A Comparative Review of Component Tree Computation Algorithms

Edwin Carlinet, Thierry Géraud

A Comparative Review of Component Tree Computation Algorithms

Edwin Carlinet*, *Student Member, IEEE*, Thierry Géraud, *Member, IEEE*

***Abstract*—Connected operators are morphological tools that have the property of filtering images without creating new contours and without moving the contours that are preserved. Those operators are related to the max-tree and min-tree representations of images, and many algorithms have been proposed to compute those trees. However, no exhaustive comparison of these algorithms has been proposed so far, and the choice of an algorithm over another depends on many parameters. Since the need for fast algorithms is obvious for production code, we present an in-depth comparison of the existing algorithms in a unique framework, as well as variations of some of them that improve their efficiency. This comparison involves both sequential and parallel algorithms, and execution times are given w.r.t. the number of threads, the input image size, and the pixel value quantization. Eventually, a decision tree is given to help the user choose the most appropriate algorithm with respect to the user requirements. To favor reproducible research, an online demo allows the user to upload an image and bench the different algorithms, and the source code of every algorithms has been made available.**

***Keywords*—Component tree, max-tree, min-tree, connected operators, filtering, mathematical morphology, algorithms.**

EDICS Categories: ARS-RBS, TEC-PRC, SMR-STM

I. Introduction

In mathematical morphology, connected filters are those that modify an original image by only merging flat zones, hence those that preserve some of the original image contours. Originally, they were mostly used for image filtering [1, 2]. Major advances came from max- and min-tree as hierarchical representations of connected components and from an efficient algorithm able to compute them [3]. Since then, usage of these trees has soared for more advanced forms of filtering: based on attributes [4, 5], using new filtering strategies [3, 6], allowing new types of connectivity [7]. They are also a base for other image representations. In [8] a tree of shapes is computed from a merge of the min- and max- trees. In [9] a component tree is computed over the attributes values of the max-tree. Max-trees have been involved in many applications: computer vision through motion extraction [3], features extraction with MSER [10], segmentation, 3D visualization [11]. With the increase of applications comes an increase of data type to process: 12-bit images in medical imagery [11], 16-bit or float images in astronomical imagery [12], and even multivariate data with special ordering relation [13]. With the improvement of optical sensors, images are getting bigger (so do image data sets) which argues for the need for fast algorithms. Many algorithms have been proposed to compute the max-tree efficiently but only partial comparisons have been proposed. Moreover, some of them are dedicated to a particular task (e.g., filtering) and are unusable for other purposes.

In a short paper [14], we have presented a first comparison of many state-of-the-art max-tree algorithms in a unique framework, i.e., same architecture, same language (C++) and same outputs. Yet this comparison was performed on a single image, the pseudo-code of all the algorithms were not listed, and the description of those algorithms and their comparison were short. This paper aims at correcting those three drawbacks, so it presents a full and exhaustive comparative review of the state-of-art component tree computation algorithms.

The paper is organized as follows. Section II recalls basic notions and manipulations of max-tree. Section III describes the algorithms and implementations used in this study; in particular, a new technique that improves the efficiency of union-find-based algorithms is presented in Section III-A4. Section IV is dedicated to the comparison of those algorithms both in terms of complexity and running times through experimentations. Last we conclude in Section V.

II. A tour of max-tree: definition, representation and algorithms

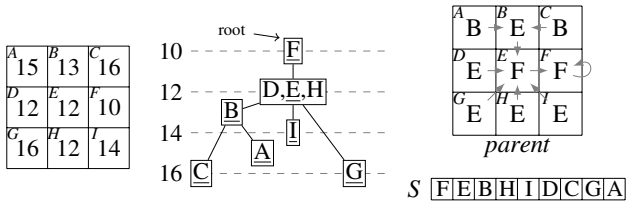
A. Basic notions for max-tree

Let $f : \Omega \to V$ be an image on a regular domain Ω, having values on a totally preordered set $(V, \leq)$ and let $\mathcal{N}$ be a neighborhood on Ω. Let $\lambda \in V$, we note $[f \leq \lambda]$ the set $\{p \in \Omega,\ f(p) \leq \lambda\}$. Let $X \subset \Omega$, we note $CC(X) \subset \mathcal{P}(\Omega)$ the set of connected components of X w.r.t. the neighborhood $\mathcal{N}$; $\mathcal{P}(\Omega)$ being the power set of all the possible subsets of Ω. $\{\, CC([f = \lambda]),\ \lambda \in V \,\}$ are *level components* and $\Psi = \{\, CC([f \geq \lambda]),\ \lambda \in V \,\}$ (resp. $\leq$) is the set of upper components (resp. lower components). The latter endowed with the inclusion relation form a tree called the max-tree (resp. min-tree). Since min- and max-trees are dual, this study obviously holds for min-tree as well. Finally, the peak component of p at level λ noted P_p^λ is the upper component $X \in CC([f \geq \lambda])$ such that $p \in X$.

B. Max-tree representation

Berger et al. [12], and Najman and Couprie [15] rely on a simple and effective encoding of component-trees using an

* edwin.carlinet@lrde.epita.fr
The authors are both with EPITA Research and Development Laboratory (LRDE), 14-16 rue Voltaire, FR-94276 Le Kremlin-Bicêtre, France, and with Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge, A3SI, ESIEE Paris, 2 bd Blaise Pascal, B.P. 99, FR-93162 Noisy-le-Grand Cedex, France.



(a) Original image f (b) Max-tree of f (c) Max-tree representation

Fig. 1. Representation of a max-tree using the 4-connectivity with a parent image and an array. Canonical elements are underlined.

image that stores the *parent* relationship. The latter exists between two components A and B whenever A is *directly* included in B (*parent* is actually the covering relation of $(\Psi, \subseteq)$). An upper component is represented by a single point called the *canonical element* [12, 15] or *level root*. Let two points $p, q \in \Omega$, and p_r be the root of the tree. We say that p is canonical if $p = p_r$ or $f(parent(p)) < f(p)$. A $parent$ image shall satisfy the following three properties: 1) $parent(p) = p \Rightarrow p = p_r$ - the root points to itself and it is the only point verifying this property - 2) $f(parent(p)) \leq f(p)$ and 3) $parent(p)$ is canonical.

Furthermore, having just the $parent$ image is an incomplete representation since it is not sufficient to easily perform classical tree traversals. For that, we need an extra array of points, $S : \mathbb{N} \to \Omega$, where points are stored so that $\forall i, j \in \mathbb{N}\ i < j \Rightarrow S[j] \neq parent(S[i])$. Thus browsing S elements allows to traverse the tree downwards i.e from the top (the root) to the bottom of the tree (the leaves). On the contrary, a reverse browsing of S is an upward tree traversal. Note that having both S and $parent$ thus makes it useless to store the children of each node. Fig. 1 shows an example of such a representation of a max-tree. This representation only requires $2nI$ bytes memory space where n is the number of pixels and I the size in bytes of an integer, since points stored in S and $parent$ are actually positive offsets in a pixel buffer. The algorithms we compare have all been modified to output the same tree encoding, that is, the couple ($parent$, S).

