HAL
open science

A Comparison of Many Max-tree Computation Algorithms

Edwin Carlinet, Thierry Géraud

To cite this version:

Edwin Carlinet, Thierry Géraud. A Comparison of Many Max-tree Computation Algorithms. 11th International Symposium on Mathematical Morphology (ISMM), May 2013, Uppsala, Sweden. pp.73 - 85, 10.1007/978-3-642-38294-9_7 . hal-01476238

HAL Id: hal-01476238

https://inria.hal.science/hal-01476238v1

Submitted on 24 Feb 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A comparison of many max-tree computation algorithms

Edwin Carlinet* and Thierry Géraud

EPITA Research and Development Laboratory (LRDE)
edwin.carlinet@lrde.epita.fr, thierry.geraud@lrde.epita.fr

Abstract. With the development of connected filters in the last decade, many algorithms have been proposed to compute the max-tree. Max-tree allows computation of the most advanced connected operators in a simple way. 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 five algorithms and some variations of them in a unique framework. Finally, a decision tree will be proposed to help the user choose the most appropriate algorithm according to their requirements.

1 Introduction

In mathematical morphology, connected filters are those that modify an original signal by only removing connected components, hence those that preserve image contours. Originally, they were mostly used for image filtering [19, 16]. Major advances came from max and min-tree as hierarchical representations of connected components and from an efficient algorithm able to compute them [15]. Since then, usage of these trees has soared for more advanced forms of filtering: based on attributes [4], using new filtering strategies [15, 18], allowing new types of connectivity [12]. They are also a base for other image representations. In [9] a tree of shapes is computed from a merge of min and max trees. In [24] a component tree is computed over the attributes values of the max-tree. Max-trees have been used in many applications: computer vision through motion extraction [15], features extraction with MSER [6], segmentation, 3D visualization [20]. With the increase of applications comes an increase of data type to process: 12-bit images in medical imagery [20], 16-bit or float images in astronomical imagery [1], 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 this paper, we provide a full and exhaustive comparison of state-of-the-art max-tree algorithms in a unique framework, i.e., same architecture, same language (C++) and same outputs. The paper is organized as

* Edwin Carlinet is now also with Université Paris-Est, Laboratoire d'Informatique Gaspard-Monge, Équipe A3SI, ESIEE Paris, France.

follows: Section 2 recalls basic notions and manipulations of max-tree, describes algorithms and implementations used in this study. Section 3 is dedicated to the comparison of those algorithms both in terms of complexity and running times through experimentations.

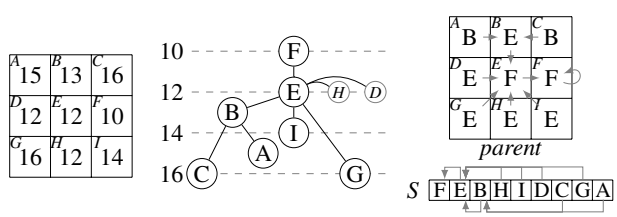
2 A tour of max-tree: definition, representation and algorithms

2.1 Basic notions for max-tree

Let $ima : \Omega \to V$ an image on regular domain Ω, having values on a totally preordered set $(V, <)$ and let $\mathcal{N}$ a neighborhood on Ω. Let $\lambda \in V$, we note $[ima \leq \lambda]$ the set $\{p \in \Omega, ima(p) \leq \lambda\}$. 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. $\{CC([ima = \lambda]), \lambda \in V\}$ are *level components* and $\{CC([ima \geq \lambda]), \lambda \in V\}$ (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 $X \in CC([ima \geq \lambda])$ such that $p \in X$.

2.2 Max-tree representation

Berger et al. [1], Najman and Couprie [10] rely on a simple and effective encoding of component-trees using an image that stores the *parent* relationship that exists between components. A connected component is represented by a single point called the *canonical element* [1, 10] or *level root*. Let two points $p, q \in \Omega$, and p_r the root of the tree. We say that p is canonical if $p = p_r$ or $ima(parent(p)) < ima(p)$. A *parent* image shall verify 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) $ima(parent(p)) \leq ima(p)$ and 3) $parent(p)$ is canonical.



