Efficient morphological tools for astronomical image processing

Moschini, Ugo

Submitted as:

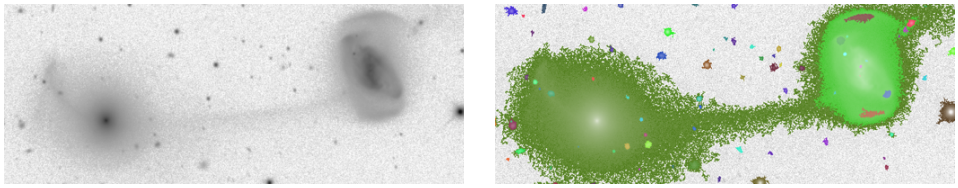
U. Moschini, A. Mejster and M.H.F. Wilkinson, *"A Hybrid Shared-Memory Parallel Max-Tree Algorithm for Extreme Dynamic-Range Images"*, submitted (under review) to IEEE Transactions on Pattern Analysis and Machine Intelligence.

Chapter 4

A Hybrid Shared-Memory Parallel Max-Tree Algorithm for Extreme Dynamic-Range Images

Abstract

Max-trees, or component trees, are graph structures that represent the connected components of an image in a hierarchical way. Nowadays, many application fields rely on images with high-dynamic range or floating point values. Efficient sequential algorithms exist to build trees and compute attributes for images of any bit depth. However, we show that the current parallel algorithms perform poorly already with integers at bit depths higher than 16 bits per pixel. We propose a parallel method combining the two worlds of flooding and merging max-tree algorithms. First, a pilot max-tree of a quantized version of the image is built in parallel using a flooding method. Later, this structure is used in a parallel leaf-to-root approach to compute efficiently the final max-tree and to drive the merging of the sub-trees computed by the threads. We present an analysis of the performance both on simulated and actual 2D images and 3D volumes. Execution times are about $20\times$ *better than the fastest sequential algorithm and speed-up goes up to* $30 - 40$ *on 64 threads.*



(a) (b)

Figure 4.1: (a) image containing stars and two large interacting galaxies; (b) output of connected filtering: identified objects and nested structures are coded in different colours.

4.1 Introduction

Max-trees (Salembier et al., 1998), or component trees, are versatile and efficient data structures that represent the connected components at every threshold level of an image in a hierarchical fashion, through parent relationships between nodes. Leaf nodes represent the image maxima. The connected components organised in such trees can be filtered with different strategies (Breen and Jones, 1996; Salembier et al., 1998) and they can model different types of connectivity (Ouzounis and Wilkinson, 2007a). For these reasons, they are powerful tools for image information mining and visualization tasks. Recent applications are in the field of astronomy (Berger et al., 2007; Perret et al., 2010; Ouzounis and Wilkinson, 2011; Teeninga et al., 2015) and remote sensing (Wilkinson et al., 2011; Pesaresi et al., 2012, 2013), to identify structures such as galaxies or building footprints from satellites. Fig. 4.1 shows an example of the segmentation of astronomical objects using a filter based on the noise statistics in the image (Teeninga et al., 2015).

Many current applications rely on high dynamic-range or floating point imagery, due to either the increasing sensitivity and technological improvements of the instruments or simply to the type of measurement observed, e.g. radio emissions and CT scans. Astronomical or remote sensing images routinely show high dynamic range integers or floating point values.

In the rest of the chapter, we refer to images or volumes up to 64-bit integers or single or double precision floating point values per element as *XDR* (extreme dynamic range). As we will show in Section 4.5, there is no state-of-the-art parallel algorithm suitable for building max-trees of XDR images. Existing parallel algorithms rely on partitioning the image and building sub-trees of each partition that are then merged in the final max-tree. We will show that merging sub-trees of XDR images is costly and makes the parallel building phases of no use. In this chapter, we provide a parallel algorithm which deals well with XDR images. It opens up the possibility of creating max-trees representing the whole image up to 40 times faster on 64 processors. There is no need to divide an image in smaller sections or lower its dynamic range, thus breaking structures or causing loss of information:

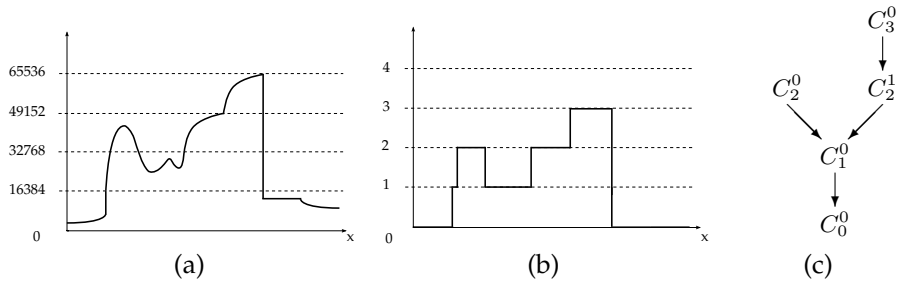


Figure 4.2: (a) represents a 16-bit integer 1D image. (b) is a quantized image after mapping the original intensities onto 4 intensities. (c) illustrates the max-tree structure of connected components of the quantized image shown in (b).

images can be processed as they are. Several sequential max-tree algorithms suitable for XDR images exist. They can be divided in two classes. Root-to-leaf flooding methods start from the root node, which is a pixel with the lowest intensity, and perform a depth-first traversal of the connected components at higher intensities (Salembier et al., 1998; Wilkinson, 2011a). Leaf-to-root merging methods start from the image maxima (Jones, 1999) or work on sorted pixels that are merged into nodes of a tree using Tarjan's union find algorithm (Tarjan, 1975; Najman and Couprie, 2006). The parallel solution proposed in this work combines the most efficient root-to-leaf flooding and leaf-to-root merging algorithms to derive a two-stage, parallel algorithm suitable for XDR imagery. To tackle the problem of the costly merging phase, the idea is to create initially a *pilot* max-tree of a quantized version of the input image. The pixel intensities are quantized over a low number of values, equal to the number of threads of the parallel program. Using a low number of quantization levels, the existing parallel algorithm based on hierarchical queues (Wilkinson et al., 2008) performs well in the task of building the *pilot* max-tree. This tree is used as a support data structure during a parallel *refinement* stage that yields the final max-tree of the input image. Without the support of the *pilot* tree, the high cost of merging the sub-trees together would still be present. The term "refinement" comes from the fact that the whole process of building the final max-tree can be seen as a refining operation on the *pilot* max-tree. In this last phase, the quantization is used to partition the pixels according to their intensities, rather than spatially.

4.2 The max-tree structure

The image domain can be fully partitioned into connected components. Connected components are sets of path-wise connected pixels of maximal extent, having the same intensity value. When filtering an image, components can be preserved or

removed according to a threshold value on a given attribute (Breen and Jones, 1996). For example, regions with area smaller than a certain value or with a certain shape (Urbach and Wilkinson, 2002; Urbach et al., 2007) are deleted from the image. Efficient filtering is implemented by organising connected components in a tree structure, on which filtering reduces to pruning nodes in the tree and assigning new intensities. Several rules stating how the nodes in the tree are removed and new intensities are assigned (Salembier et al., 1998; Urbach and Wilkinson, 2002). Given an image f, a peak component P_h^i is defined as the set of all the pixels of a connected component of the thresholded image $T_h(f)$, where $T_h(f) = \{p \mid f(p) \geq h\}$. Of course, there can be many connected components at each level h. These components are indexed by i. Connected components can be organized in a max-tree structure, where each peak component P_h^i in the image corresponds to a node C_h^i. For example, Fig. 4.2c shows the max-tree structure of the 1D image in Fig. 4.2b. The arrows denote a parent relation between the nested components.

4.3 State of the art sequential algorithms for XDR images

We can divide the max-tree building algorithms into two categories: root-to-leaf *flooding* and leaf-to-root *merging*. Following the latter approach, Najman and Couprie (Najman and Couprie, 2006) described an algorithm to build a max-tree using Tarjan's union-find (Tarjan, 1975) method. Berger et al. (Berger et al., 2007) published a more memory efficient version which we will refer to as the "Berger algorithm" in the following. The sequential Berger algorithm is suitable for highly quantized or even floating point images, i.e. XDR images. Both Tarjan's and Berger algorithms will be explained in more details in Section 4.4.

Root-to-leaf algorithms follow the idea presented in the paper from Salembier et al. (Salembier et al., 1998). A depth-first construction for the tree is used, starting from the pixel with lowest intensity. The main drawback of this approach is faced precisely with XDR images, and is due to the fact that it uses hierarchical queues to handle the pixel values during the flooding. It scales linearly with the number of grey levels and with floating point values the worst-case time complexity is $O(N^2)$ (Wilkinson et al., 2008), with N the number of pixels. Memory complexity is $O(N + G)$, with G the number of intensities, in principle infinite for XDR images. Wilkinson (Wilkinson, 2011a) presented a new sequential algorithm that is based on a priority queue and a stack, based on a combination of the algorithms in Salembier et al. (1998) and Hesselink (2003). It cancels the costly effect of updating the hierarchical queues when dealing with XDR images. The memory usage is simply linear in the data size. Therefore, the size of the data structures does not scale proportionally to the range of the possible intensity values. A similar problem was

addressed with the leaf-to-root algorithm in Jones (1999) that uses a priority queue that is intrinsically $O(N^2 \log N)$ (Meijster and Wilkinson, 2002). Both the approaches by Berger et al. and by Wilkinson deal well with XDR images in the sequential case, with the former faster than the latter, but with a larger memory footprint.

