin. In the shared-memory parallel case, each process does this with a private copy of the pattern spectrum, only for those level roots in its tile, and these are added together in the final stages [30]. In the distributed case, things are slightly more complicated, because each node in the complete component tree may be represented by multiple level roots of different modified component trees in the DCF. This means that the above process will lead to double counting of component tree nodes that span more than one tile.

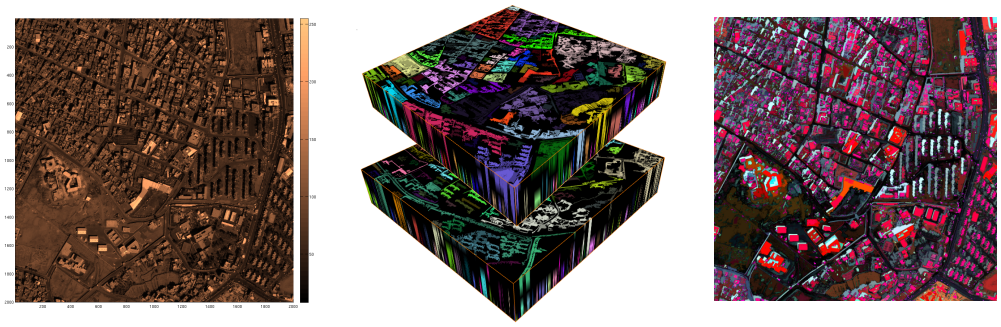
The solution is fairly simple: we need to add only that part of the component tree node that intersects the local tile to the private copy of the pattern spectrum. No extra computation is needed, as we compute this area anyway while building local component trees. Before starting the merge process, we simply store the computed area in an extra field `privateArea`. During the merge process, this field is unchanged.

At the end of the merge, only original level roots of the component tree of the tile have a non-zero `privateArea`. After the merge and update stage, they may no longer be level roots, but they must be accounted for. Therefore, we scan all nodes within the area of the tile T_p of process p. For every node v that has a non-zero `privateArea`, we find the level root of v and determine the appropriate scale with in the pattern spectrum the node belongs to. After this, the product of the `privateArea` field and the difference in grey level with the parent of the level root is added

Algorithm 2 Computation of the area pattern spectrum for each tile, for process p. Function *Par* returns the level root of the parent of the current node.

```
procedure AreaPatternSpectrum (Tp : Tile; maxtree : Max-Tree;
                               var spectrum : integer[numscales];
                               lambda : integer[numscales])
    for all v ∈ Tp do
        if maxtree[v].privateArea ≠ 0 then
            scale ← findScale(maxtree[get_levelroot(v)].area, lambda);
            privateArea ← maxtree[v].privateArea
            parent ← get_levelroot(maxtree[get_levelroot(v)].parent);
            spectrum[scale] ← spectrum[scale]+
                (maxtree[u].gval − maxtree[parent].gval) ∗ privateArea;
            u ← parent;
            while not IsRoot(maxtree[u].parent)
              and maxtree[u].gval ≠ node[v].gval_par do
                scale ← findScale(maxtree[u].area, lambda);
                parent ← get_levelroot(maxtree[u].parent);
                spectrum[scale] ← spectrum[scale]+
                    (maxtree[u].gval − maxtree[parent].gval) ∗ privateArea;
                u ← parent;
            end while;
        end if;
    end for;
end procedure

procedure findScale (var area : integer; lambda : integer[numscales])
    upper ← numscales − 1;
    lower ← 0;
    if area ≥ lambda[upper] then
        return numscales
    end if;
    mid ← (upper + lower)/2;
    while mid ≠ lower do
        if area ≥ lambda[mid] then
            lower ← mid;
        else
            upper ← mid;
        end if;
        mid ← (upper + lower)/2;
    end while;
    return lower
end procedure
```



(a) (b) (c)

Figure 3: Example of two instances of a DAP vector field. The original image in (a); the DAP vector field with the first and last attribute zone at the bottom and top of each instance in (b) respectively. The top volume in each image corresponds to the opening, and the bottom to the closing instance. (c) The blended CSL triplet. Figure from [31]

to the appropriate bin. We then check whether this node has been reconnected to an other parent with higher grey level during the update phase. To do so, we use on extra variable `gval_par` that stores the grey value of the local parent at the end of the initial creation of the local component tree. For each node, we compare the grey value of its parent to the grey value of its initial parent given by `gval_par`. If they are different, we iterate up the tree towards the root, determining the correct bin from the area of the current node, and the grey level difference with its parent. This grey level difference is multiplied by the `privateArea` field of the node we started from, and the result is added to the appropriate bin. When all processes have computed their pattern spectra, we sum the private pattern spectra as before. The algorithm to compute each pattern spectrum for tile T_p is shown in Algorithm 2.

3.4 Differential Area Profiles

The Differential Area Profiles (DAP) is a multi-scale representation of an image. DAPs are related to pattern spectra in that they can be considered as a vector image in which each pixel of the original image is replaced by a vector representing a local pattern spectrum at that point. They can be computed as a series of top-hat filtered images, using a series of area openings of increasing area threshold λ. This contains all the bright features in the image. To extract the dark features, a series of bottom-hat filters based on area closings is used, and the two patterns spectra are concatenated in a single vector. An example is shown in Figure 3.

In many cases, it is not the DAP we are interested in, but a more accessible, 2D representation of the salient features in the DAP, derived from the so-called multi-scale leveling segmentation [18].

For each pixel, we determine the scale S (in terms of area) which has the highest amplitude to represent that pixel. We also record the amplitude C (for contrast) at that scale (S), and the original luminance L, forming the so-called CSL segmentation of that image, shown in Figure 3(c), with the three bands mapped to R, G, and B channels respectively.

Clearly, as we can compute area openings from a distributed component forest (DCF), it is trivial to use them to compute a series of area openings, and consequently difference the results to obtain the required stack of area top-hat images. Alternatively we can use the same component-tree filtering algorithm used in parallel computation of the CSL segmentation on each of the modified component trees in the DCF. The only modification needed is that we must only store data from the current tile, not the nodes added in the merging process. As each node contains a field indicating which process it belongs to, this is straightforward.