C. Attribute filtering and reconstruction

A classical approach for object detection and filtering is to compute some features called attributes on max-tree nodes. A usual attribute is the number of pixels in components. Followed by a filtering, it leads to the well-known area opening. More advanced attributes have been used like elongation, moment of inertia [16] or even mumford-shah like energy [9]. Some max-tree algorithms [17, 10] construct the $parent$ image only; they do not compute S. As a consequence, they do not provide a "versatile" tree, i.e., a tree that can be easily traversed upwards and downwards, that allows attribute computation and non-trivial filtering. Here we require every algorithms to output a "complete" tree representation ($parent$ *and* S) so that it can be multi-purposedly usable. The rationale behind this requirement is that, for some applications, filtering parameters are not known yet at the time the tree is built (e.g., for interactive visualization [11]). In the algorithms we compare in this paper, no attribute computation nor filtering are performed during tree construction for clarity reasons; yet they can be augmented to compute attribute and filtering at the same time. Fig. 2 provides an implementation of attribute computation and direct-filtering with the representation. $\hat{a} : \Omega \times V \to \mathcal{A}$ is an application that projects a pixel p and its value $f(p)$ in the attribute space $\mathcal{A}$. $\hat{+} : \mathcal{A} \times \mathcal{A} \to \mathcal{A}$ is an associative operator used to merge attributes of different nodes. COMPUTE-ATTRIBUTE starts with computing attributes of each singleton node and merges them from leaves toward root. Note that this simple code relies on the fact that a node receives all information from its children before passing its attribute to the parent. Without any ordering on S, it would not have been possible. DIRECT-FILTER is an implementation of direct filtering as explained in [3] that keeps all nodes passing a criterion λ and lowers nodes that fail to the last ancestor "alive". This implementation has to be compared with the one in [17] that only uses $parent$. This one is shorter, faster and clearer above all.

```
function COMPUTE-ATTRIBUTE(S, parent, f)
    p_root ← S[0]
    for all p ∈ S do attr(p) ← â(p, f(p))
    for all p ∈ S backward, p ≠ p_root do
        q ← parent(p)
        attr(q) ← attr(q) +̂ attr(p)
    return attr

function DIRECT-FILTER(S, parent, f, attr)
    p_root ← S[0]
    if attr(p_root) < λ then out(p_root) ← 0
    else out(p_root) ← f(p_root)
    for all p ∈ S forward do
        q ← parent(p)
        if f(q) = f(p) then out(p) ← out(q)          ▷ (1)
        else if attr(p) < λ then out(p) ← out(q)     ▷ (2)
        else out(p) ← f(p)                           ▷ (3)
    return out
(1) p not canonical, (2) Criterion failed, (3) Criterion passed
```

Fig. 2. Computation of attributes and filtering.

III. MAX-TREE ALGORITHMS

Max-tree algorithms can be classified in three classes:

Immersion algorithms. It starts with building N disjoints singleton for each pixel and sort them according to their gray value. Then, disjoint sets merge to form a tree using the union-find algorithm [18, 19].

Flooding algorithms. A first scan allows to retrieve the root which is a pixel at lowest level in the image. Then, it performs a propagation by flooding first the neighbor at highest level i.e. a depth first propagation [3, 20].

Merge-based algorithms. They divide an image in blocks and compute the max-tree on each sub-image using another max-tree algorithm. Sub max-trees are then merged to form the tree of the whole image. Those algorithms are well-suited for parallelism using a *map-reduce* (or *divide-and-conquer*) approach [21, 17]. When blocks are image lines, dedicated 1D max-tree algorithms can be used [22, 23, 24].

A. Immersion algorithms

1) Basic principle: Berger et al. [12], Najman and Couprie [15] proposed two algorithms based on Tarjan's union-find. They consist in tracking disjoints connected components and merge them in a bottom-up fashion. First, pixels are sorted in an array S where each pixel p represent the singleton set $\{p\}$. Then, we process pixels of S in backward order. When a pixel p is processed, it looks for already processed neighbors ($\mathcal{N}(p)$) and merges with neighboring connected components to form a new connected set rooted in p. The merging process consists in updating the *parent* pointer of neighboring component roots toward p. Thus, the union-find relies on three processes: `make-set(parent, x)` that builds the singleton set $\{x\}$, `find-root(parent, x)` that finds the root of the component that contains x, and `merge-set(parent, x, y)` that merges components rooted in x and y and set x as the new root. Based on the above functions, a simple max-tree algorithm is given below:

```
procedure MAXTREE(f)
    S ← sort pixels increasing
    for all p ∈ S backward do
        make-set(parent, p)
        for all n ∈ N_p processed do
            r ← find-root(parent, n)
            if r ≠ p then merge-set(parent, p, r)
```

`find-root` is a $O(n)$ function that makes the above procedure a $O(n^2)$ algorithm. Tarjan [19] discussed two important optimizations to avoid a quadratic complexity: root path compression and union-by-rank.

When *parent* is traversed to get the root of the component, points of the path used to find the root collapse to the root the component. However, path compression should not be applied on $parent$ image because it removes the hierarchical structure of the tree. As consequence, we apply path compression on an intermediate image $zpar$ that stores the root of disjoints components. Path compression bounds union-find complexity to $O(n \log n)$ and has been applied in [12] and [15].

When merging two components A and B, we have to select one of the roots to represent the newly created component. If A has a *rank* greater than B then $root_A$ is selected as the new root, $root_B$ otherwise. When rank matches the depth of trees, it enables tree balancing and guaranties a $O(n \log n)$ complexity for union-find. When used with path compression, it allows to compute the max-tree in quasi-linear time ($O(n.\alpha(n))$ where $\alpha(n)$ is the inverse of Ackermann function which is very low-growing). Union-by-rank has been applied in [15].

Note that *parent* and *zpar* encode two different things, *parent* encodes the max tree while *zpar* tracks disjoints set of points and also uses a tree. Thus, union-by-rank and root path compression shall be applied on *zpar* but never on *parent*.

The algorithm given in Fig. 3 is the union-find-based max-tree algorithm as proposed by [12]. It starts with sorting pixels that can be done with a counting sort algorithm for low-quantized data or with a radix sort-based algorithm for high quantized data [25].Then it annotates all pixels as *unprocessed* with -1 (in common implementations, pixels are positive offsets in a pixel buffer). Later in the algorithm, when a pixel p is processed it becomes the root of the component

```
function FIND-ROOT(par, p)
    if par(p) ≠ p then par(p) ← FIND-ROOT(par, par(p))
    return par(p)

function MAXTREE(f)
    for all p do parent(p) ← −1
    S ← sort pixels increasing
    for all p ∈ S backward do
        parent(p) ← p; zpar(p) ← p                          ▷ make-set
        for all n ∈ N_p such that parent(n) ≠ −1 do
            r ← FIND-ROOT(zpar, n)
            if r ≠ p then
                zpar(r) ← p                                 ▷ merge-set
                parent(r) ← p
    CANONICALIZE(parent, S)
    return (parent, S)

procedure CANONICALIZE(f, parent, S)
    for all p in S forward do
        q ← parent(p)
        if f(q) = f(parent(q)) then parent(p) ← parent(q)
```

Fig. 3. Union-find without union-by-rank.