4.4 Two important max-tree algorithms

As mentioned in the Introduction, our new parallel algorithm is based on a two-step process involving building the *pilot* max-tree of a quantized version of the input image and a refinement phase that builds the final tree of the original image. We will describe our hybrid algorithm in Section 4.6, but for now it is important to point out that the two steps are based on two existing algorithms. The first step is based on the parallel method in Wilkinson et al. (2008) (see Section 4.4.3), while the second step, that forms the core of our novel parallel solution, is based on the sequential Berger algorithm (see Section 4.4.2). A good knowledge of these two algorithms is essential to fully understand how we modified and combined them together to our end. Both algorithms are based on the Tarjan's union-find algorithm (Tarjan, 1975) introduced in the following section.

4.4.1 The Tarjan's union-find algorithm

The union-find algorithm by Tarjan (Tarjan, 1975) was designed to keep track of disjoint sets. Image connected components do not overlap, and therefore examples of disjoint subsets of the image domain. Each set is described by a tree, rooted in an arbitrary element of the set chosen to be the representative for that set. Each root points to itself, while the other elements point to their parent. To determine if two elements belong to the same set, it is sufficient to check if they share the same root. The shorter the root path, the more efficient this becomes. Tarjan proposed an efficient algorithm to maintain a collection of disjoint sets. Its complexity is quasi-linear (Tarjan, 1975). His method uses three operations on the set elements p, q, namely MakeSet(p), FindRoot(p), Union(p,q). Makeset(p) makes the p element a singleton set; FindRoot(p) returns the root of the tree containing p; Union(p,q) implements the merging of the two sets containing elements p and q. A union-find algorithm used for the computation of attribute openings and closings was proposed in Meijster and Wilkinson (2002) where it was suggested it could possibly be suitable for parallel implementation. Pixels are processed in grayscale order, so peak components could be processed simultaneously. Peak components are created and then merged, their attributes being updated. Najman et al. (Najman and Couprie, 2006) later adapted the union-find approach to build max-trees.

In the rest of the chapter, any rooted tree will be represented as an array of node

Algorithm 4.1. Pseudo-code of the Berger algorithm. The output is the max-tree of image f, represented in the *node* array.

```
1:  procedure RUNBERGER(Image f)
2:      S ← SORTPIXELSDECREASING(f);
3:      for all pixel p ∈ S do
4:          node[p].parent ← p; zpar[p] ← p;
5:          for all pixel q neighbours of p with zpar[q] ≠ −1 do
6:              r ← FINDROOT(q);
7:              if r ≠ p then
8:                  zpar[r] ← p; node[r].parent ← p;
9:                  node[p].Area ← node[p].Area + node[r].Area;
10:             end if
11:         end for
12:     end for
13: end procedure
14: procedure FINDROOT(Pixel p)
15:     if zpar[p] ≠ p then
16:         zpar[p] ← FINDROOT(zpar[p]);
17:     end if
18:     return p;
19: end procedure
20: procedure INIT(Image f)
21:     for all pixel p ∈ f do
22:         zpar[p] ← −1;
23:         node[p].parent ← −1;
24:         node[p].Area ← 1;
25:     end for
26:     RunBerger(f);
27: end procedure
```

structures. Each structure contains a parent pointer *parent* and the area attribute value in *area*. Other attributes can be computed as explained in Breen and Jones (1996), but we used area for simplicity. This representation involves one node structure per image pixel. The execution of the union-find method will set the parent pointers accordingly to the connected components (disjoint sets). Actually, only the nodes which have a parent node at a lower intensity are necessary to represent the whole tree and hold the correct attribute values. These nodes are called *level roots* (Wilkinson et al., 2008) (known also as canonical elements in Najman and Couprie (2006)).

4.4.2 The sequential Berger algorithm.

The sequential Berger algorithm (Berger et al., 2007) uses the Tarjan's method and structures to build the max-tree. Parent pointers in the tree are correctly updated via

Algorithm 4.2. Concurrent construction of the max-tree for thread th on K threads.

```
1:  procedure ParallelHierarchialQAlgorithm(Thread th, Image partition P)
2:      m ← min(P);
3:      Add m to the Queue at level P(m);
4:      isVisited[m] ← true;
5:      levelroot[P(m)] ← m;
6:      Flood(P(m), P, 0);                                   ▷ see Alg. 4.3
7:      i ← 1; q ← p;
8:      while (th + i < K) ∧ (q%2 = 0) do
9:          Wait to glue with right-hand neighbour;
10:         for all Edges (u,v) b/w partition P_th and P_{th+i} do
11:             Connect(th, i, (u, v));                      ▷ see Alg. 4.4
12:         end for
13:         i ← 2 · i; q ← q/2;
14:     end while
15:     if th ≠ 0 then
16:         Notify left-hand neighbour;
17:         Wait for Thread 0;
18:     end if
19: end procedure
```

the union-find algorithm. For a fast computation of the parent pointers, root nodes of every disjoint set must be accessed efficiently (path compression). The FINDROOT procedure at lines 14-19 in Alg. 4.1 is now also used to implement path compression: the root node of a set is efficiently found be letting each element in the set point directly to it. Since path compression cannot be applied on on the parent pointers directly because it would destroy the hierarchy of the nodes, an auxiliary data structure is used: the array *zpar* with length equal to the number of elements (pixels) in some image f. Therefore, path compression is applied onto the *zpar* structure, that contains the root nodes of disjoint sets while they are computed. As Alg. 4.1 shows, function INIT(f) initialises the data structures *zpar* and *node*. The area attribute is set to 1 for all the pixels (singleton sets). Then, the Berger algorithm is run: pixels are sorted and retrieved from S in decreasing order. Function RUNBERGER(f) ultimately returns the tree with the correct parent pointers of the tree. When a pixel p is retrieved from the sorted array is marked as processed by setting $zpar[p] = p$ (line 4 in Alg. 4.1). Following the way the union-find algorithm works, the already processed neighbours of p are checked and the set made of just p is merged with the neighbouring existing sets, updating the area attribute, to form a new connected component rooted in it. That is the core of the union-find procedure. On floating point images, the building algorithm by Najman and Couprie mentioned in Section 4.3 is said to have better performance (Berger et al., 2007) than the Berger one thanks to a technique called union-by-rank (Tarjan, 1975). Union-by-rank avoids creating degenerated trees in flat zones by keeping as small as possible the depth of

Algorithm 4.3. FLOOD procedure implements the recursive root-to-leaf flooding approach based on a priority queue. The output is the max-tree of the image partition P in the *node* array.

```
1:  procedure FLOOD(Level lev, Partition P, Attribute thisarea)
2:      area ← thisarea;
3:      while Queue at level lev is not empty do
4:          Extract pixel p from the Queue at level lev;
5:          for all neighbours q of p with q ∈ P do
6:              if isVisited[q] = false then
7:                  isVisited[q] ← true;
8:                  if levelroot[f(q)] = not set then
9:                      levelroot[f(q)] ← q;
10:                 else
11:                     node[q].parent = levelroot[f(q)];
12:                 end if
13:                 Add q to the Queue at level f(q);
14:                 if f(q) > lev then
15:                     childarea =← 0;
16:                     while f(q) > lev do
17:                         FLOOD(f(q), P, childarea);
18:                     end while
19:                     area ← at + childarea;
20:                 end if
21:             end if
22:         end for
23:     end while
24:     m ← lev − 1;
25:     while m ≥ 0 ∧ levelroot[m] = not set do
26:         m ← m − 1;
27:     end while
28:     if m ≥ 0 then
29:         node[levelroot[lev]].parent ← levelroot[m];
30:     end if
31:     node[levelroot[lev]].area = area;
32:     levelroot[lev] ← not set;
33:     thisarea ← area;
34: end procedure
```

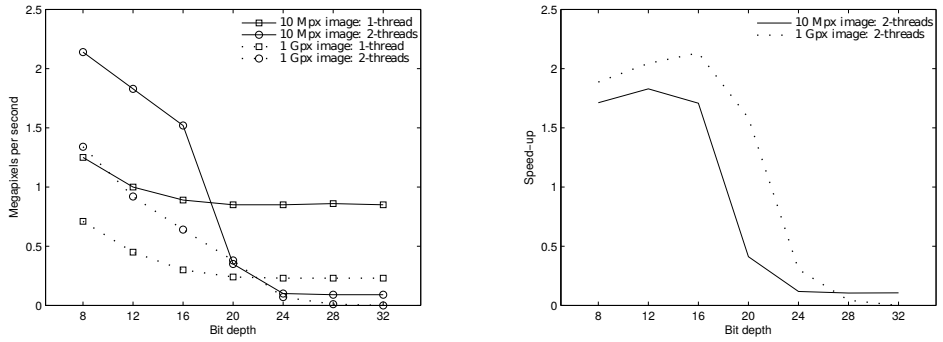
the trees. This problem arises because the last processed pixel always becomes the new root. A detailed explanation is in Tarjan (1975). If union-by-rank is disabled in the Najman and Couprie algorithm, then the Berger algorithm is not only more memory efficient, but also faster. We will see next that our two-step parallel solution does not use union-by-rank.