4 Results

We implemented the distributed memory algorithm based on the previous work of [10] and using the new *correct_borders* procedure detailed in Section 3.2. We then added the extra code needed to compute pattern spectra and CSL segmentation based on DAPs respectively. We used different data sets to test the performance of our method. The first image is a 30000×40000 pixels (1.2 Gigapixels) remote sensing image of Haiti, with 8-bit per pixel, and a small part of it is shown in Figure 4. The second image is a pan-sharpened, 47830 × 41629 (≈ 2 Gigapixels) remote-sensing image of the bay of Naples, with 8-bit per pixel, and is displayed in Figure 5. The third image is an astronomical image of the Andromeda Galaxy, captured with the NASA/ESA Hubble Space Telescope, of dimension 66312 × 21199 pixels (≈ 1.4 Gpixels) and originally 32-bit RGB. In order to process it, we combined the three R,G and B channels into a luminance channel using the formula $L = 0.2126R + 0.7152G + 0.0722B$. The resulting floating point image has then been re-quantized into two data sets of 8-bit and 16-bit per pixel. The original and 8-bits luminance image are presented in the Figure 6. Additionally, we generated a 16-bit version of the Haiti data set by multiplying all the pixel values of the 8-bits image by 255 and adding random values between 0 and 255 to fill the least significant bits. Our final set of bench-marking data includes three 8-bit and two 16-bit per pixel images.

The three original images have different structure patterns: while the astronomical data has a fairly smooth dark background with bright peaks, the Haiti remote-sensing image has a large number of individual structures, and the Naples bay has larger ones spanning over the whole

Figure 4: Cut into the remote sensing image of Haiti

image. Given these differences, we expect some variation in the performance of our method for each of them. We ran different experiments to study both the timings and memory consumption of our method. The experimental tests were performed on two different clusters: the Zeus compute server which includes a 64-core AMD Opteron Processor 6276 (2.3 GHz) with 512 GB of RAM, and the Peregrine cluster of the Center for Information Technology of the University of Groningen. This cluster has 162 *standard* nodes, with 24 Intel Xeon 2.5 GHz cores and 128 GB of RAM each. In all the figures presented in Section 4.1, the timings reported are the minimum timing value obtained over 10 runs. This value includes the local tree computation, the boundary correction and the filtering phase, but does not account for the delays in reading/writing the input/output images. Additionally, we used a slightly modified version of the code to keep track of the memory consumption during the execution of the morphological area opening. These results are presented in Section 4.2.

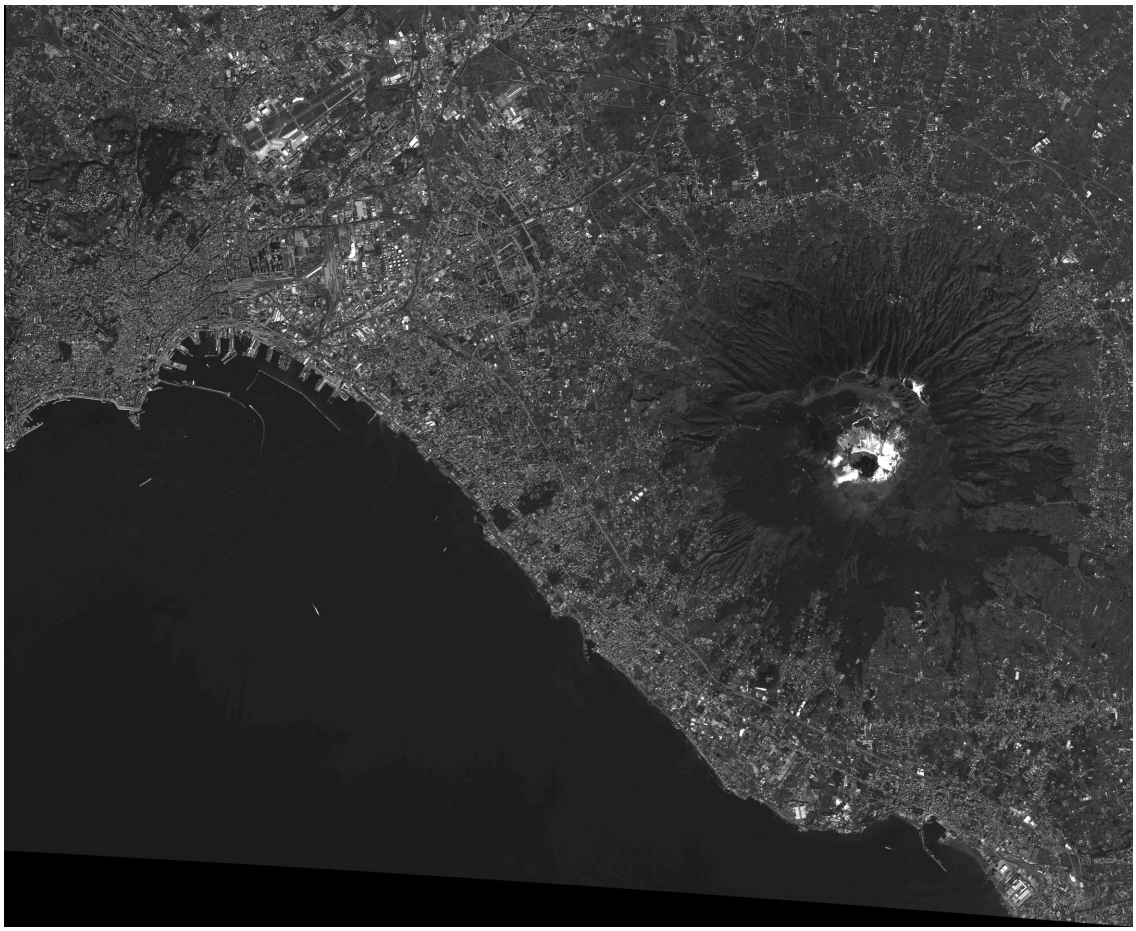


Figure 5: Pan-sharpened remote sensing image of the Bay of Naples

4.1 Timings and speed-up

Initial tests were performed on the three 8-bit per pixel data sets. Each image was cut in p tiles, where p is the number of processes. We measured wall-clock time for 1, 2, 4, ..., 64 processes (as the number of processes increases, the size of each tile decreases), and computed the ratio $t(1)/t(p)$, where $t(1)$ is the wall-clock time for one process and $t(p)$ the wall-clock time for p processes, as the corresponding speed-up. The Haiti and the NASA images were processed on Zeus using at maximum 64 processes, and the results are shown on Figure 7 and Figure 8 respectively. The Naples image was handled on Peregrine, using at maximum of 256 processes, and the results are plotted in Figure 9. We used 32 scales for the pattern spectrum and the CSL representation. Due to memory overload issues on Peregrine, we were not able to perform the pattern spectra computation for the Naples data set. The results show that the timings of the two multi-scale operations are slightly larger than the single scale filtering. As expected, computing the more