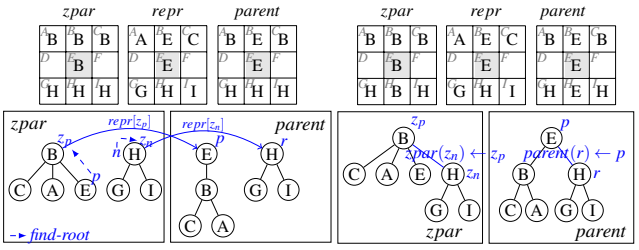
i.e $parent(p) = p$ with $p \neq -1$, thus testing $parent(p) \neq -1$ stands for *is p already processed*. Since S is processed in reverse order and `merge-set` sets the root of the tree to the current pixel p ($parent(r) \leftarrow p$), it ensures that the parent p will be seen before its child r when traversing S in the direct order.

```
procedure MAXTREE(f)
    for all p do parent(p) ← −1
    S ← sort pixels increasing
    for all p ∈ S backward do
        parent(p) ← p; zpar(p) ← p                          ▷ make-set
        rank(p) ← 0; repr(p) ← p
        z_p ← p
        for all n ∈ N_p s.t. parent(n) ≠ −1 do
            z_n ← FIND-ROOT(zpar, n)
            if z_n ≠ z_p then
                parent(repr(z_n)) ← p
                if rank(z_p) < rank(z_n) then swap(z_p, z_n)
                zpar(z_n) ← z_p                             ▷ merge-set
                repr(z_p) ← p
                if rank(z_p) = rank(z_n) then rank(z_p) ← rank(z_p)+1
    CANONICALIZE(parent, S)
    return (parent, S)
```

Fig. 4. Union-find with union-by-rank

2) Union-by-rank: The algorithm given in Fig. 4 is similar to the one in Fig. 3 but augmented with union-by-rank. It first introduces a new image $rank$. The `make-set` step creates a tree with a single node, thus with a rank set to 0. The $rank$ image is then used when merging two connected sets in $zpar$. Let z_p be the root of the connected component of p, and z_n be the root of connected component of $n \in \mathcal{N}(p)$. When merging two components, we have to decide which of z_p or z_n becomes the new root w.r.t their rank. If $rank(z_p) < rank(z_n)$, z_p becomes the root, z_n otherwise. If both z_p and z_n have the same rank then we can choose either z_p or z_n as the new root, but the rank should be incremented by one. On the other

hand, the relation $parent$ is unaffected by the union-by-rank, p becomes the new root whatever the rank of z_p and z_n. Whereas without balancing the root of any point p in $zpar$ matches the root of p in parent, this is not the case anymore. For every connected components we have to keep a connection between the root of the component in $zpar$ and the root of max-tree in $parent$. Thus, we introduce an new image $repr$ that keeps this connection updated.



(a) (b)

Fig. 5. Union-by-rank. (a) State of the algorithm before processing the neighbor H from E. (b) State of the algorithm after processing.

The union-by-rank technique and structure update are illustrated in Fig. 5. The algorithm has been running until processing E at level 12, the first neighbor B has already been treated and neighbors D and F are skipped because not yet processed. Thus, the algorithm is going to process the last neighbor H. z_p is the root of p in $zpar$ and we retrieve the root z_n of n with `find-root` procedure. Using $repr$ mapping, we look up the corresponding point r of z_n in $parent$. The tree rooted in r is then merged to the tree rooted in p ($parent$ merge). Back in $zpar$, the components rooted in z_p and z_n merge. Since they have the same rank, we choose arbitrary z_p to be the new root.

The algorithm in Fig. 4 is slightly different from the one of [15]. They use two union-find structures, one to build the tree, the other to handle flat zones. In their paper, `lowernode[`z_p`]` is an array that maps the root of a component z_p in *zpar* to a point of current level component in *parent* (just like `repr (`z_p`)` in our algorithm). Thus, they apply a second union-find to retrieve the canonical element. This extra union-find can be avoided because `lowernode[x]` is already a canonical element, thus $findoot$ on $lowernode(z_p)$ is useless and so does $parent$ balancing on flat zones.

3) Canonicalization: Both algorithms call the CANONICALIZE procedure to ensure that any node's parent is a canonical node. In Fig. 3, canonical property is propagated downward. S is traversed in direct order such that when processing a pixel p, its parent q has the canonical property that is $parent(q)$ is a canonical element. Hence, if q and $parent(q)$ belongs to the same node i.e $f(q) = f(parent(q))$, the parent of p is set to the component's canonical element: $parent(q)$.

4) Level compression: Union-by-rank provides time complexity guaranties at the price of an extra memory requirement. When dealing with huge images, it results in a significant drawback (e.g. RAM overflow...). Since the last point processed always becomes the root, union-find without rank

```
function MAXTREE(f)
    for all p do parent(p) ← −1
    S ← sort pixels increasing
    j = N − 1
    for all p ∈ S backward do
        parent(p) ← p; zpar(p) ← p                    ▷ make-set
        z_p = p
        for all n ∈ N_p s.t. parent(n) ≠ −1 do
            z_n ← FIND-ROOT(zpar, n)
            if z_p ≠ z_n then
                if f(z_p) = f(z_n) then SWAP(z_p, z_n)
                zpar(z_n) ← z_p                       ▷ merge-set
                parent(z_n) ← z_p
                S[j] ← z_n; j ← j − 1
    S[0] ← parent[S[0]]
    CANONICALIZE(parent, S)
    return (parent, S)
```

Fig. 6. Union-find with level compression.

technique tends to create a degenerated tree in flat zones. Level compression avoids this behavior by a special handling of flat zones. In Fig. 6, p is the point in process at level $\lambda = f(p)$, n a neighbor of p already processed, z_p the root of P_p^λ (at first $z_p = p$), z_n the root of P_n^λ. We suppose $f(z_p) = f(z_n)$, thus z_p and z_n belong to the same node and we can choose any of them as a canonical element. Normally p should become the root with child z_n but level compression inverts the relation, z_n is kept as the root and z_p becomes a child. Since $parent$ may be inverted, S array is not valid anymore. Hence S is reconstructed, as soon as a point p gets attached to a root node, p will not be processed anymore so it is inserted in back of S. At the end S only misses the tree root which is $parent[S[0]]$.