Algorithm 4.4. Pseudo-code of the CONNECT procedure. Symbol $\perp$ is defined as the root node of every sub-trees and $f(\perp) = -\infty$.

```
1:  procedure CONNECT(Thread th, Index i, Edge (u, v))
2:      area ← 0; areat ← 0;
3:      x ← GETLEVELROOTOF(u);
4:      y ← GETLEVELROOTOF(v);
5:      if f(x) < f(y) then
6:          Swap(x,y);
7:      end if
8:      while x ≠ y ∧ x ≠⊥ do
9:          z ← GETLEVELROOTOF(node[x].parent);
10:         if f(z) >= f(y) ∧ z ≠⊥ then
11:             node[x].Area ← node[x].Area + area;
12:             x ← z;
13:         else
14:             areat ← node[x].Area + area;
15:             area ← node[x].Area;
16:             node[x].Area ← areat;
17:             node[x].parent ← y; x ← y; y ← z;
18:         end if
19:     end while
20:     if y =⊥ then
21:         while y ≠⊥ do
22:             node[x].Area ← node[x].Area + area;
23:             x ← GETLEVELROOTOF(node[x].parent);
24:         end while
25:     end if
26: end procedure
```

4.4.3 The parallel hierarchical queue algorithm

The algorithm reported in this section is a parallelization of the sequential algorithm in Salembier et al. (1998). A thorough description is found in the work by Wilkinson et al. (Wilkinson et al., 2008). It follows a root-to-leaf approach and it is based on a hierarchical FIFO queue. The queue *Queue* has a number of entries equal to number of levels (intensities) in the image. It is initialised with the lowest intensity pixel in the image at line 3 in Alg. 4.2 and the recursive root-to-leaf flooding starts from there. The flooding process populates the queue. The parallelization of a queue-based algorithm is never trivial, but it is possible to partition the image f into sections P, build trees of every section and then merge them together. In the sequential algorithm (Salembier et al., 1998), pixels were given arbitrary labels to signal that they belong to a certain component. That makes it hard to implement the merging, because pixels need to be relabelled. The algorithm in Salembier et al. (1998) was adapted to use the same tree structures as in the Tarjan's method and to be con-



(a) (b)

Figure 4.3: (a) shows the performance of the parallel algorithm based on priority queues on images with randomly generated floating point values. The two-threaded version performs better than the single-threaded only up to 16-20 bits per pixel. With higher bit depths, the cost of merging cancels any benefit. (b) Speed-up is close to its optimal value up to about 16 bits per pixels, then it drops dramatically.

sistent with the union-find approach by letting each element of a component point directly to the first pixel found. This alows union-find type merging of the nodes in the merging phase. Alg. 4.2 shows how the max-tree of a partition P of f assigned to each thread is built and how the data structures are initialised. Array *isVisited* keeps track of the visited pixels, while array *levelroot* is used to hold the latest level root at the currently flooded intensity level. The flooding procedure is illustrated in Alg. 4.3. We know that each connected component is characterised by a unique element (level root): the set of nodes that an element belongs to is rooted in such unique element. Path compression is implemented by making the elements point directly to the level root (see line 11 in Alg. 4.3). The Union routine that sets the root of one of the trees to the root of the other one is implemented in the flooding procedure by adjusting the parent pointers. Our *pilot* max-tree construction described in Section 4.7 will be based on a modified version of the FLOOD. It will be shown that the only difference lies in the definition of level root and the way the array *levelroot* is handled.

Once the trees of the partitions have been built, the issue is how to merge them together. All the trees are stored in the *node* array: each root node of the partition trees has a parent pointer equal to -1. Thread 0 eventually computes the final merging that yields a correct max-tree of f. Lines 7-15 of Alg. 4.2 refer to this process. The crucial step is the procedure CONNECT invoked at line 11 of Alg. 4.2 and reported in Alg. 4.4 for each pair of neighbouring nodes u,v between two adjacent partitions. Their parent pointers are followed down in the tree until they meet in a node or in the root node. If u and v belong to a component that was split in two partitions,

one of the two level roots is chosen as the new representative and pointers and area value are updated in the chosen level root. The updating is propagated down till the root of the tree or until the same level root is reached (see the *while* loop at line 8 of Alg. 4.4). The implementation of this phase is not trivial. It shows similarities with the merge sort algorithm. We refer to the work in Wilkinson et al. (2008) for a detailed explanation and proof of correctness. For our purposes, however, it is important to stress the fact that while working nicely on images with low quantization (8, 12 or 16 bits per pixel), this merging approach turns out to be impracticable in case of XDR images. The exact merging points are checked parsing the trees from the leaf nodes till the root node, in the worst case, for every edge. Therefore, the complexity of a CONNECT operation grows exponentially with the bit depth, as shown in the following Section 4.5. Thus, we propose a different approach to implement the merging of sub-trees in our two-step parallel algorithm.

4.5 Issues with possible parallel algorithms for XDR images

While looking for a parallel version of the state-of-the-art sequential algorithms presented in Section 4.3 that can deal with very high-dynamic range images, we made a first attempt implementing straight away a parallel version of the method by Wilkinson (Wilkinson, 2011a), based on priority queues that uses the same merging method as the hierarchical queue algorithm of Section 4.4.3. That is, the merging phase for XDR images becomes very costly and makes the parallel algorithm of no use when dealing with floating point or 32-bit integer images. Even though the sequential Berger algorithm could be used to build the max-trees of the image partitions, this would not solve the problem because it lies in the merging phase, which is independent of how the sub-trees are built. Fig. 4.3 shows indeed that higher bit depths cause a huge drop in performance already on two threads, at a bit depth of about 16 - 20 bits per pixel. Also in Carlinet and Géraud (2013), the exponential complexity of merging the sub-trees as the number of bits increases was shown for several building algorithms and it was stated that parallel algorithms are unsuitable for data at high bit depths.

The algorithm by Berger et al. (Berger et al., 2007) does not lend itself in a natural way to be parallelized, due to the way an image should be partitioned: the pixels are in fact sorted by intensity and retrieved in descending order during the leaf-to-root process. That leads to a partitioning criterion based on pixel intensity rather than pixel position. A spatial partition would in principle be possible with a trivial implementation: the preservation of the ordering could be achieved by maintaining a semaphore for each pixel, as remarked in Wilkinson et al. (2008). Unfortunately, that would make the approach impracticable because of the high overhead due to

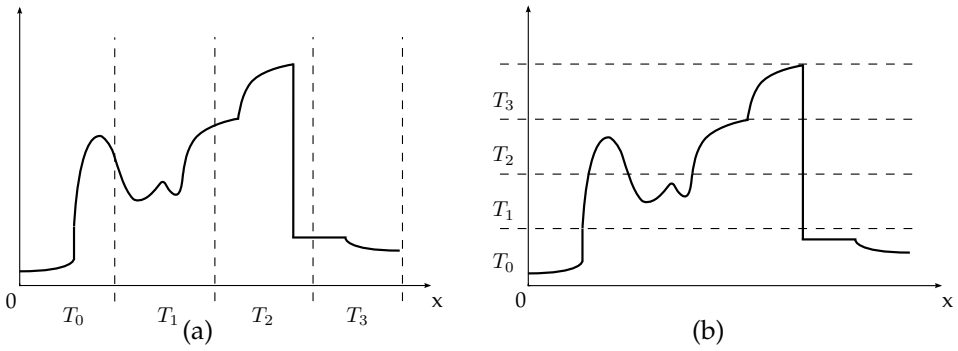


Figure 4.4: Given a 1-D image, an example of spatial partition of the pixels is given in (a). In (b), the image is partitioned according to ranges of intensities.

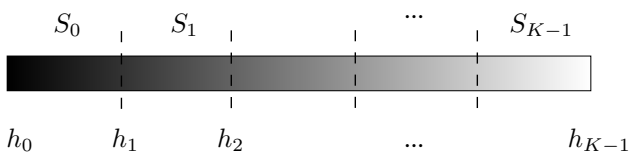


Figure 4.5: Pixels are sorted according to their intensity, from low to high. A partition S_i is made of the pixels with intensity values within h_i and h_{i+1}.

the many synchronizations and locking sections needed. Another solution could be to assign a thread to every maximum in the image. Again, the ordering could be preserved by putting a barrier synchronization method on every intensity level change: this way, the leaf-to-root process would proceed in parallel, intensity by intensity. Mutual exclusion structures would be needed on every pixel that shows a level change with respect to its neighbouring pixels, leading in the worst case to having a barrier for every pixel and therefore no parallelism at all. Due to the high number of mutexes, it does not look like a feasible algorithm. Therefore, those possibilities were not explored any further.