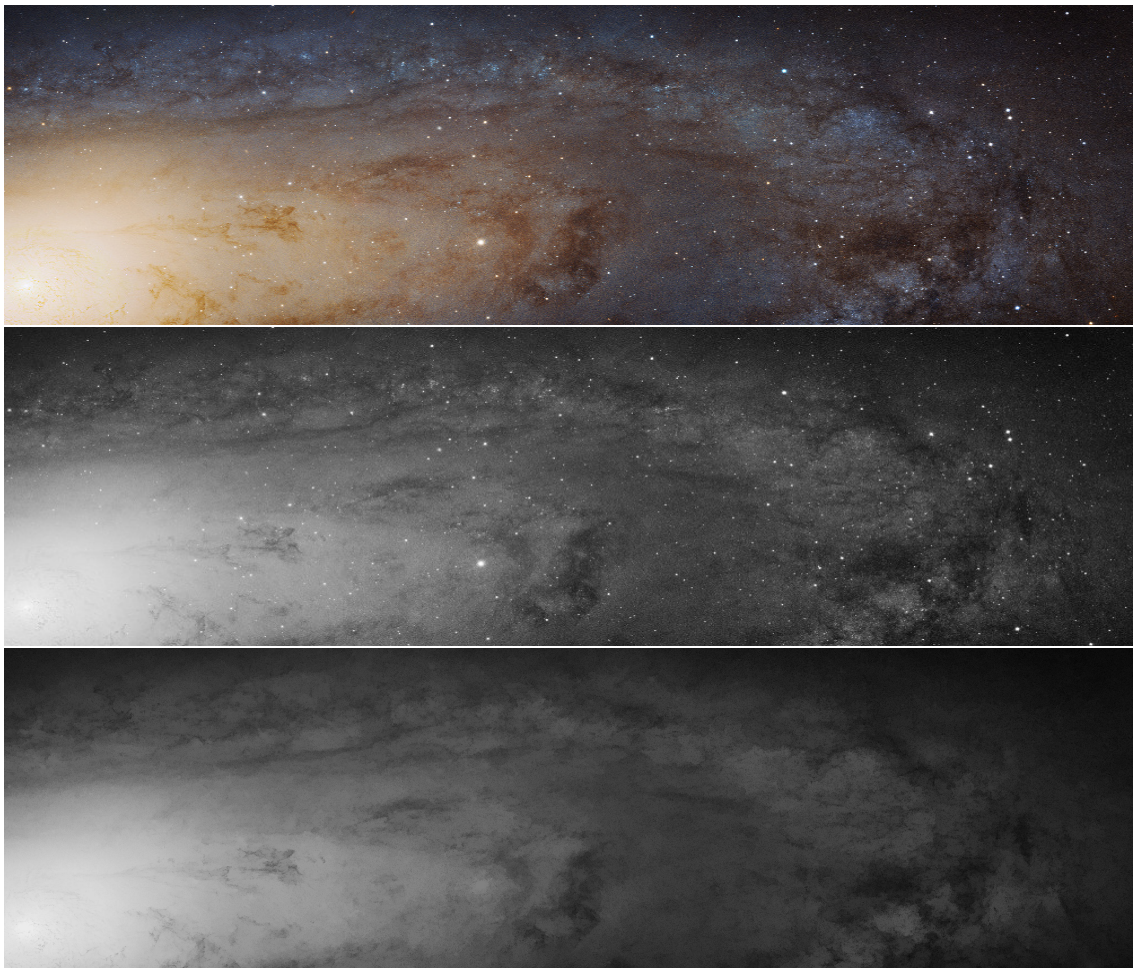
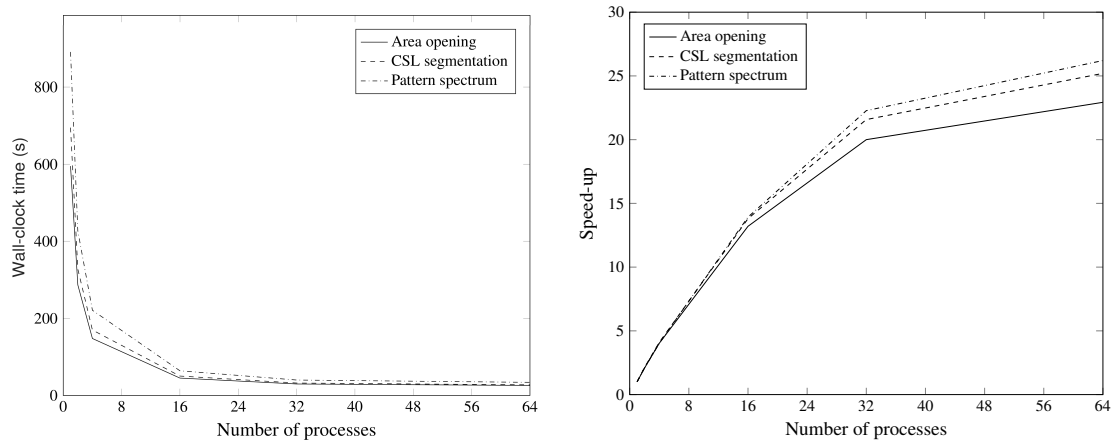