B. Flooding algorithms

A second class of algorithms, based on flooding, contrasts with the immersion-based algorithms described in the previous section III-A. Salembier et al. [3] proposed the first efficient algorithm to compute the max-tree. A propagation starts from the root that is the pixel at lowest level l_{min}. Pixels in the propagation front are stored in a hierarchical queue composed by as many FIFO queues as the number of levels. It allows to access directly any pixel in the FIFO queue at a given level. Fig. 7 shows a slightly modified version of Salembier's original algorithm where the original *STATUS* image is replaced by the $parent$ image having the same role. The `flood(`λ`, r)` procedure is in charge of flooding the peak component P_r^λ and building the corresponding sub max-tree rooted in r. It proceeds as follows: first pixels at level λ are retrieved from the queue, their $parent$ pointer is set to the canonical element r and their neighbors n are analyzed. If n is not in the queue and has not yet been processed, then n is pushed in the queue for further processing and n is marked as processed ($parent(n)$ is set to `INQUEUE` which is any value different from -1). If the level l of n is higher than λ then n is in the childhood of the current node, thus flooding is halted at the current level and a recursive call to `flood` initiates the process for the peak component P_n^l rooted in n. During the recursive flooding, some points can be pushed in the queue between level λ and l. Hence, when `flood` ends, it returns the level l' of n's parent.

```
function FLOOD(λ, r)
    while hqueue[λ] not empty do
        p ← POP(hqueue[λ])
        parent(p) ← r
        if p ≠ r then INSERT_FRONT(S, p)
        for all n ∈ N(p) s.t. parent(p) = −1 do
            l ← f(n)
            if levroot[l] = −1 then levroot[l] ← n
            PUSH(hqueue[l], n)
            parent(n) ← INQUEUE
            while l > λ do
                l ← flood(l, levroot[l])
                                                  ▷ Attach to parent
    levroot[λ] ← −1
    lpar ← λ − 1
    while lpar ≥ 0 and levroot[lpar] = −1 do
        lpar ← lpar − 1
    if lpar ≠ −1 then
        parent(r) ← levroot[lpar]
    INSERT_FRONT(S, r)
    return lpar

function MAX-TREE(f)
    for all h do levroot[h] ← −1
    for all p do parent(p) ← −1
    l_min ← min_p f(p)
    p_min ← arg min_p f(p)
    PUSH(hqueue[l_min], p_min)
    levroot[lmin] ← p_min
    FLOOD(l_min, p_min)
```

Fig. 7. Salembier et al. [3] max-tree algorithm.

```
1: procedure PROCESSSTACK(r, q)
2:     λ ← f(q)
3:     POP(levroot)
4:     while levroot not empty and λ < f(TOP(levroot)) do
5:         INSERT_FRONT(S, r)
6:         r ← parent(r) ← POP(levroot))
7:     if levroot empty or f(TOP(levroot)) ≠ λ then PUSH(levroot, q)
8:     parent(r) ← TOP(levroot)
9:     INSERT_FRONT(S, r)

10: function MAX-TREE(f)
11:     for all p do parent(p) ← −1
12:     p_start ← any point in Ω
13:     PUSH(pqueue, p_start); PUSH(levroot, p_start)
14:     parent(p_start) ← INQUEUE
15:     loop
16:         p ← TOP(pqueue); r ← TOP(levroot)
17:         for all n ∈ N(p) such that parent(p) = −1 do
18:             PUSH(pqueue, n)
19:             parent(n) ← INQUEUE
20:             if f(p) < f(n) then
21:                 PUSH(levroot, n)
22:                 goto 16
23:         { p is done }
24:         POP(pqueue)
25:         parent(p) ← r
26:         if p ≠ r then INSERT_FRONT(S, p)
27:     while pqueue not empty do;
28:         { all points at current level done ? }
29:         q ← TOP(pqueue)
30:         if f(q) ≠ f(r) then                  ▷ Attach r to its parent
31:             PROCESSSTACK(r, q)
32:     repeat
33:     root ← POP(levroot)
34:     INSERT_FRONT(S, root)
```

Fig. 8. Non-recursive max-tree algorithm [26, 20].

If $l' > \lambda$, we need to flood levels l' until $l' \leq \lambda$ i.e. until there are no more points in the queue above λ. Once all pixels at level λ have been processed, we need to retrieve the level $lpar$ of the parent component and attach r to its canonical element. A $levroot$ array stores canonical element of each level component and -1 if the component is empty. Thus we just have to traverse $levroot$ looking for $lpar = \max\{h < \lambda, levroot[h] \neq -1\}$ and set the parent of r to $levroot[lpar]$. Since the construction of $parent$ is bottom-up, we can safely insert p in front of the S array each time $parent(p)$ is set. For a level component, the canonical element is the last element inserted ensuring a correct ordering of S. Note that the pass which gets the minimum level of the image is not necessary. Instead, we could have called `flood` in `Max-tree` procedure until the parent level returned by the function was -1, i.e the last flood call was processing the root. Anyway, this pass has other advantages for optimization that will be discussed in the implementation details section.

Salembier et al. [3]'s algorithm was rewritten in a non-recursive implementation in [27] and later by [26] and [20]. These algorithms differ in only two points. First, [20] uses a pass to retrieve the root before flooding to mimics the original recursive version while [26] does not. Second, priority queues in [26] use an unacknowledged implementation of heap based on hierarchical queues while in [20] they are implemented using a standard heap (based on comparisons). The algorithm given in Fig. 8 is a code transcription of the method described in [26]. The array $levroot$ in the recursive version is replaced by a stack with the same purpose: storing the canonical element of level components. The hierarchical queue $hqueue$ is replaced by a priority queue $pqueue$ that stores the propagation front. The algorithm starts with some initialization and choose a random point p_{start} as the flooding point. p_{start} is enqueued and pushed on $levroot$ as a canonical element. During the flooding, the algorithm picks the point p at highest level (with the highest priority) in the queue, and the canonical element r of its component which is the top of $levroot$ (p is not removed from the queue). Like in the recursive version, we look for neighbors n of p and enqueue those that have not yet been seen. If $f(n) > f(p)$, n is pushed on the stack and we immediately flood n (a *goto* that mimics the recursive call). On the other hand, if all neighbors are in the queue or already processed then p is *done*, it is removed from the queue, $parent(p)$ is set its the canonical element r and if $r \neq p$, p is added to S (we have to ensure that the canonical element will be inserted last). Once p removed from the queue, we have to check if the level component has been fully processed in order to attach the canonical element r to its

parent. If the next pixel q has a different level than p, we call the procedure `ProcessStack` that pops the stack, sets parent relationship between canonical elements and inserts them in S until the top component has a level no greater than $f(q)$. If the stack top's level matches q's level, q extends the component so that no more processing is needed. On the other hand, if the stack gets empty or the top level is lesser than $f(q)$, then q is pushed on the stack as the canonical element of a new component. The algorithm ends when all points in queue have been processed, then S only misses the root of the tree which is the single element that remains on the stack.

C. Merge-based algorithms and parallelism

Whereas the algorithms of the two first classes (Sections III-A and III-B) are sequential, this section is dedicated to parallel algorithms. Merge-based algorithms consist in computing max-trees on sub-parts of images and merging back trees to get the max-tree of the whole image [21, 17, 22]. Those algorithms are typically well-suited for parallelism since they adopt a map-reduce idiom. Computation of sub max-trees (map step), done by any sequential method and merge (reduce-step) are executed in parallel by several threads. In order to improve cache coherence, images should be split in contiguous memory blocks that is, splitting along the first dimension if images are row-major. Fig. 9 shows an example of parallel processing using a map-reduce idiom. The domain has been split into five sub-domains $\{D_1, D_2, \ldots, D_5\}$, we thus have 5 *map* operations which run a sequential algorithm and 4 *joins* that merge the sub-trees. Figs. 9b and 9c show a possible distribution of the tasks on 3 threads. Note that *map*-steps and *reduce*-steps may occur in parallel, but a single thread may also be in charge of several sub-tree construction. For instance, the first thread is in charge of computing the sub-trees T_1 and T_2 for D_1 and D_2, merging them into a tree T_{12} and then merging it with the tree computed by the second thread. Choosing the right number of splits and jobs distribution between threads is a difficult topic that depends on the architecture (number of threads available, power frequency of each core). If the domain is not split enough (a number of chunks no greater than the number of threads) the parallelism is not maximal, some threads become idle once they have done their jobs, or wait for other thread to merge. On the other hand, if the number of split gets too large, merging and thread synchronization cause significant overheads. Since work balancing and thread management are outside the current topic, they are delegated to high level parallelism libraries such as Intel Threading Building Blocks (TBB).