4.6 The parallel hybrid algorithm

To tackle the merging problem, we start first from computing a quantized version $\lfloor f$ of the input image f. A tree called *pilot* max-tree can be built in parallel using any existing algorithm due to the low number of quantized values. The merging phase is not an issue here. We chose the algorithm illustrated in Section 4.4.3, because we found it faster than other methods on such low-quantized values. The image is divided into spatial partitions as in Fig. 4.4a, one for each thread. The next

step is a parallel refinement stage that yields the final max-tree, with the correct attributes. This process can be seen as a "refinement" operation that shapes the tree of the quantized image into the max-tree of the original image. The *pilot* max-tree allows for correct attribute computation and merging of the sub-trees. Each thread of the refinement stage uses the sequential Berger algorithm of Section 4.4.2 on the pixels of its partition and takes advantage from the *pilot* max-tree to compute correctly the attributes and merge with the other sub-trees that are being built by the threads. In this parallel approach, a trivial spatial partition is not suitable, because leaf-to-root algorithms like the Berger algorithm work on pixels that have been ordered by intensity. The image must be partitioned into spatially irregular partitions as in Fig. 4.4b, so that each partition contains only the pixels in a given range of intensities and each thread can start the execution from the local maxima present in a given range. As Fig. 4.5 shows, $S_0, ..., S_{K-1}$ form K partitions. S_0 is defined as the set of all the pixel locations with intensities in the interval $[h_0, h_1)$, S_1 contains the pixels with values in the interval $[h_1, h_2)$ and so on, where $h_0 < h_1 < h_2$ and h_0 is the lowest intensity in the image. The set of the intensity values in a partition S_i is indicated with H_i. Let K be the number of threads of the parallel refinement stage. The total number of partitions that is created must be equal to K, so that every thread handles all the pixels within a given range of intensities. The intensity ranges H_i are chosen so that about the same number of pixels is present within each partition. It is not always possible to achieve a perfect load balance, since it is constrained by the fact that there cannot be any overlap among the intensity ranges in two different sets H_i. Since the refinement stage of our parallel solution is based on the Berger algorithm, pixel coordinates will need to be sorted by their intensity value in the *original* image. A parallel stable radix sort (Cormen et al., 2001) was used. It relies on the parallelization of the inner counting sort algorithm (Amato et al., 1998) used iteratively within the radix sort. Radix sort was originally developed for integers: we adapted it to support floating point values following the methods in Terdiman (2000) and Herf (2001), also mentioned in other works (Lengyel, 2011; Ha et al., 2009). We are going to explain more in detail the *pilot* max-tree and the refinement stage. We highlight that without the support of the *pilot* tree, the fact of having intensity-based partitions is not enough per se to avoid the high cost of merging the sub-trees of each S_i. When two regions that belong to a given partition are separated by a peak with maximum higher than the partition they belong to, it would be necessary to go down through all the intensity levels to check at which point the regions should be merged together. The *pilot* max-tree structure was designed exactly to reduce the cost of such operations.

Algorithm 4.5. FLOOD_PILOT procedure for the *pilot* max-tree.

```
1: procedure FLOOD_PILOT(Level lev, Partition P, Attribute thisarea)
2:     area ← thisarea;
3:     while Queue at level lev is not empty do
4:         Extract pixel p from the Queue at level lev;
5:         for all neighbours q of p with q ∈ P do
6:             if isVisited[q] = false then
7:                 isVisited[q] ← true;
8:                 if levelroot[g(q)] = not set then
9:                     levelroot[g(q)] ← q;
10:                else
11:                    KEEPLOWESTLEVELROOT(q);
12:                end if
13:                Add q to the Queue at level f̄(q);
14:                if f̄(q) > lev then
15:                    childarea =← 0;
16:                    while f̄(q) > lev do
17:                        FLOOD_PILOT(f̄(q), P, childarea);
18:                    end while
19:                    area ← area + childarea;
20:                end if
21:            end if
22:        end for
23:    end while
24:    m ← lev − 1;
25:    while m ≥ 0 ∧ levelroot[m] = not set do
26:        m ← m − 1;
27:    end while
28:    if m ≥ 0 then
29:        node_qu[levelroot[lev]].parent ← levelroot[m];
30:    end if
31:    node_qu[levelroot[lev]].area = area;
32:    levelroot[lev] ← not set;
33:    thisarea ← area;
34: end procedure
```

4.7 The *pilot* max-tree

The *pilot* max-tree is simply the max-tree of the quantized image $\overline{f}$. We chose the root-to-leaf approach by Wilkinson et al. (2008), illustrated in Section 4.4.3, to build it. The nodes of the *pilot* max-tree are stored in the array *node_qu*, whose length is equal to the number of pixels. Due to the low number of intensity levels in $\overline{f}$, the merging phase does not represent a problem and the algorithm of Section 4.4.3 is perfectly suitable. We underline that the quantized image does not show new peak components that are not present in the original image. In fact, according to our

Algorithm 4.6. Implements the stricter definition of level root.

```
1: procedure KEEPLOWESTLEVELROOT(Pixel q)
2:     cond1 ← f(q) < f(levelroot[f̄(q)]);
3:     cond2 ← f(q) = f(levelroot[f̄(q)]) ∧ q < levelroot[f̄(q)];
4:     if cond1 ∨ cond2 then                          ▷ set the new lowest level root
5:         node_qu[levelroot[f̄(q)]].parent = q;
6:         levelroot[f̄(q)] = q;
7:     end if
8:     node_qu[q].parent = levelroot[f̄(q)];
9: end procedure
```

definition of quantization, the actual hierarchy of components in the original image now belonging to a partition S_i is simply flattened on the ancestor component with lowest intensity in S_i. The same ancestor component is then present in the original image as well as in the quantized image. The flattening can be observed comparing Fig. 4.2a and Fig. 4.2b. Exploiting the fact that there are no new peak components, a relation is enforced between the *level root* nodes (introduced in Section 4.4.1) of the *pilot* tree and those of the final refined tree: the former are a subset of the latter. We will see how the merging phase of the sub-trees and their attributes relies on such nodes. To ensure this correspondence, the only change made on the max-tree implementation in Wilkinson et al. (2008) is a stricter definition of level root of a connected component. The level root node where the attributes (area, in our case) are accumulated is not any more an arbitrary pixel of the component but it is chosen to correspond to the pixel with lowest coordinate among the pixels belonging to the component. Function FLOOD_PILOT in Alg. 4.5 is the same as in Alg. 4.3 except for the call to function KEEPLOWESTLEVELROOT, detailed in Alg. 4.6, that implements the new definition of level root. At the end, the array *node_qu* contains the *pilot* max-tree. The loop with the recursive call to FLOOD_PILOT at line 17 in Alg. 4.5 stops when a local maximum has been reached. We make explicit here that, in the refinement stage, the last processed pixel of every connected component in the *original* image f will be the one with the lowest image coordinate. Since a stable sort was chosen, the pixels with equal value in every original component are sorted so that the pixel with the lowest coordinate appears after all the others with equal intensity. The stricter definition of level root, the quantization step and the stable sort grant that the level roots of the *pilot* max-tree are a subset of the level roots of the refined tree. We note that union-by-rank that makes the algorithm by Najman and Couprie faster (Berger et al., 2007) than the Berger one cannot be used in our solution: using this technique, the correspondence among the level roots in the *pilot* and in the final max-tree would not be guaranteed any more. Union-by-rank could possibly be tweaked, but at the cost of checking for every processed pixel a correspondence to a level root and, anyway, not always returning a balanced set, due to this restriction.

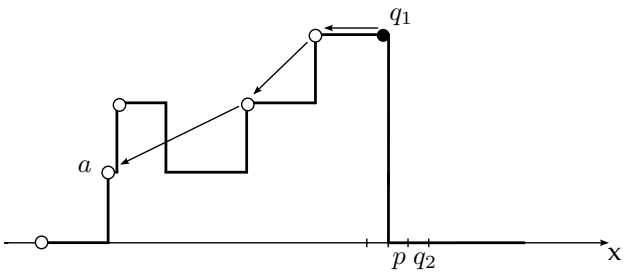


Figure 4.6: The same quantized image as in Fig. 4.2b is reported here. Level roots of the *pilot* max-tree are indicated with open circles. When a pixel p is retrieved from the sorted array, its neighbours q_1 and q_2 are processed, according to the Berger algorithm. Since $\overline{f}(q_1) > \overline{f}(p)$, function DESCENDROOTS is called on the *pilot* max-tree, starting from node $node\_qu[q_1]$. Node $node\_qu[a]$ is returned, because its parent points to the partition where p belongs.