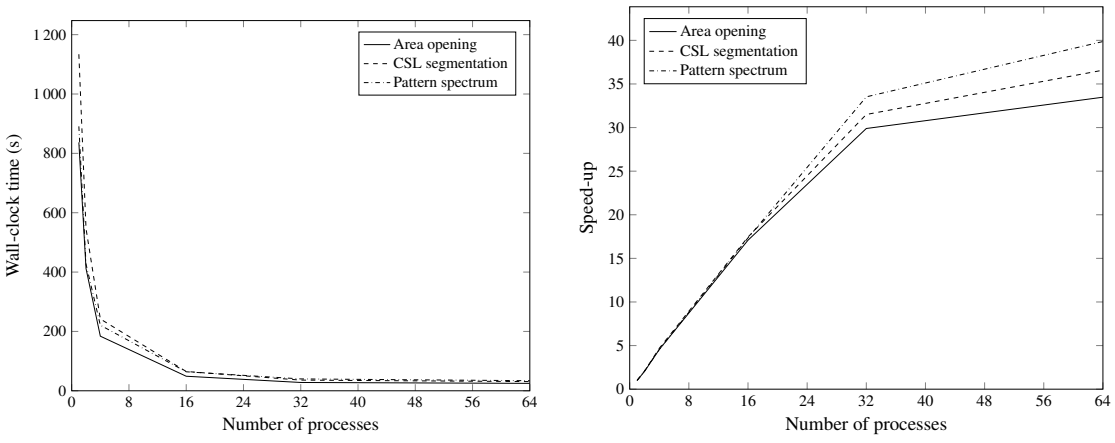
Figure 6: Andromeda Galaxy, captured with the NASA/ESA Hubble Space Telescope. Top: original RGB image (Credit: NASA, ESA, J. Dalcanton (University of Washington, USA), B. F. Williams (University of Washington, USA), L. C. Johnson (University of Washington, USA), the PHAT team, and R. Gendler). Middle: The 8-bit luminance image derived from the R, G and B channels from the original image. Bottom: after a morphological area opening, all bright connected components larger than 5×10^5 pixels have been removed.

complex CSL segmentation is more costly than computing a single area opening, and pattern spectra computations proved a bit more expensive, probably due to possible multiple visits to nodes not in the current tile by Algorithm 2. In all cases, speed-up is good up to 32 processes, after which the slope of the curve is slightly lowered. This flattening may be stronger depending on the machine used, as a similar pattern was observed in [14]. On the Peregrine cluster, the speed-up curve obtained for the Naples data set is less affected than for the images handled on the Zeus cluster. Additionally, and as mentioned previously, the structure configuration in the images can also affect the performance. This is enhanced by the speed-up difference obtained on the Haiti and NASA images on the same cluster. Using 32 processes, the former reaches a speed-up of $\sim$20 while



(a) (b)

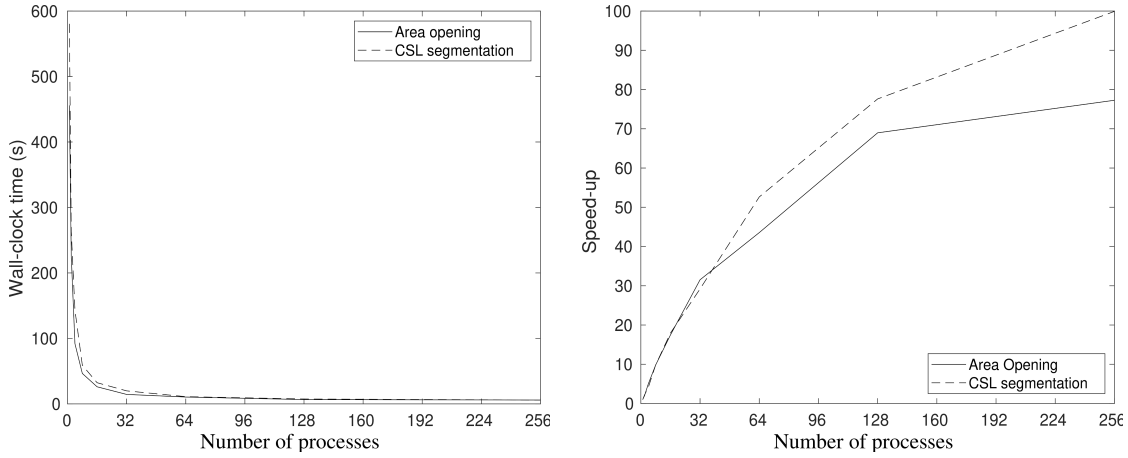
Figure 7: (a) Timings and (b) speed-up on Haiti 8-bit per pixel data set on the Zeus 64-core compute server. As the number of processes increases, the size of the tiles decreases.



(a) (b)

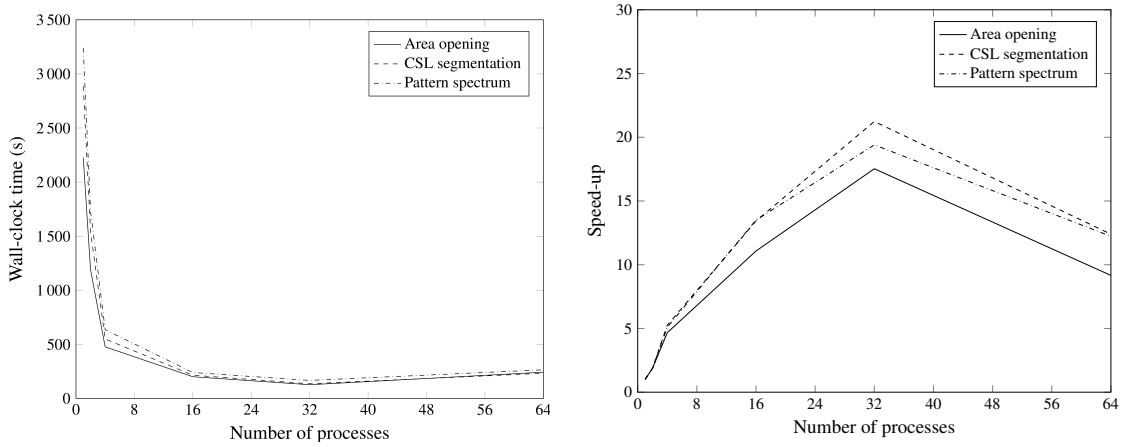
Figure 8: (a) Timings and (b) speed-up on the astronomical NASA 8-bit per pixel data set on the Zeus 64-core compute server. As the number of processes increases, the size of the tiles decreases.