The procedure in charge of merging sub-trees T_i and T_j of two adjacent domains D_i and D_j is given in Fig. 10. For two neighbors p and q in the junction of D_i, D_j, it connects components of p's branch in T_i to components of q's branch in T_j until a common ancestor is found. Let x and y be the canonical elements of the components to merge with $f(x) \geq f(y)$ (x is in the childhood to y) and z be the canonical element of the parent component of x. If x is the root of the sub-tree then it gets attached to y and the procedure ends. Otherwise, we traverse up the branch of x to find the component that will

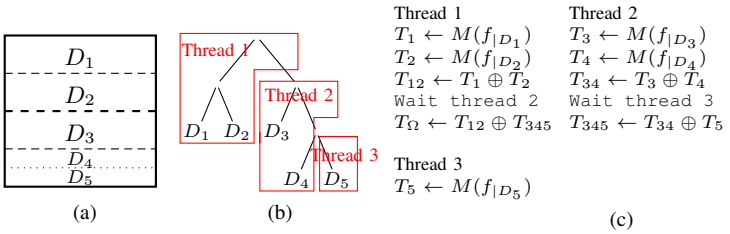


Fig. 9. Map-reduce idiom for max-tree computation. (a) Sub-domains of f. (b) A possible distribution of jobs by threads. (c) Map-reduce operations. M is the map operator, $\oplus$ the merge operator.

```
procedure CONNECT(p,q)
    x ← FINDREPR(parent, p)
    y ← FINDREPR(parent, q)
    if f(x) < f(y) then SWAP(x, y)
    while x ≠ y do                          ▷ common ancestor found ?
        parent(x) ← FINDREPR(parent, parent(x));
        z ← parent(x)
        if x = z then parent(x) ← y ; y ← x
        else if f(z) ≥ f(y) then x ← z
        else parent(x) ← y ; x ← y ; y ← z

function FINDREPR(par, p)
    if f(p) ≠ f(par(p)) then return p
    par(p) ← FINDREPR(par, par(p))
    return par(p)

procedure MERGETREE(D_i, D_j)
    for all (p, q) ∈ D_i × D_j such that q ∈ N(p) do
        CONNECT(p,q)
```

Fig. 10. Tree merge algorithm.

be attached to y that is the lowest node having a level greater than $f(y)$. Once found, x gets attached to y, and we now have to connect y to x's old parent. Function `findrepr(p)` is used to get the canonical element of p's component whenever the algorithm needs it.

Once sub-trees have been computed and merged into a single tree, it does not hold canonical property (because non-canonical elements are not updated during merge). Also, the reduction step does not merge the S arrays corresponding to sub-trees (it would imply reordering S which is more costly than just recomputing it at the end). Fig. 11 shows an algorithm that canonicalizes and reconstructs S array from $parent$ image. It uses an auxiliary image $dejavu$ to track nodes that have already been inserted in S. As opposed to other max-tree algorithms, construction of S and processing of nodes are top-down. For any points p, we traverse in a recursive way its path to the root to process its ancestors. When the recursive call returns, $parent(p)$ is already inserted in S and holds the canonical property, thus we can safely insert back p in S and canonicalize p as in Fig. 3.

D. Implementation details

Algorithms have been implemented in pure C++ using STL implementation of some basic data structures (heaps, priority queues), MILENA image processing library to provide fundamental image types and I/O functionality, and Intel TBB [28]

```
procedure CANONICALIZEREC(p)
    dejavu(p) = true
    q ← parent(p)
    if not dejavu(q) then                          ▷ Process parent before p
        CANONICALIZEREC(q)
    if f(q) = f(parent(q)) then                    ▷ Canonicalize
        parent(p) ← parent(q)
    INSERTBACK(S, p)

procedure POST-PROCESS(parent, f)
    for all p do dejavu(p) ← False
    for all p ∈ Ω such that not dejavu(p) do
        CANONICALIZEREC(p)
    return (parent, S)
```

Fig. 11. Canonicalization and S computation algorithm.

for parallelism. Specific implementation optimizations are listed below:

Sort optimization. A counting sort is used when quantization is lower than 18 bits. For large integer of q bits, it switches to 2^{16}-based radix sort requiring $q/16$ counting sorts.

Pre-allocation. Queues and stacks are pre-allocated to avoid dynamic memory reallocation. Hierarchical queues are also pre-allocated by computing image histogram as a pre-processing.

Priority-queues. An heap is implemented with hierarchical queues when quantization is less than 18 bits. For large integer it switches to the STL standard heap implementation. A "y-fast trie" data structure [29] can be used for large integer ensuring a better complexity (see Section IV-A) but no performance gain has been obtained.

Map-reduce. In the parallel version of the algorithms, all instructions that deal about S construction and canonicalization have been removed since S is reconstructed from scratch and $parent$ canonicalized by the procedure in Fig. 11

IV. ALGORITHMS COMPARISON

A. Complexity analysis

Let $n = H * W$ with H the image height, W the image width and n the total number of pixels. Let k be the number of values in V.

1) Immersion algorithms: They require sorting pixels, a process of $\Theta(n + k)$ complexity ($k \ll n$) for small integers (counting sort), $O(n \log\log n)$ for large integers (hybrid radix sort), and $O(n \log n)$ for generic data types with a more complicated ordering relation (comparison sort). Union-find is $O(n \log n)$ and $O(n\alpha(n))$ when used with union-by-rank [1]. Canonicalization is linear and does not use extra memory. Memory-wise, sorting may require an auxiliary buffer depending on the algorithm and histograms for integer sorts thus $\Theta(n + k)$ extra-space. Union without rank requires a $zpar$ image for path compression ($\Theta(n)$) and the system stack for recursive calls in `findroot` which is $O(n)$ (`findroot` could be non-recursive, but memory space is saved at cost of a higher computational time). Union-by-rank requires two extra images ($rank$ and $repr$) of n pixels each.