4.8 The refinement stage

The correct max-tree representation of the original image f is the output of the refinement stage. In the refinement stage, the *pilot* max-tree is not modified. Hence, all threads can safely access it without need for synchronization mechanisms. It can be seen indeed as a parallel version of the algorithm proposed by Berger et al. (Berger et al., 2007). The final max-tree nodes (the tree structure) are stored in an array called *node_ref*, with length equal to the image size. Every node corresponds to a pixel, as in the *pilot* max-tree, with the same structure as before (parent pointer, the area attribute value) with the addition of a value to store the intensity of the pixel after filtering. The whole algorithm is detailed in Alg. 4.7. As recalled in Section 4.4.2, our solution uses the same array structure named *zpar* in Section 4.4.2. The final tree is created in parallel by K threads that work on the original pixel values of f. The number of threads is equal to the number of partitions S. Each thread T_i retrieves the sorted pixels belonging to its partition S_i, each one with the same quantized intensity, in descending order. Computation starts from pixels corresponding to the local maxima of the partition S_i. Like in the sequential Berger algorithm, for every pixel p retrieved from the sorted array and belonging to the partition S_i, the set of its neighbour pixels is calculated. As shown in Section 4.4.2, in the Berger algorithm only the already processed neighbours are considered: given that only the pixels with intensity equal to or larger than my current pixel's intensity could have been retrieved (processed) from the sorted array, only two situations are possible, described in Section 4.8.1 and Section 4.8.2.

Algorithm 4.7. Pseudo-code of the refinement stage.

```
1:  procedure REFINEMENT(Thread i, Partition S_i)
2:      lwb ← min(S_i);
3:      upb ← max(S_i);
4:      qi ← quantized intensity managed by T_i;
5:      for j = upb to lwb do
6:          p ← SortedArray[j];
7:          zpar[p] = p;
8:          for all neighbours q of p do
9:              if f̄(q) > qi then                          ▷ case of Section 4.8.2
10:                 desc ← DESCENDROOTS(q, i);
11:                 if node_ref[desc].parent = not set then
12:                     node_ref[desc].parent ← p;
13:                     node_ref[p].Area ← node_ref[p].Area + node_qu[desc].Area;
14:                 else
15:                     z ← FINDROOT(node_ref[desc].parent);
16:                     if z ≠ p then
17:                         node_ref[z].parent ← p;
18:                         zpar[z] ← p;
19:                         node_ref[p].Area ← node_ref[p].Area + node_ref[z];
20:                     end if
21:                 end if
22:             else if f̄(q) = qi then                     ▷ case of Section 4.8.1
23:                 if zpar[q] ≠ −1 then
24:                     r ← FINDROOT(q);
25:                     if r ≠ p then
26:                         node_ref[r].parent ← p;
27:                         zpar[r] ← p;
28:                         node_ref[p].Area ← node_ref[p].Area + node_ref[r].Area;
29:                     end if
30:                 end if
31:             end if
32:         end for
33:     end for
34: end procedure
35:
36: procedure DESCENDROOTS(Pixel q, int i)
37:     c ← q;
38:     while f̄(node_qu[c].parent) > i do
39:         c ← node_qu[c].parent;
40:     end while
41:     return c;
42: end procedure
```

4.8.1 Intensity $\overline{f}(q) \in H_i$

If the intensity $\overline{f}(q)$ of the neighbour pixel q belongs to the set H_i managed by thread T_i, the computation of the tree continues as in the sequential Berger algorithm. This is detailed at lines 22-29 of the pseudo-code in Alg. 4.7.

4.8.2 Intensity $\overline{f}(q) \notin H_i$

Let us define $h_m^i = \textbf{min}\,(H_i)$ and $h_M^i = \textbf{max}\,(H_i)$, the minimum and maximum intensity carried by pixels in S_i. As in the Berger algorithm, only the neighbours already visited must be processed. Since pixels must be processed in decreasing order of intensity, if $\overline{f}(q) < h_m^i$ then neighbour q is considered as not yet been visited and the computation goes on with retrieving the next neighbour.

On the other hand, if $\overline{f}(q) > h_M^i$ then q is considered visited, since it has a higher intensity. In this case, a neighbour q belongs to a partition S_j with $i < j$. The sections of the tree that are being built by T_i and T_j must be now be merged. In the sequential algorithm, since all the pixels are processed sequentially from the highest to the lowest intensity, the pixel q was indeed visited before. In the parallel algorithm, a tricky situation is encountered: pixels in S_i are processed concurrently with the pixels in S_j, and not after. Moreover, another problem is faced: the procedures of merging and updating the attributes take place while the sub-trees of partitions S_i and S_j are still being built and the attributes of a component have not been determined completely. To handle this situation, the *pilot* max-tree is used to retrieve the attribute of the closest descendant of the component to which p belongs in the *pilot* max-tree, and to drive the merging of both sub-trees and attributes. For example, in case of the area attribute, the area of the closest descendant is correct and independent of the actual hierarchy of the components at higher levels.

These steps are detailed in lines 9-21 of Alg. 4.7. Given a pixel $p \in S_i$ and a neighbour pixel q with higher quantized intensity, the function DESCENDROOTS parses the *pilot* max-tree structure starting from the node $node\_qu[q]$. The pixel $desc$ with lowest coordinate among the pixels of the quantized component is returned. The node $node\_ref[desc]$ corresponds to the closest *descendant* of the peak component that contains p, as Fig. 4.6 illustrates. Index $desc$ addresses a level root in the *pilot* max-tree and therefore it must also address a level root of the refined tree, as explained in Section 4.7. The parent pointer of $node\_ref[desc]$ is then checked, at line 11 of Alg. 4.7. If it was not set, p is set as parent and the node $node\_qu[desc]$ of the *pilot* max-tree is used to update the attributes consistently. If the parent pointer has already been set, the algorithm proceeds as in the Berger algorithm: function FINDROOT (same as in Alg. 4.1) is called on $node\_ref[desc].parent$ to retrieve the current parent pointer z. If z is different from the current pixel p, its parent is set to p and the attributes of the $node\_ref[p]$ in the final tree are merged with the attributes of

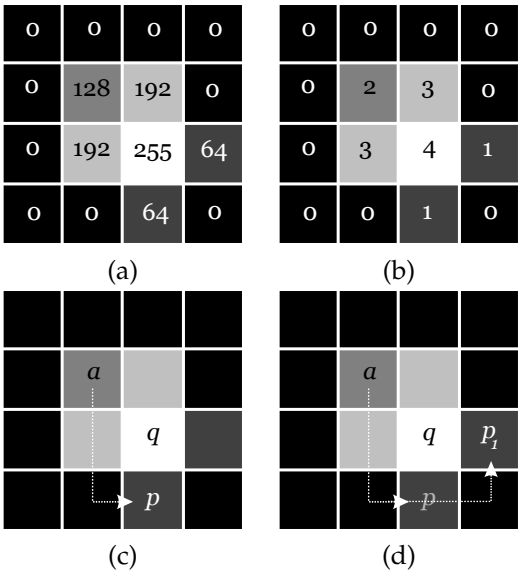


Figure 4.7: (a) original image; (b) a quantization of (a); (c) pixel p is extracted from the sorted array and its neighbour q that belongs to a higher partition is processed. Pixel a is the closest descendant of p, returned by DESCENDROOTS. If $node\_ref[a]$ has no parent set yet, then the pixel p is assigned as parent, see white dotted arrow. (d) Later, pixel p_1 will be extracted, its neighbour q processed and once again a is returned by DESCENDROOTS. Its parent pointer was set in step (c). Therefore, as in the Berger algorithm, pixel p_1 is accessed and p_1 is set as parent of $node\_ref[p]$.

$node\_ref[z]$. Fig. 4.7 illustrates the two cases above.

4.9 Thread-safety and correctness

The first possible reason of concern stems from which descendant node is returned by the DESCENDROOTS, at line 10 in Alg. 4.7: if two pixels are processed and have same descendant as neighbour, two situations arise. If the two pixels carry the same quantized value, they will just be processed sequentially by the same thread, thus concurrent writing on $node\_ref[desc]$ at line 12 and 13 will not happen. If the two pixels are being processed by two different threads, again no race condition occurs: the two calls to the DESCENDROOTS function will always return two different descendant nodes. In fact, the closest descendant of two components at two different quantized intensity levels simply cannot be the same. The DESCENDROOTS function itself has a read-only access to the *node_qu* structure. Calls to the FINDROOT function are also thread-safe. In fact, every thread sets only the entries of the *zpar* structure corresponding to the locations of the pixels under its partition: no mutex

structure is necessary on *zpar*. The only mutexes required by both the *pilot* max-tree building and the refinement stage are only the ones required to flag that the threads completed their task and processed all the pixels in their partition.