the latter reaches ∼30. This is somewhat expected, if the tiles border are located on a single dark fairly smooth background, like in the NASA astronomical data set, the number of level-root nodes to be merged remains small. Conversely, when the tiles border cut several individual structures like in the Haiti remote sensing data set, the number of individual components to be merged is larger, and the overall merging operation is more expensive. Finally, there is likely to be a limit in the performance achieved as we reach lower tile sizes. Indeed, timings then become predominantly affected by the number of merging/updating steps and the communication between the processes rather than by the computation and filtering of the local trees. The combination of these three effects can explain the flattening of the curve as seen in the Figures 7, 8 and 9.



(a) (b)

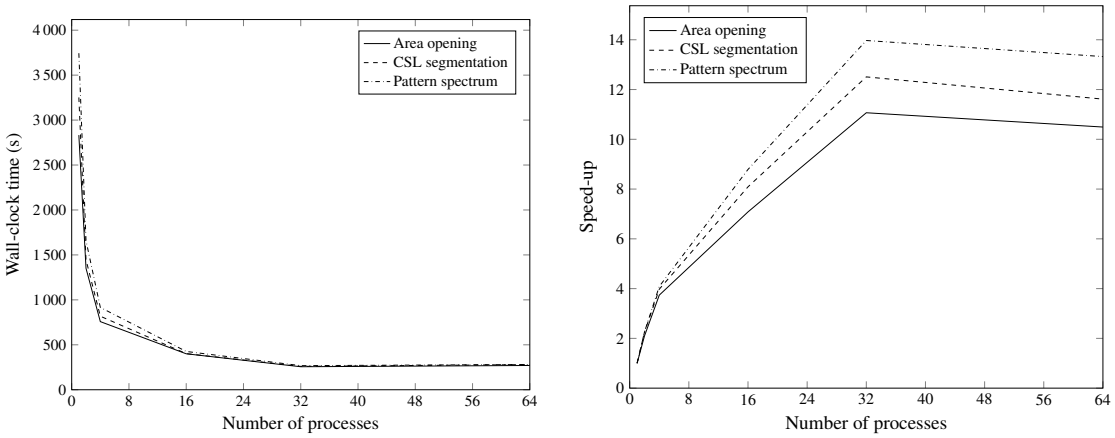
Figure 9: (a) Timings and (b) speed-up on Naples 8-bit per pixel data set on Peregrine cluster. As the number of processes increases, the size of the tiles decreases.



(a) (b)

Figure 10: (a) Timings and (b) speed-up on Haiti 16-bit per pixel data set on the Zeus 64-core compute server. As the number of processes increases, the size of the tiles decreases.

We then performed the same experiment on the two 16-bit per pixel images, using the Zeus cluster. Figure 10 and Figure 11 show the results for the Haiti and for the NASA data respectively. The overall trend is similar to the 8-bit per pixel case, except that the effects are stronger for this higher dynamic range. The slope of the speed-up curve is considerably lowered for the NASA image, and even decline for the Haiti data set when the method uses more than 32 processes. This is expected, as the number of gray levels is directly related to the total number of nodes in the tree. For 16-bits dynamic range, the merging and updating stages of our method are more computationally expensive (more possible level-roots to merge and to duplicate) than for the 8-bits case.



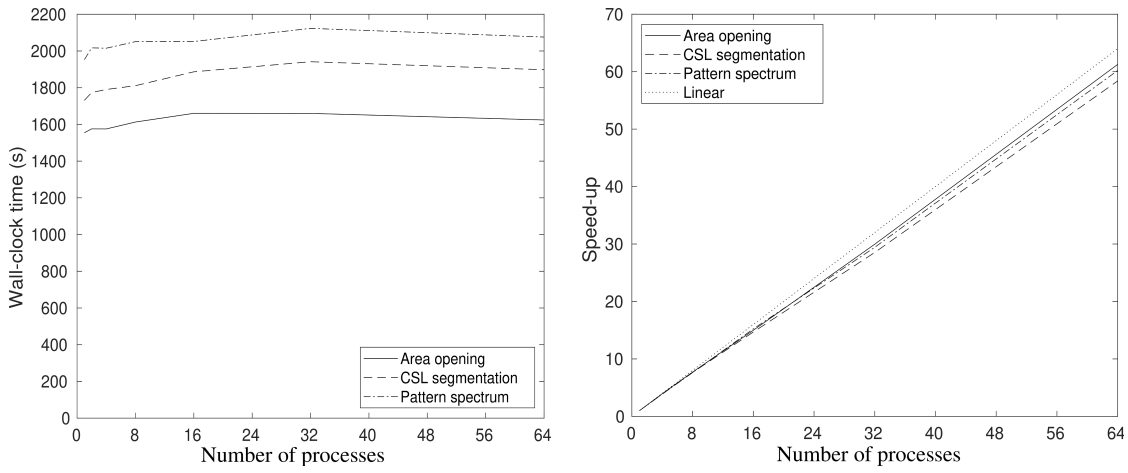
(a) (b)

Figure 11: (a) Timings and (b) speed-up on the astronomical NASA 16-bit per pixel data set on the Zeus 64-core compute server. As the number of processes increases, the size of the tiles decreases.

The first experiment is limited to relatively small images since, to compute the speed-up analysis, the data set needs to fit in a single node of the cluster. To explore how the method performs with larger dataset, we used a different approach. The idea is to use multiple copies of the same image to generate synthetic larger images of size p times the original size (p is the number of processes). We then measured the wall-clock time for 1, 2, 4, ..., 64 processes (corresponding to images of 1.2, 2.4, 4.8 ..., 76.8 Gpixels). The speed-up is estimated by $p \times t(1)/t(p)$. Figure 12 shows the timings and speed-up obtained, using the 16-bit per pixel image of Haiti. For all cases, the wall-clock time remains roughly constant when increasing the number of processes (almost linear speed-up). The multi-scale analysis (CSL segmentation and pattern spectra) performs slightly less well than the basic area opening. As discussed previously, this is expected because these operations are more costly to perform. This experiment enhances the good performance of the method for large, low-dynamic range images.