2) Flooding algorithms: They require a priority queue to retrieve the highest point in the propagation front. Each point is inserted and removed once, thus the complexity is $\Theta(np)$ where p is the cost of pushing or popping from the heap. If the priority queue is encoded with a hierarchical queue as in [3, 26], it uses $n + 2k$ memory space, provides constant insertion and constant access to the maximum but popping is $O(k)$. In practice, in images with small integers, gray level difference between neighboring pixels is far to be as large as k. With high dynamic image, a heap can be implemented with a y-fast trie [29], which has insertion and deletion in $O(\log\log k)$ and access to maximum element in $O(1)$. For any other data type, a "standard" heap based on comparisons requires n extra space, allows insertion and deletion in $O(\log n)$ and has a constant access to its maximal element. Those algorithms need an array or a stack of respective size k and n. Salembier's algorithm uses the system stack for a recursion of maximum depth k, hence $O(k)$ extra-space.

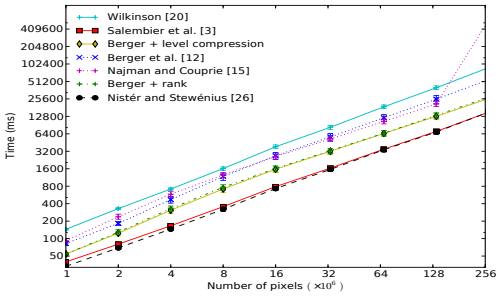
3) Merge-based algorithms: The complexity depends on $\mathcal{A}(k, n)$, the complexity of the underlying method used to compute the max-trees of sub-domains. Let $s = 2^h$ the number of sub-domains. The map-reduce algorithms require s mapping operations and $s - 1$ merges. A good map-reduce algorithm would split the domain to form a full and complete tree so we assume all leaves to be at level h. Merging sub-trees of size $n/2$ has been analyzed in [17] and is $O(k \log n)$ (we merge nodes of every k levels using union-find without union-by-rank). Thus, the complexity of a single reduction is $O(Wk \log n)$. Assuming s constant and $H = W = \sqrt{n}$ the complexity as a function of n and k of the map-reduce algorithm is $O(\mathcal{A}(k, n)) + O(k\sqrt{n} \log n)$. When there is as many splits as rows, s is now dependent on n. This leads to Matas et al. [22] algorithm whose complexity is $O(n) + O(k\sqrt{n}(\log n)^2)$. Contrary to what they claim, when values are small integers the complexity stays linear and is not dominated by merging operations. Finally, canonicalization and S reconstruction have a linear time complexity (`CanonicalizeRec` is called only once for each point) and only use an image of n elements to track points already processed. The complexity analysis for each algorithm as well as the memory required by any auxiliary data structure (including preallocated stacks and queues) is summerized in Table I.

B. Experiments

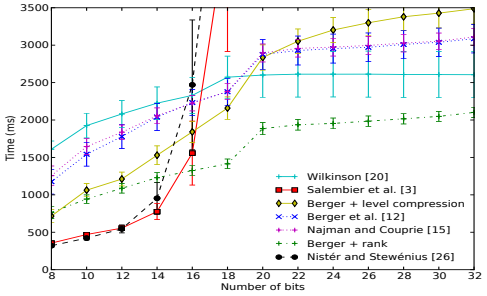
Benchmarks were performed on an Intel Core i7 (4 physical cores, 8 logical cores). The programs were compiled with gcc 4.7, optimization flags on (`-O3 -march=native`). Tests were conducted on a dataset of 8-bit images that were re-sized by cropping or tiling the original image. Over-quantization was performed by shifting the eight bits left and generating missing lower bits at random. Fig. 12 depicts performance of the sequential algorithms w.r.t to the size and the quantization. As a first remark, we notice that all algorithms are linear in practice. On natural images, the $n \log n$ upper bound complexity of the [20] and [12] algorithms is not reached. Algorithms from [12] and [15] have quite the same running time ($\pm 6\%$ on average), however the performance of [15] algorithm drops

TABLE I. COMPLEXITY AND SPACE REQUIREMENTS OF MANY MAX-TREE ALGORITHMS. n IS THE NUMBER OF PIXELS AND k THE NUMBER OF GRAY LEVELS.

	Time complexity			Auxiliary space requirement		
Algorithm	Small int	Large int	Generic V	Small int	Large int	Generic V
Berger [12]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$n + k + O(n)$	$2n + O(n)$	$n + O(n)$
Berger + rank	$O(n\,\alpha(n))$	$O(n \log \log n)$	$O(n \log n)$	$3n + k + O(n)$	$4n + O(n)$	$3n + O(n)$
Najman and Couprie [15]	$O(n\,\alpha(n))$	$O(n \log \log n)$	$O(n \log n)$	$5n + k + O(n)$	$6n + O(n)$	$5n + O(n)$
Salembier et al. [3]	$O(nk)$	$O(nk) \simeq O(n^2)$	N/A	$3k + n + O(n)$	$2k + n + O(n)$	N/A
Nistér and Stewénius [26]	$O(nk)$	$O(nk) \simeq O(n^2)$	N/A	$2k + 2n$	$2k + 2n$	N/A
Wilkinson [20]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$3n$	$3n$	$3n$
Salembier non-recursive	$O(nk)$	$O(n \log \log n)$	$O(n \log n)$	$2k + 2n$	$3n$	$3n$
Map-reduce	$O(A(k,n))$	$O(A(k,n)) + O(k\sqrt{n} \log n)$		$\ldots + n$	$\ldots + n$	$\ldots + n$
Matas et al. [22]	$O(n)$	$O(n) + O(k\sqrt{n}(\log n)^2)$		$k + n$	$2n$	$2n$

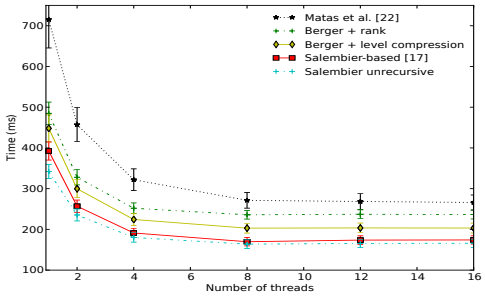


(a)

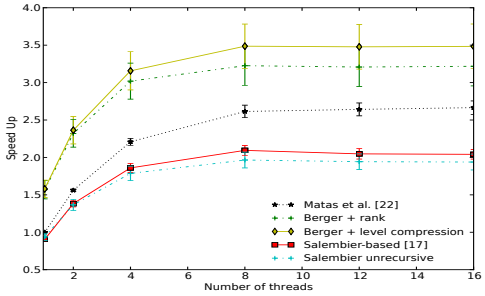


(b)

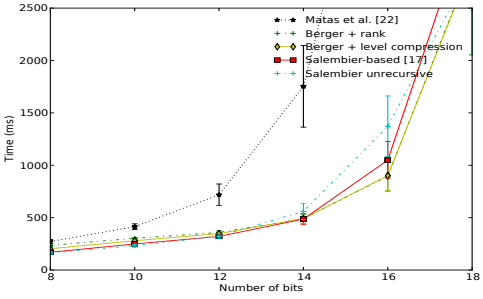
Fig. 12. (a) Comparison of the algorithms on 8-bit images as a function of the size; (b) Comparison of the algorithms on 8 Mega-pixels images as a function of the quantization.