The hybrid algorithm is based on two algorithms whose correctness was demonstrated in their respective papers in Wilkinson et al. (2008) and Berger et al. (2007). The refinement stage follows the same procedure as in Berger et al. (2007) when $\overline{f}(q) \in H_i$. The only difference with the original algorithm happens when $\overline{f}(q) \notin H_i$ and the parent of the node returned by `DescendRoots` is not set: in this case, the correspondence between the level roots of the *pilot* and the final tree grants that the merging points are the correct ones and so is the whole algorithm. A more extensive proof can be subject for a future work.

4.10 Time complexity and memory use

The time complexity of the parallel algorithm is defined by the complexity if its three main steps: sorting the pixels, building the *pilot* max-tree, and building the final refined tree. The parallel radix sort can be decomposed into three phases, each one executed at each iteration of the radix sort. In the first phase, each thread in parallel implements the counting sort on its local partition: the complexity of this operation is $O(N/K)$, where N is number of pixels in the image and K the number of threads. When every thread has completed its partial histogram, thread T_0 computes the whole image histogram by summing the local histograms in $O(K \cdot 2^r)$, calculates the prefixes in $O(2^r)$ and signal them to the other threads in $O(K \cdot 2^r)$. Value r is equal to 16: decomposing the image data types in chunks of 16 bits has proven to result in a cache-friendly size of the histograms that are used in the counting sort algorithms. It also limits the number of iterations to at most four in case of 64 bit `double` values. Lastly, every thread writes the pixel positions in the sorted array in parallel in $O(N/K)$. A similar and more detailed analysis was presented in Amato et al. (1998).

The complexity related to building the *pilot* max-tree in parallel can be summarised as follows. On K threads, the worst case time complexity of the building phase of each tree for every partition of the quantized image (therefore with $G = K$ intensities) is $O(N(C+G)/K)$ with C the connectivity and N the number of pixels, which reduces to $O(NC/K + N)$, since $G = K$. Interestingly, the second term of the addition shows no parallelism. We recall here that this is a worst-case scenario, that is rarely seen in practice. We refer to the description in Meijster and Wilkinson (2002); Wilkinson et al. (2008) for more details. As for the merging phase, if K neighbouring partitions have J bridging edges, the total merging has complexity $O(JG \log N \log K)$ (Wilkinson et al., 2008). For every merge, we need to go down O(G) nodes, in a worst case scenario. Since the image was quantized in K intensi-

ties, usually equal to 16, 32, 64, the merging phase is not excessively costly.

The refinement stage shows aspects in common with the Berger algorithm and its complexity analysis is similar to the one in Najman and Couprie (2006). It is known from Berger et al. (2007) that the algorithm by Berger et al. has quasi-linear complexity, once the pixels have been sorted, if union-by-rank (Tarjan, 1975) is applied. In our case, as mentioned at the end of Section 4.7, the ranking technique is disabled and the Berger algorithm has $O(N \log N)$ complexity. In our parallel implementation, the only difference with the original sequential Berger algorithm is the function DESCENDROOTS that parses the *pilot* max-tree. This function is not in the set of operations defined by the union-find algorithm for which quasi-linearity was demonstrated in Tarjan (1975). Its complexity is then analysed in the following. For every neighbouring pixels with intensity larger than the intensity of the current pixel, the node hierarchy in the *pilot* max-tree is parsed from the quantized level of the neighbour to the quantized level of the current pixel. Therefore, the time complexity of DESCENDROOTS cannot be larger than $O(GCN/K)$ with C number of neighbours and K the number of threads. Since $G = K$, the complexity is equal to $O(CN)$, linear in the number of pixels in the worst case scenario with CN/K higher neighbouring pixels.

Memory-wise, the parallel algorithm needs two arrays to store the original image and the quantized image, the sorted array and the *zpar* array: they require together $4N$ memory space, assuming that pixel values, pixel coordinates and (area) attribute are encoded with the same number of bits. Moreover, the *pilot* max-tree and the final tree require two arrays of length N, the former with each entry containing the parent index and the attribute value, the latter the parent index, the attribute value and the value after filtering. In total, they require additional $5N$ memory space. The sequential Berger algorithm required $6N$ memory space: it needs the array for the original image, the sorted array, the *zpar* array and the tree structure as an array of length N, each one containing attribute value, parent index and value after filtering. Summarising, the parallel algorithm needs $3N$ memory more than the Berger algorithm due to the arrays to store the quantized image and the *pilot* max-tree.

4.11 Performance testing

The proposed parallel algorithm was implemented in C with POSIX Threads. The code is available at `http://www.cs.rug.nl/~michael/ParMaxTree` Timings were performed on a Dell R815 Rack Server with four 16-core AMD Opteron processors and 512 GB of RAM memory. It comes with 32 floating point units, each one shared by a pair of cores. A measure of the performance is given in processed megapixels per second (Mpx/s), to normalize for image size. The attribute com-

Image Name	Data type	Bits per pixel	Megapixels	Type	Image content	Source
Float1	float	32	870	2D	Random pixel values	
Float4	float	32	3480	2D	Random pixel values	
Double1	double	64	870	2D	Random pixel values	
Double4	double	64	3480	2D	Random pixel values	
ESO	float	32	7143	2D	Luminosity from RGB channels	ESO Paranal Observatory (Saito et al., 2012)
PRAGUE	float	32	4032	2D	Luminosity from RGB channels	Jeffrey Martin / 360cities.net
LOFAR	float	32	1134	3D	Field of radio sources	LOFAR radio telescope (LOFAR, 2016)
Aneurysm	float	32	134	3D	Blood vessels with aneurysm	www.volvis.org

Table 4.1: A summary of the images used for testing the performance of the algorithm.

Image Name	Data type	Mpx/s (Berger)	Mpx/s (Hybrid 64 threads)	Time (Berger)	Time (Hybrid 64 threads)
Float1	float	0.45	5.47	32 min	2.6 min
Float4	float	0.28	4.65	3 hours 28.1 min	12.5 min
Double1	double	0.42	4.52	34.46 min	3.21 min
Double4	double	0.26	4.09	3 hours 46.23 min	14.2 min
ESO	float	0.28	5.86	7 hours 2 min	20.3 min
PRAGUE	float	0.44	7.16	2 hours 32.9 min	9.39 min
LOFAR	float	0.23	5.42	1 hour 21.9 min	3.49 min
Aneurysm	float	0.40	6.42	5.56 min	20.92 seconds

Table 4.2: Performance in megapixels per second and completion time of the sequential Berger algorithm and the hybrid algorithm presented in this chapter.

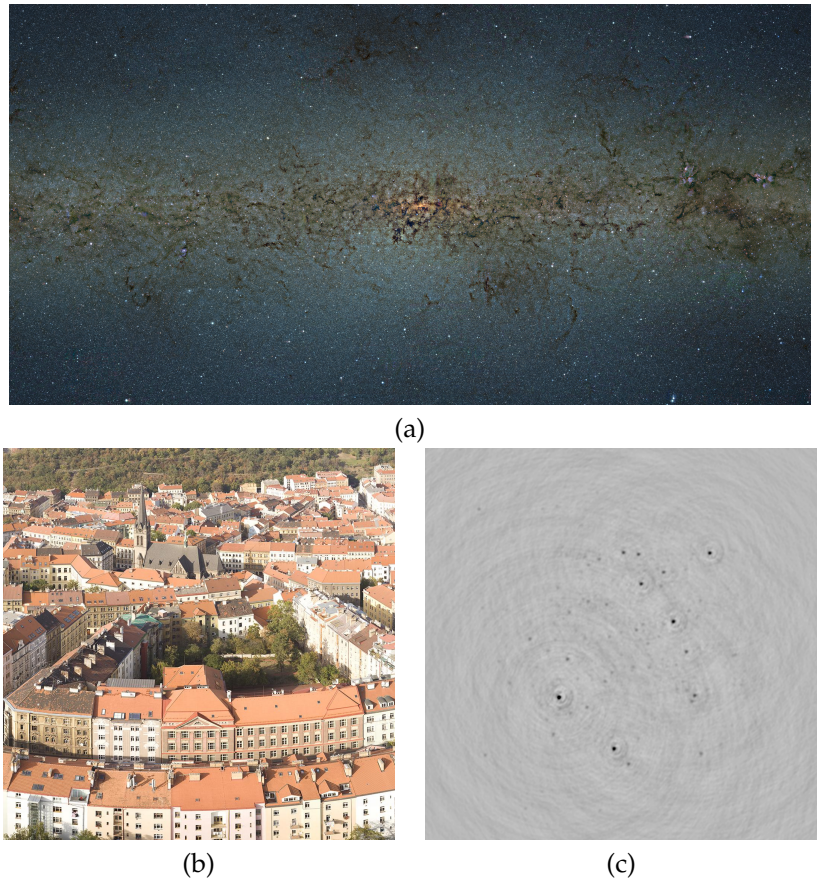
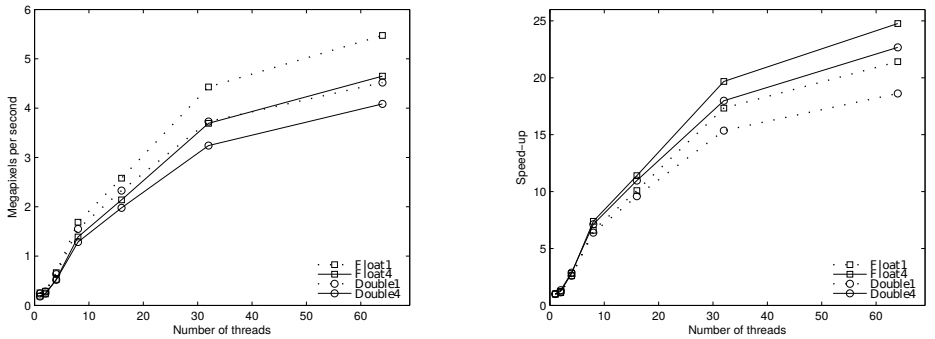