4.2 Memory efficiency

We explored the memory usage of the method when the latter is applied to the five data sets previously described. To do so, the full image was cut in p tiles, where p is the number of processes, and we measured the number of nodes in the tree structures, their size (in bytes), and reported the maximum memory used by the process 0. All the values are presented in the Table 1 based on a morphological area opening operation. Using multi-scale computations should result in a slightly larger memory consumption as these operations use additional variables in the local component



(a) (b)

Figure 12: (a) Timings and (b) speed-up on Haiti 16-bit per pixel data set on Peregrine cluster. The tile size remains constant, such that the image size increases linearly with the number of processes used. The times stay roughly constant for linearly increasing compute load.

tree (e.g *privateArea* and *gval_par*). Nevertheless, this should not affect the global trend of the memory usage relatively to the increasing number of processes used. Several conclusions can be derived from Table 1. First, similarly to [10], it shows that the size of the boundary trees is negligible compared to the local max-tree. It never exceeds a few percents of the max-tree size, and the larger sizes are reached with the 16-bit version of the Haiti data set. As expected, processing higher dynamic range image increases the size of the boundary tree, as its number of nodes scales with the number of gray values. Note that, in our method, we always assume that the input data is a 16-bit per pixel image, i.e that the size of a single node in the max-tree or in the boundary tree is fixed whether we are dealing with 8-bit or 16-bit per pixel images. The size of the tree structures then only varies with the number of nodes it contains.

The last column in Table 1 gives useful insights on the maximum memory usage in the process 0. For the 8-bit data sets, the maximum memory allocated is reached during the local max-tree creation, the tree size itself being more than 80% of it. The additional allocations originate from the different structures used during the tree flooding. For the 16-bits case, a similar behavior is observed up to 32 processes. When using 64 processes, the peak of memory consumption is reached during the merging/updating phase. Merging and storing the successive boundary trees only impacts the memory used when the ratio of the boundary tree size over the max-tree size increases (smaller tiles and/or higher dynamic range). In most of the cases, the relatively small size of the boundary trees stored, and the modest maximum number of nodes that are duplicated

from one process to an other have a limited impact on the memory. One should notice the lower memory consumption in the 16-bit case compared to the 8-bit version of the Haiti or NASA data set regardless of the number of processes. These discrepancies arise from the different algorithms used to build to local tree in both cases. As briefly mentioned in the Section 3.2, in the 8-bit case, we use the method of [20] which is adapted for low dynamic range images. In the 16-bit case, we use the method of [28], which performs better for this dynamic range and requires less memory allocations than the first method. Then, the differences between the memory usage for two distinct dynamic ranges are mostly related to the characteristics of the flooding algorithm it refers to. In a last step, we explored how the memory consumption on the process 0 decreases with the number of processes. Process 0 is taken as a reference as it has the largest number of boundary trees stored when all the merges have been completed. Figure 13 shows the evolution of this maximal memory usage and the memory efficiency, defined as $M(1)/M(p)$, where $M(1)$ and $M(p)$ represent the maximal memory usage when 1 and p processes are used respectively. Using more processes strongly reduce the maximum memory needed per process. For 8-bit per pixel data sets, dividing the image over p process reduces the maximal memory needed by a factor p. For 16-bit per pixel data sets, the efficiency is slightly lower (memory usage on a single node is reduced by a factor of 40 when using 64 processes for the 16-bit Haiti data set). This is expected, boundary trees are larger in the 16-bit case rather than in the 8-bit case as they include more nodes. When the number of processes increases, the number of boundary trees stored increases, and their size may not be negligible compared to the local tree size. Nevertheless, even in these cases, the maximum memory used in a node remains low. These results enhances the ability of our method to process Tera-scale images using an efficient distributed-memory approach.

5 Conclusions

In this work we present the first complete method to perform attribute filtering and multi-scale analysis using distributed component forests on images of up to 16 bits per pixel. We have shown that this required little modification of existing component tree processing methods. This implies that distributed component forests can be adapted to a range of component-tree-based methods for very large images. Some may of course be more challenging than others. For example, if we want to do clustering-based filtering of component trees [11] considerable inter-process communication may well be needed in the filtering stage as well. Likewise, connected filtering using tree-based shape spaces [32], which build a component tree on the component tree of an image, might also be

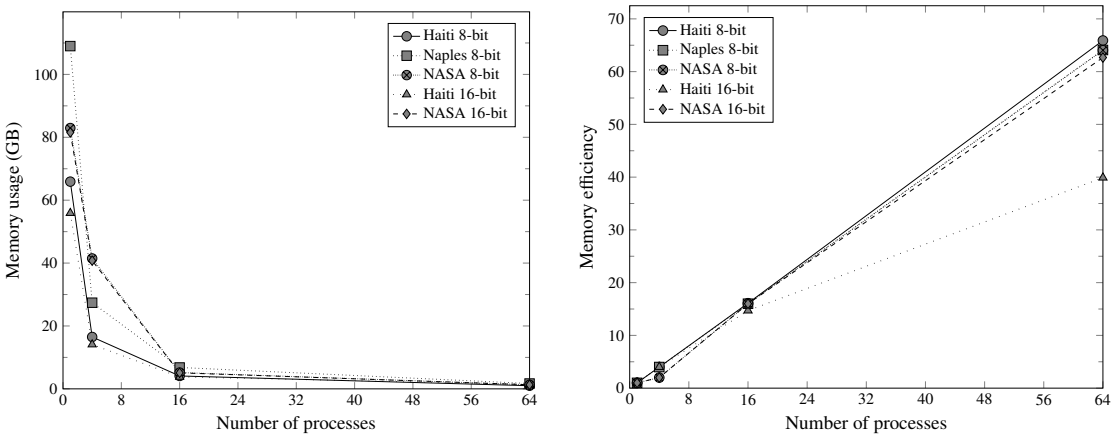
Table 1: Memory usage for the five data sets with different size and dynamic range.