(a)



(b)



(c)

Fig. 13. (a,b) Comparison of the parallel algorithms on a 6.8 Mega-pixels 8-bits image as a function of number of threads. (a) Wall clock time; (b) speedup w.r.t the sequential version; (c) Comparison of the parallel algorithms using 8 threads on a 6.8 Mega-pixels image as a function of the quantization.

significantly at 256 Mega-pixels. Indeed, at that size each auxiliary array/image requires 1 GB memory space, thus [15] who use a lot of memory exceed the 6 GB RAM limit and need to swap. Our implementation of union-by-rank uses less memory and is on average 42% faster than [15]. Level compression is an efficient optimization that provides 35% speedup on average on [12]. However, this optimization is only reliable on low quantized data. Fig. 12b shows that it is relevant up to 18 bits. It operates on flat-zones but when quantization gets higher, flat-zones are less probable and the tests add worthless overheads (see Fig. 14). Union-find is not affected by the quantization but sorting does, counting sort and radix sort complexities are respectively linear and logarithmic with the number of bits. The break in union-find curves between 18 and 20 bits stands for the switch from counting to radix sort. Flooding-based algorithms using hierarchical queues outperform our union-find by rank on low quantized image by 41% on average. As expected, [3] and [26] (which is the exact non-recursive version of the former) closely match. However, the exponential cost of hierarchical queues w.r.t the number of bits is evident on Fig. 12b. By using a standard heap instead of hierarchical queues, [20] does scale well with the number of bits and outperforms every algorithms except our implementation of union-by-rank. In [20], the algorithm is supposed to match [3]'s method for low quantized images, but in our experiments it remains 4 times slower. Since [15]'s algorithm is always

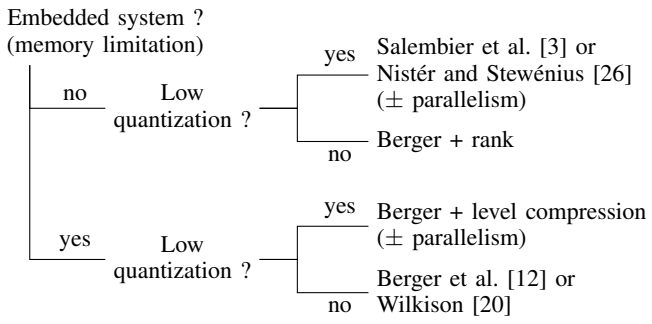


Fig. 16. Decision tree to choose the appropriate max-tree algorithm.

outperformed by our implementation of union-find by rank, it will not be tested any further. Furthermore, because of the strong similarities of [26] and [20], they are merged in our single implementation (called *Non-recursive Salembier* below) that will use hierarchical queues when quantization is below 18 bits and switches to a standard heap implementation otherwise. Finally, the algorithm *Berger + level compression* will enable level compression only when the number of bits is below 18.

Fig. 13 shows the results of the map-reduce idiom applied on many algorithms and their parallel versions. As a first result, we can see that better performance is generally achieved with 8 threads that is when the number of threads matches the number of (logical) cores. However, since there are actually only 4 physical cores, we can expect a $\times 4$ maximum speedup. Some algorithms benefit more from map-reduce than others. Union-find-based algorithms are particularly well-suited for parallelism. Union-find with level compression achieves the best speedup, 3.6 times faster that the sequential version while the union-find by rank, second, performs a $\times 3.1$ speedup. More surprising, the map-reduce pattern achieves significant speedup even when a single thread is used ($\times 1.7$ and $\times 1.4$ for union-find with level compression and union-find by rank respectively). This result is explained by a better cache coherence when working on sub-domains that balances tree merges overheads. On the other hand, flooding algorithms do not scale that well because they are limited by post-processes. Indeed, Fig. 15 shows that 76% of the time of parallelized Salembier's algorithm is spent in post-preprocessing (that is going to happen as well for union-find algorithms on architectures with more cores). In [17] and [22], they obtain a speedup almost linear with the number of threads because only a $parent$ image is built. If we remove the canonicalization and the S construction steps, we also get those speedups. Fig. 13c shows the exponential complexity of merging trees as number of bits increases that makes parallel algorithms unsuitable for high quantized data. In light of the previous analysis, Fig. 16 provides some guidelines on how to choose the appropriate max-tree algorithm *w.r.t.* to image types and architectures.

V. Conclusion

In this paper, we tried to lead a fair comparison of max-tree algorithms in a unique framework. We highlighted the fact that there is no such thing as the "best" algorithm that outranks all the others in every case and we provided a decision tree to choose the appropriate algorithm *w.r.t.* to data and hardware. We proposed a max-tree algorithm using union-by-rank that outperforms the existing one from [15]. Furthermore, we proposed a second one that uses a new technique, *level compression*, for systems with strict memory constraints. Extra-materials including the image dataset used for this comparison, and a "reproducible research" code, intensively tested, is available on the Internet at http://www.lrde.epita.fr/Olena/MaxtreeReview.

Actually the union-find algorithm is a versatile tool used in many algorithms. A recent publication [30] shows that the morphological *tree of shapes*, which is a self-dual representation of the image contents, can also be computed using union-find. In [31], a specific binary tree, corresponding to an ordered version of the edges of the minimum spanning tree, is computed thanks to a Kruskal-like algorithm and involves the union-find algorithm. Thus, the results presented in this paper also apply to obtain those trees in the most efficient way.

References

[1] L. Vincent, "Grayscale area openings and closings, their efficient implementation and applications," in *Proc. of EURASIP Workshop on Mathematical Morphology and its Applications to Signal Processing*, 1993, pp. 22–27.

[2] P. Salembier and J. Serra, "Flat zones filtering, connected operators, and filters by reconstruction," *IEEE Trans. on Image Processing*, vol. 4, no. 8, pp. 1153–1160, 1995.

[3] P. Salembier, A. Oliveras, and L. Garrido, "Antiextensive connected operators for image and sequence processing," *IEEE Transactions on Image Processing*, vol. 7, no. 4, pp. 555–570, 1998.

[4] R. Jones, "Connected filtering and segmentation using component trees," *Computer Vision and Image Understanding*, vol. 75, no. 3, pp. 215–228, 1999.

[5] J. Hernández and B. Marcotegui, "Shape ultimate attribute opening," *Image and Vision Computing*, vol. 29, no. 8, pp. 533–545, 2011.

[6] E.R. Urbach and M.H.F. Wilkinson, "Shape-only granulometries and grey-scale shape filters," in *Mathematical Morphology and Its Applications to Signal and Image Processing (Proceedings of ISMM)*, 2002, pp. 305–314.

[7] G.K. Ouzounis and M.H.F. Wilkinson, "Mask-based second-generation connectivity and attribute filters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 6, pp. 990–1004, 2007.

[8] P. Monasse and F. Guichard, "Fast computation of a contrast-invariant image representation," *IEEE Trans. on Image Processing*, vol. 9, no. 5, pp. 860–872, 2000.

[9] Y. Xu, T. Géraud, and L. Najman, "Morphological filtering in shape spaces: Applications using tree-based image representations," in *Proc. of International Conference on Pattern Recognition*, 2012, pp. 485–488.

[10] J. Matas, O. Chum, M. Urban, and T. Pajdla, "Robust wide-baseline stereo from maximally stable extremal regions," *Image and Vision Computing*, vol. 22, no. 10, pp. 761–767, 2004.