Figure 4.8: (a) ESO image - *Release No.: eso1242. Credit: ESO/VVV, Survey/D. Minniti. Acknowledgement: Ignacio Toledo, Martin Kornmesser*; (b) a sample section of the cropped PRAGUE image. (c) average of the 1080 frames of the LOFAR cube.

puted is area or volume. Other attributes, e.g. Hu's moments (Hu, 1962), could be easily computed. The minimum timing of a series of runs of the algorithm was considered. Experiments with 1, 2, 4, 8, 16, 32 and 64 threads were performed. Table 4.1 summarises all the images tested, with their resolution and data type. Table 4.2 reports the performance and wall-clock times for all the images tested.

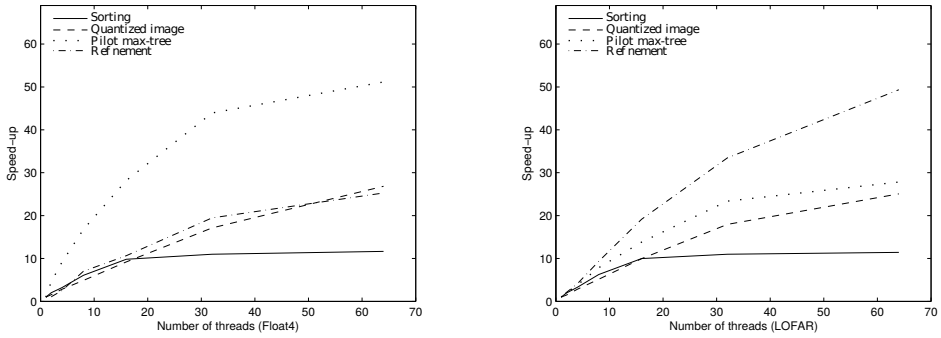
4.11.1 Performance tests on simulated images

Figure 4.9 shows the results for the images in the first four rows of Table 4.1. These images have a resolution of about 1 Gigapixel (Gpx) and 4Gpx and they are re-



(a) (b)

Figure 4.9: Performance measurements: processed Mpx per second (a) and speed-up (b) for Float1, Float4, Double1 and Double4. The pixels carry randomly generated floating points with single (square marker) and double (circle marker) precision.



(a) (b)

Figure 4.10: Speed-up of each of the four phases of the parallel algorithm as function of the number of threads on (a) Float4 image and (b) LOFAR image.

ferred to as Float1, Float4, Double1 and Double4, according to their data types and sizes. Bit depths of 32 and 64 bits per pixel correspond to `float` and `double` types, respectively. They contain randomly generated pixel values with a uniform distribution. On the images with `double` values, Double1 and Double4, our parallel solution obtains a throughput of 4.52 Mpx/s and 4.09 Mpx/s on 64 threads, respectively. On the Double4 image, the wall-clock time needed to build the component tree drops from 5 hours and a half to 14 minutes, going from one to 64 threads. The sequential Berger algorithm is the fastest sequential algorithm for XDR images. It yields a throughput of 0.42 Mpx/s and 0.26 Mpx/s on Double1 and Double4 im-

ages, respectively. Wall-clock times are 34 minutes for the Double1 image and about 4 hours for the Double4 image. A similar behaviour can be observed for the other two images with `float` values, Float1 and Float4. Other tests were performed on satellite imagery, changing the values from 8-bit integers into `float` values and filling lower order decimals so to roughly preserve the image structures, but with the same intensity rarely repeated twice. They showed indeed comparable performance, perhaps even slightly better due to the smoother features that are rarely found in random-valued images: the number of calls to the DESCENDROOTS is lower. For the four simulated images mentioned above, the influence of the pixel data type is observed only in the sorting phase: when dealing with `double` values, the radix sort has to perform two iterations more than in the `float` case. The other three steps, the creation of the quantized image, the *pilot* max-tree construction and the refinement stage are independent of the pixel data type and they take the same time.

As shown in Fig. 4.9b, the speed-up is close to optimal up to 8 threads, it keeps on being at a good level up to 16 threads and then it starts to decrease. Looking at the speed-up of the four phases separately in Fig. 4.10, we see that the creation of the quantized image, the building of the *pilot* max-tree and the refinement stage scale better than the sorting phase. In the sorting phase, every iteration of the radix sort needs a barrier synchronization methods that degrade the parallelism. The speed-up values got in the parallel sorting algorithm confirm the values obtained in an other work (Rashid et al., 2010) that describes a very similar implementation. Moreover, the speed-up of the phase where the quantized image is created is not as high as expected: probably the huge number of memory accesses in a short time fills the memory bandwidth of the machine, thus preventing the performance of this "embarrassingly parallel" section of the code from scaling properly. It is worth to point out that on 64 threads the speed-up values of the section related to the *pilot* max-tree vary greatly from 25 to 50 between Fig. 4.10a and Fig. 4.10b. We observed that for image sizes around 4Gpx or more, the time spent on building the *pilot* max-tree on a single thread is about 4 times longer than on 2 threads, rather than 2 times as expected. It could be due to cash trashing or latency issues as the processor accesses farther memory banks. Function KEEPLOWESTLEVELROOT is not the reason: the same performance are observed also in the max-tree algorithm in Wilkinson et al. (2008), where this function is not called.

4.11.2 Considerations on completion time and speed-up

For the Double4 image on 64 threads, the fractions of time spent by each phase with respect to a complete execution of the algorithm are: sorting the pixels, 25%; creating the quantized image, 1%; creating the *pilot* max-tree, 6%; refinement stage, 68%. For the Float4 image on 64 threads, results are: sorting the pixels, 15%; creating

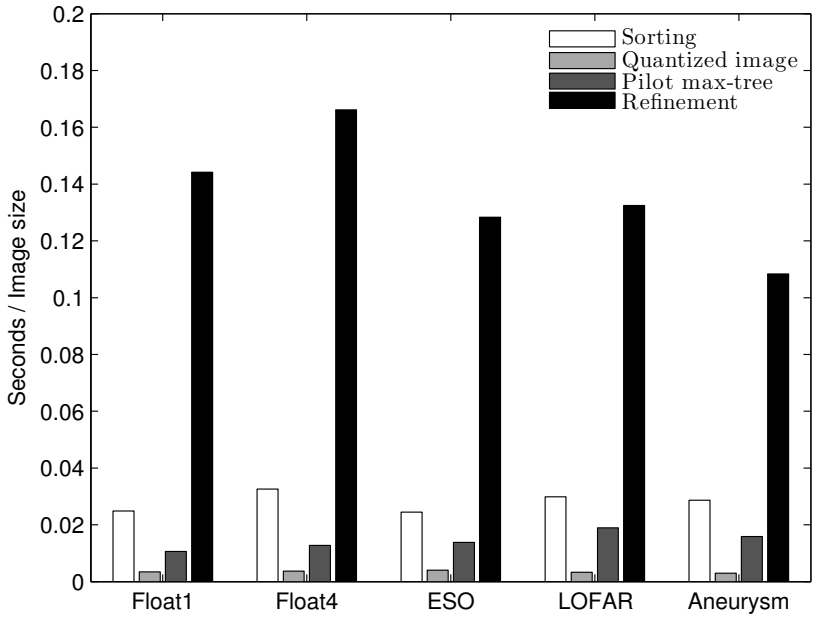


Figure 4.11: The completion time of each separate phase of the parallel algorithm, normalized with the image size.

the quantized image, 1%; creating the *pilot* max-tree, 6%; refinement stage, 78%. The plot in Fig. 4.11 shows the impact of each phase on the total completion time of the algorithm for several images. The refinement stage takes on average ten times more than the time needed to build the *pilot* max-tree. The impact of the function KEEPLOWESTLEVELROOT was also tested. Overall, it affects the completion time of less than 1%.