Data	Proc	Number of nodes		Nodes added	Memory Used
		Max-tree	Boundary tree		
	1	$1.2\ 10^9$ (53.6 GB)	-	-	65.9 GB
Haiti	4	$3.0\ 10^8$ (13.4 GB)	$9.6\ 10^4$ (7 MB)	0	16.5 GB
8-bit	16	$7.5\ 10^7$ (3.4 GB)	$9.0\ 10^4$ (7 MB)	$2.7\ 10^3$ (<1 MB)	4.1 GB
	64	$1.9\ 10^7$ (0.9 GB)	$4.7\ 10^4$ (4 MB)	$6.6\ 10^3$ (<1 MB)	1.0 GB
	1	$1.4\ 10^9$ (67.5 GB)	-	-	82.9 GB
NASA	4	$3.5\ 10^8$ (16.9 GB)	$1.2\ 10^5$ (9 MB)	$4.0\ 10^4$ (2 MB)	20.7 GB
8-bit	16	$8.8\ 10^7$ (4.2 GB)	$1.2\ 10^5$ (9 MB)	$4.2\ 10^4$ (2 MB)	5.2 GB
	64	$2.2\ 10^7$ (1.1 GB)	$6.4\ 10^4$ (5 MB)	$2.3\ 10^4$ (1 MB)	1.3 GB
	1	$2.0\ 10^9$ (89 GB)	-	-	109 GB
Naples	4	$4.9\ 10^8$ (22.3 GB)	$5.5\ 10^4$ (4 MB)	$9.3\ 10^2$ (<1 MB)	27.4 GB
8-bit	16	$1.2\ 10^8$ (5.6 GB)	$5.7\ 10^4$ (4 MB)	$1.1\ 10^3$ (<1 MB)	6.8 GB
	64	$3.1\ 10^7$ (1.4 GB)	$3.1\ 10^4$ (2 MB)	$8.2\ 10^2$ (<1 MB)	1.7 GB
	1	$1.2\ 10^9$ (53.6 GB)	-	-	55.9 GB
Haiti	4	$3.0\ 10^8$ (13.4 GB)	$4.1\ 10^5$ (31 MB)	$2.9\ 10^5$ (19 MB)	14.1 GB
16-bit	16	$7.5\ 10^7$ (3.4 GB)	$5.6\ 10^5$ (42 MB)	$3.3\ 10^5$ (22 MB)	3.8 GB
	64	$1.9\ 10^7$ (0.9 GB)	$3.4\ 10^5$ (25 MB)	$1.9\ 10^5$ (13 MB)	1.4 GB
	1	$1.4\ 10^9$ (67.5 GB)	-	-	81.5 GB
NASA	4	$3.5\ 10^8$ (16.9 GB)	$2.1\ 10^5$ (15 MB)	$8.5\ 10^4$ (4 MB)	20.4 GB
16-bit	16	$8.8\ 10^7$ (4.2 GB)	$2.6\ 10^5$ (19 MB)	$1.1\ 10^5$ (5 MB)	5.1 GB
	64	$2.2\ 10^7$ (1.1 GB)	$1.5\ 10^5$ (11 MB)	$7.4\ 10^4$ (4 MB)	1.3 GB

Notes: First column describes the data set processed. The second column gives the number of processes used. Third and fourth column report the number of nodes and the corresponding size in the memory for the initial local max-tree and boundary tree respectively. GB and MB stand for GigaBytes (1024^3 bytes) and MegaBytes (1024^2 bytes) respectively. Fifth column details the maximal number of nodes duplicated to correct a local component tree at the end of the *correct_borders* procedure. Sixth column shows the maximal amount of memory allocated in the process 0.

harder to implement. In future work, we will extend the method to 3-D, which will incur higher overheads, due to the comparatively larger boundary surface between the 3-D tiles. Furthermore, the algorithm will be adapted to perform opening/closing, CSL segmentation and pattern spectra based on different attributes. Modifications will be made such that the inclusion of new attributes suitable for specific applications is easier. Additionally, we are planning an extension of the method to floating point or even 32-bit per pixel. This will be much more difficult, due to the huge potential increase in maximum boundary-tree size and to the limitation of the merging algorithm at these dynamic ranges (see [14] for discussion), and may require a very different approach.

A similar approach could be extended to the alpha tree [17], for filtering and segmentation of colour and hyper-spectral images, which has applications in information mining in very large



(a) (b)

Figure 13: (a) Maximum memory usage (in GigyBytes) and (b) the corresponding efficiency measured when a morphological area opening is applied to the different data sets. In these tests, the image is divided in smaller tiles as the number of processes used increases. The efficiency is linear for 8-bit per pixel data sets, meaning that the memory used decreases linearly with the number of processes used. In the 16-bit case, the efficiency deviates from the linear trend when a large number of processes is used. This is expected as, when the tile size decreases, the boundary tree size is not negligible anymore compared to the local max-tree size.

images [16]. The self-dual Tree-of-Shapes [7] might also allow a distributed forest variant.

More tests are needed on larger images, and on bigger machines to assess the full potential of these methods, but already, they offer the only working method for distributed computation of several connected filters and related operators. The code is available upon request.

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 authors would also like to thank Jan Kazemier for his initial version of the distributed-memory code.

References

[1] J. Benediktsson, J. Palmason and J. Sveinsson, Classification of hyperspectral data from urban areas based on extended morphological profiles, *IEEE Trans. Geosci. Remote Sensing* **43** (march 2005) 480 – 491.

[2] C. Berger, T. Geraud, R. Levillain, N. Widynski, A. Baillard and E. Bertin, Effective component tree computation with application to pattern recognition in astronomical imaging, in *Proc. Int. Conf. Image Proc. 2007*, Vol. IV (September 16–19 2007) pp. 41–44.

[3] E. J. Breen and R. Jones, Attribute openings, thinnings and granulometries, *Comp. Vis. Image Understand.* **64**(3) (1996) 377–389.

[4] E. Carlinet and T. Géraud, A comparative review of component tree computation algorithms, *IEEE Trans. Image Proc.* **23**(9) (2014) 3885–3895.