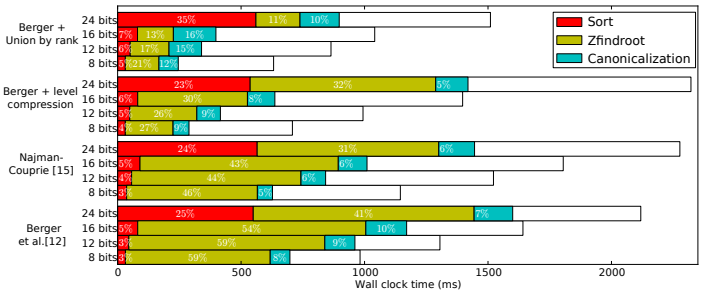


Fig. 14. Time distribution of the sequential union-find-based algorithms.

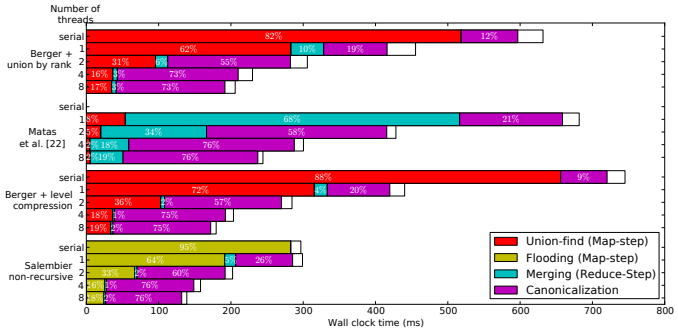


Fig. 15. Time distribution of the parallel versions of the algorithms.

[11] 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 Transactions on Image Processing*, vol. 16, no. 12, pp. 2943–2952, 2007.

[12] C. Berger, T. Géraud, R. Levillain, N. Widynski, A. Baillard, and E. Bertin, "Effective component tree computation with application to pattern recognition in astronomical imaging," in *Proc. of IEEE International Conference on Image Processing*, vol. 4, 2007, pp. IV–41.

[13] B. Perret, S. Lefèvre, C. Collet, and É. Slezak, "Connected component trees for multivariate image processing and applications in astronomy," in *Proc. of Intl. Conference on Pattern Recognition*, 2010, pp. 4089–4092.

[14] E. Carlinet and T. Géraud, "A comparison of many max-tree computation algorithms," in *Mathematical Morphology and Its Applications to Signal and Image Processing (Proceedings of ISMM)*, ser. Lecture Notes on Computer Science, vol. 7883. Springer, 2013, pp. 73–85.

[15] L. Najman and M. Couprie, "Building the component tree in quasi-linear time," *IEEE Transactions on Image Processing*, vol. 15, no. 11, pp. 3531–3539, 2006.

[16] M.H.F. Wilkinson and M.A. Westenberg, "Shape preserving filament enhancement filtering," in *Proc. of Medical Image Computing and Computer-Assisted Intervention*, 2001, pp. 770–777.

[17] M.H.F. Wilkinson, H. Gao, W.H. Hesselink, J.E. Jonker, and A. Meijster, "Concurrent computation of attribute filters on shared memory parallel machines," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 10, pp. 1800–1813, 2008.

[18] A.V. Aho, J.D. Ullman, and J.E. Hopcroft, *Data Structures and Algorithms*. Addison Wesley, 1983.

[19] R.E. Tarjan, "Efficiency of a good but not linear set union algorithm," *Journal of the ACM*, vol. 22, no. 2, pp. 215–225, 1975.

[20] M.H.F. Wilkinson, "A fast component-tree algorithm for high dynamic-range images and second generation connectivity," in *Proc. of IEEE International Conference of Image Processing*, 2011, pp. 1021–1024.

[21] G.K. Ouzounis and M.H.F. Wilkinson, "A parallel implementation of the dual-input max-tree algorithm for attribute filtering," in *Mathematical Morphology and Its Applications to Signal and Image Processing (Proceedings of ISMM)*, vol. 1. "Instituto Nacional de Pesquisas Espaciais (INPE)", 2007, pp. 449–460.

[22] P. Matas, E. Dokladalova, M. Akil, T. Grandpierre, L. Najman, M. Poupa, and V. Georgiev, "Parallel algorithm for concurrent computation of connected component tree," in *Advanced Concepts for Intelligent Vision Systems*, 2008, pp. 230–241.

[23] D. Menotti, L. Najman, and A. de Albuquerque Araújo, "1D component tree in linear time and space and its application to gray-level image multithresholding," in *Mathematical Morphology and Its Apps. to Signal and Image Processing (Proc. of ISMM)*, 2007, pp. 437–448.

[24] V. Morard, P. Dokládal, and E. Decenciere, "One-dimensional openings, granulometries and component

trees in $\mathcal{O}(1)$ per pixel," *IEEE Journal of Selected Topics in Signal Processing*, vol. 6, pp. 840–848, 2012.
[25] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman, "Sorting in linear time?" in *Proc. of the Annual ACM symposium on Theory of computing*, 1995, pp. 427–436.
[26] D. Nistér and H. Stewénius, "Linear time maximally stable extremal regions," in *Proc. of European Conference on Computer Vision*, 2008, pp. 183–196.
[27] W.H. Hesselink, "Salembier's min-tree algorithm turned into breadth first search," *Information Processing Letters*, vol. 88, no. 5, pp. 225–229, 2003.
[28] J. Reinders, *Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, 2007.
[29] D. E. Willard, "Log-logarithmic worst-case range queries are possible in space $\theta(N)$," *Information Processing Letters*, vol. 17, no. 2, pp. 81–84, 1983.
[30] T. Géraud, E. Carlinet, S. Crozet, L. Najman *et al.*, "A quasi-linear algorithm to compute the tree of shapes of nD images," in *Mathematical Morphology and Its Applications to Signal and Image Processing (Proceedings of ISMM)*, ser. Lecture Notes on Computer Science, vol. 7883. Springer, 2013, pp. 98–110.
[31] L. Najman, J. Cousty, and B. Perret, "Playing with Kruskal: Algorithms for morphological trees in edge-weighted graphs," in *Mathematical Morphology and Its Applications to Signal and Image Processing (Proceedings of ISMM)*, ser. Lecture Notes on Computer Science, vol. 7883. Springer, 2013, pp. 135–146.

Edwin Carlinet received the Ing. degree from EPITA, Paris, France, in 2011, and a M.Sc. in applied mathematics for computer vision and machine learning from the École Normale Supérieure Cachan, in 2012. He is currently a Ph.D. candidate with EPITA and Université Paris-Est. His research interests include bio-informatics mathematical morphology, and statistical learning

Thierry Géraud received a Ph.D. degree in signal and image processing from Télécom ParisTech in 1997, and the Habilitation à Diriger les Recherches from Université Paris-Est in 2012. He is one of the main authors of the Olena platform, dedicated to image processing and available as free software under the GPL licence. His research interests include image processing, pattern recognition, software engineering, and object-oriented scientific computing. He is currently working at EPITA Research and Development Laboratory (LRDE), Paris, France.