In an earlier implementation, it was noticed that the *pilot* max-tree had an extremely poor speed-up. That happened because, after the tree was built, a technique called *level root fix* (Wilkinson et al., 2008) or *canonization* (Berger et al., 2007) was applied. The purpose of this technique is to ensure that the parent pointer of every node points directly to the level root node of its component (or the component below). At first, it was thought to be useful because it would ease the task of the DESCENDROOTS function, lowering the number of hops on the *pilot* max-tree and thus providing robustness towards worst-case situations to the algorithm. The level root fix can be done in parallel, but from experiments performed on several images, it turned out that there was no real benefit coming from its use, especially on higher degrees of parallelism. The plot in Fig. 4.12 shows similar completion times for four

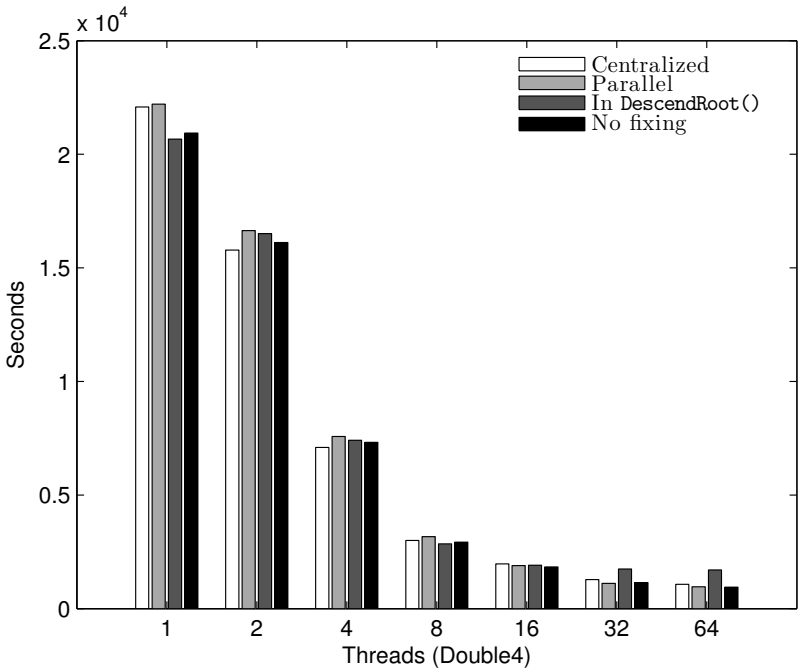
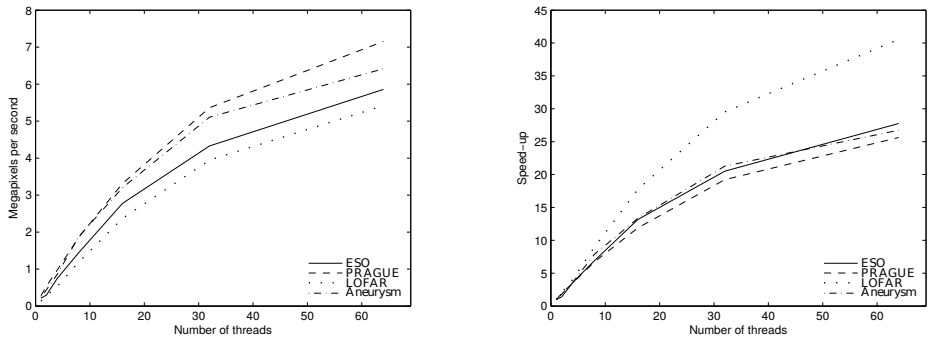


Figure 4.12: Completion time of the parallel algorithm with several *level root fixing* techniques used on image Double4.

different ways to implement the level root fix on the Double4 image: the fixing was done in a centralized way, in a parallel way, within the DESCENDROOTS function and finally without any fixing. With the first two ways, the refinement stage was a bit faster, but the extra time needed by the level root fixing procedures raised the completion time of the *pilot* max-tree and cancelled any benefit. The level root fixing directly in the DESCENDROOTS made the performance worse on a high number of threads. We chose not to apply level root fixing on the *pilot* max-tree.

4.11.3 Performance tests on real-world images

Our parallel algorithm was also tested on four real-world images with very high-dynamic range, summarised in the lower half of Table 4.1. The first image, see Fig. 4.8a (ESO image in Table 4.1), was taken at the ESO Paranal Observatory in Chile by the VISTA infra-red wide-field survey telescope. It portraits more than 84 millions stars in the central regions of the Milky Way (Saito et al., 2012). A section of about 7Gpx was cropped from the original 9Gpx so to fit the memory specifications of our machine when executing the parallel algorithm. It is an RGB image



(a) (b)

Figure 4.13: Performance measurements: processed megapixels per seconds (a) and speed-up (b) computed on the ESO, PRAGUE, LOFAR and Aneurysm images.

that we reduced to one `float` luminance channel, by weighting and summing the channels, as $0.2126R + 0.7152G + 0.0722B$. The final size of the ESO luminance image used in the tests is about 28 GB. The image named PRAGUE in the Table 4.1 is another high-resolution RGB image processed in the same way: it depicts a 4Gpx section of the city of Prague, cropped from the original 18Gpx panorama image of the whole city, courtesy of Jeffrey Martin/360cities.net. A section of the cropped image is shown in Fig. 4.8b. The third image, referred to as LOFAR, see Fig. 4.8c, is a 3D volume. It represents a portion of the sky and it is a 1024 x 1024 (spatial) x 1081 (temporal) image of a field of radio sources, i.e. astronomical objects, corresponding to 6 hours of observation of the LOFAR (LOFAR, 2016) radio telescope in The Netherlands. It contains `float` values. Our code was extended to support 3D volumes, using 6-connectivity. The fourth image is a smaller 3D volume containing an angiography scan of a head with an aneurysm. Its values have 16 bits per pixel: they have been stretched to `float` values on 32 bits, while filling them with random less significant decimals. The plot in Fig. 4.13a shows a throughput of 5.85 Mpx/s and 5.41 Mpx/s on 64 threads for the ESO and LOFAR image, respectively. The aneurysm volume shows a value of 6.41 Mpx/s. All the results show similarities with the results shown in Fig. 4.9 for the images with random `float` values. The LOFAR image shows a better speed-up, about 40 on 64 threads. The refinement stage separately reaches a speed-up of 50 on 64 threads in this case. The throughput of the sequential Berger algorithm is 0.28 Mpx/s and 0.23 Mpx/s for the ESO and LOFAR images, between 20 and 23 times slower than the parallel solution. The same trend appears on the aneurysm image, whose maximum throughput is 6.42 Mpx/s for the parallel algorithm and 0.40 Mpx/s for the sequential algorithm.

The parallel processing of the PRAGUE shows a throughput of 7.16 Mpx/s and 0.44 Mpx/s for the sequential Berger, a higher throughput than the ESO image. Such behaviour was expected because the PRAGUE image contains larger patches with the same tonality. Due to that, the refinement stage is faster because the number of hops in the functions FINDROOT or DESCENDROOTS is lower than in the ESO case: there, the stellar detail increases the number of small components, with large intensity differences with the neighbouring pixels. This was confirmed also by looking at the different phases separately: the refinement stage for the PRAGUE image performs better, while the other phases behave similarly.

We find it interesting to highlight the performance also in terms of absolute execution time. As reported in the last two columns of Table 4.2, the sequential Berger algorithm takes 7 hours and 1 hour and 20 minutes, for the ESO and LOFAR images. Wall-clock times decrease to 20 and 4 minutes, respectively, when our parallel algorithm is run on 64 threads. The PRAGUE image shows similar behaviour, going from 2 hours and a half to 10 minutes on 64 threads, as Table 4.2 summarises. Recently, experiments were made to perform object segmentation in high resolution 3D volumes containing floating point values related to the radio spectral line emission of galaxies (Moschini et al., 2014). In this work, noise background was subtracted from the volume and almost half of the pixels belonged to the root node. The load is highly unbalanced towards the thread managing the 0-valued pixels. A straightforward solution would be to treat those pixels independently, because it is known that they belong to the root node.

4.12 Conclusions

In this chapter we proposed a parallel algorithm to build max-trees of very high-dynamic range images efficiently. Existing parallel methods have shown good performance only up to 16 bits per pixel. Our algorithm combines in a two-step process the root-to-leaf flooding and leaf-to-root merging approaches, hence we named it "hybrid". A max-tree of a quantized version of the input image, called *pilot* max-tree, is built to support the merging of the sub-trees built later with a parallel leaf-to-root approach, efficiently and correctly. It has proven to deal well with even with floating point images or volumes at high resolutions. On 64 threads, speed-up values range between an average of 23 for the simulated images and about 30 for the real-world images tested. It was shown that the max-tree can now be computed about 14 times and 20 times faster, on artificial and real-world images respectively, than the fastest sequential method that supports XDR images, implemented by Berger et al. (Berger et al., 2007). Our parallel solution enables the processing of kinds of images that was prohibitive before.

In future work, optimal load balance and quantization for highly skewed grey-

level distributions will be investigated. A further development could be to apply the same parallel approach to alpha-trees (Ouzounis and Soille, 2012). They are graph structures extensively used in remote-sensing image analysis. Even though remote-sensing images do not always show high-dynamic range values, the dissimilarity metrics used to identify the component partitions that generate branches often present floating point values. Such partitions trees could be built in parallel and merged with a similar technique.