[5] F. Cheng and A. N. Venetsanopoulos, An adaptive morphological filter for image processing, *IEEE Trans. Image Proc.* **1** (1992) 533–539.

[6] F. G. A. Faas, M. C. Avramut, B. M. van den Berg, A. M. Mommaas, A. J. Koster and R. B. G. Ravelli, Virtual nanoscopy: Generation of ultra-large high resolution electron microscopy maps, *The Journal of Cell Biology* **198**(3) (2012) 457–469.

[7] T. Géraud, E. Carlinet, S. Crozet and L. Najman, A quasi-linear algorithm to compute the tree of shapes of nd images, in *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing* (2013) pp. 98–110.

[8] M. Götz, G. Cavallaro, T. Géraud, M. Book and M. Riedel, Parallel computation of component trees on distributed memory machines, *IEEE Transactions on Parallel and Distributed Systems* (2018).

[9] R. Jones, Connected filtering and segmentation using component trees, *Comp. Vis. Image Understand.* **75** (1999) 215–228.

[10] J. J. Kazemier, G. K. Ouzounis and M. H. F. Wilkinson, Connected morphological attribute filters on distributed memory parallel machines, in *Proc. Int. Symp. Math. Morphology (ISMM) 2017* (2017) pp. 357–368.

[11] F. Kiwanuka and M. Wilkinson, Cluster based vector attribute filtering, *Mathematical Morphology - Theory and Applications* **1**(1) (2016) p. 116135.

[12] P. Maragos, Pattern spectrum and multiscale shape representation, *IEEE Trans. Pattern Anal. Mach. Intell.* **11** (1989) 701–715.

[13] A. Meijster, M. A. Westenberg and M. H. F. Wilkinson, Interactive shape preserving filtering and visualization of volumetric data, in *Fourth IASTED Conf Comp. Signal Image Proc., SIP2002* (August 12-14 2002) pp. 640–643.

[14] U. Moschini, A. Meijster and M. H. F. Wilkinson, A hybrid shared-memory parallel max-tree algorithm for extreme dynamic-range images, *IEEE Trans. Pattern Anal. Mach. Intell.* **40** (2018) 513–526.

[15] L. Najman and M. Couprie, Building the component tree in quasi-linear time, *IEEE Trans. Image Proc.* **15** (2006) 3531–3539.

[16] G. K. Ouzounis, V. Syris, L. Gueguen and P. Soille, The switchboard platform for interactive information mining, in *Proc. of ESA-EUSC-JRC 8th Conference on Image Information Mining* (2012) pp. 26–30.

[17] G. K. Ouzounis and P. Soille, *The Alpha-Tree algorithm* (Publications Office of the European Union, Dec. 2012).

[18] M. Pesaresi and J. Benediktsson, A new approach for the morphological segmentation of high-resolution satellite imagery, *IEEE Trans. Geosci. Remote Sensing* **39** (feb 2001) 309 –320.

[19] I. K. E. Purnama, K. Aryanto and M. H. Wilkinson, Non-compactness attribute filtering to extract retinal blood vessels in fundus images, *International Journal of E-Health and Medical Communications (IJEHMC)* **1**(3) (2010) 16–27.

[20] P. Salembier, A. Oliveras and L. Garrido, Anti-extensive connected operators for image and sequence processing, *IEEE Trans. Image Proc.* **7** (1998) 555–570.

[21] P. Salembier and M. H. F. Wilkinson, Connected operators: A review of region-based morphological image processing techniques, *IEEE Signal Processing Magazine* **26**(6) (2009) 136–157.

[22] R. E. Tarjan, Efficiency of a good but not linear set union algorithm, *J. ACM* **22** (1975) 215–225.

[23] E. R. Urbach, N. J. Boersma and M. H. F. Wilkinson, Vector-attribute filters, in *Mathematical Morphology: 40 Years On, Proc. Int. Symp. Math. Morphology (ISMM) 2005* (18-20 April 2005) pp. 95–104.

[24] E. R. Urbach, J. B. T. M. Roerdink and M. H. F. Wilkinson, Connected shape-size pattern spectra for rotation and scale-invariant classification of gray-scale images, *IEEE Trans. Pattern Anal. Mach. Intell.* **29** (2007) 272–285.

[25] L. Vincent, Morphological area openings and closings for grey-scale images, in Y.-L. O, A. Toet, D. Foster, H. J. A. M. Heijmans and P. Meer (eds.), *Shape in Picture: Mathematical Description of Shape in Grey-level Images* (NATO, 1993) pp. 197–208.

[26] L. Vincent, Granulometries and opening trees, *Fundamenta Informaticae* **41** (2000) 57–90.

[27] M. A. Westenberg, J. B. T. M. Roerdink and M. H. F. Wilkinson, Volumetric attribute filtering and interactive visualization using the max-tree representation, *IEEE Trans. Image Proc.* **16** (2007) 2943–2952.

[28] M. H. F. Wilkinson, A fast component-tree algorithm for high dynamic-range images and second generation connectivity, in *Proc. Int. Conf. Image Proc. 2011* (2011) pp. 1041–1044.

[29] M. H. F. Wilkinson, H. Gao, W. H. Hesselink, J. E. Jonker and A. Meijster, Concurrent computation of attribute filters using shared memory parallel machines, *IEEE Trans. Pattern Anal. Mach. Intell.* **30**(10) (2008) 1800–1813.

[30] M. H. F. Wilkinson, U. Moschini, G. K. Ouzounis and M. Pesaresi, Concurrent computation of connected pattern spectra for very large image information mining, in *Proc. of ESA-EUSC-JRC 8th Conference on Image Information Mining* (2012) pp. 21–25.

[31] M. H. F. Wilkinson, M. Pesaresi and G. K. Ouzounis, An efficient parallel algorithm for multi-scale analysis of connected components in gigapixel images, *ISPRS International Journal of Geo-Information* **5** (2 2016) p. 22.

[32] Y. Xu, T. Géraud and L. Najman, Connected filtering on tree-based shape-spaces, *IEEE Trans. Pattern Anal. Mach. Intell.* **38**(6) (2016) 1126–1140.