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

Fig. 1: Representation of a max-tree with a parent image and an array.

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, whereas 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. Figure 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).

2.3 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 [22] or even mumford-shah like energy [24]. Some max-tree algorithms [21, 6] 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 [20]). 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. Algorithm 1 provides an implementation of attribute computation and direct-filtering with the representation. $\hat{f} : \Omega \times V \rightarrow \mathcal{A}$ is an application that projects a pixel p and its value $ima(p)$ in the attribute space $\mathcal{A}$. $\hat{+} : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$ 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 [15] that keeps all nodes passing a criterion λ and lowers nodes that fails to the last ancestor "alive". This implementation has to be compared with the one in [21] that only uses *parent*. This one is shorter, faster and clearer above all.

2.4 Three kinds of Max-tree algorithms

Max-tree algorithms can be classified in three classes.

Immersion algorithms. They start with building n disjoints singleton for each pixel and sort them according to their gray value. When immersion starts, highest levels are processed first such that local maxima create some bassins. While processing pixels in decreasing order of their gray value, bassins that form disjoint sets of pixels are extended and merges when the current pixel creates a connection between two of them. Finally, the pixels at lowest level are processed, all bassins have been merged and the whole image has been immersed. The way bassins grow and merge form a tree. Disjoint connected sets of pixels are handled with Tarjan's union-find algorithm where connected sets are encoded with trees. A single pixel, the root of the tree, represents the whole connected component. Tarjan [17] provides three basic manipulation routines: MAKE-SET(p) to create the singleton

Algorithm 1 Computation of attributes and filtering.

```
function
COMPUTE-ATTRIBUTE(S, parent, ima)
    p_root ← S[0]
    for all p ∈ S do
        attr(p) ← f̂(p, ima(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, ima, attr)
    p_root ← S[0]
    if attr(p_root) < λ then out(p_root) ← 0
    else out(p_root) ← ima(p_root)
    for all p ∈ S forward do
        q ← parent(p)
        if ima(q) = ima(p) then
            out(p) ← out(q)        ▷ p not canonical
        else if attr(p) < λ then
            out(p) ← out(q)        ▷ Criterion failed
        else
            out(p) ← ima(p)        ▷ Criterion pass
    return out
```

set $\{p\}$, FIND-ROOT(p) to get the root of the component that contains p, and MERGE-SET(p, q) that merge two disjoints sets of roots p and q. Tarjan discussed two important optimizations in [17] for union-find: root path compression and union-by-rank. Root path compression takes part in FIND-ROOT(p), points on the path from p to the root collapse to the actual root the component. Union-by-rank takes place in MERGE-SET(p, q), when merging two components rooted in p and q, we have to select one to represent the newly created component. If the component of p has a *rank* greater than the one of q then p is selected as the new root, q otherwise. When rank matches the depth of trees, it enables tree balancing and guaranties a better complexity for union-find. Path compression has been applied in [1] and [10], while union-by-rank only in [10].

Flooding algorithms. Those start from a flooding point and perform a propagation. Points in the propagation front are stored in a priority queue so that points at highest level are flooded first, i.e., a depth first propagation. A first implementation has been proposed by [15] which relies on a recursive function FLOOD(p) in charge of flooding p at level $\lambda = ima(p)$ and all points in P_p^λ. When FLOOD(p) returns, the corresponding node has been constructed and is attached to its parent. Hence, when FLOOD(p_{min}) terminates, where p_{min} is a point a lowest gray level in *ima*, the whole image has been flooded and the tree is constructed. To speedup the algorithm, the propagation priority queue is encoded with a hierarchical queue that offers constant time PUSH and POP operations and direct access to points at any level. Salembier et al. [15]'s algorithm was rewritten in a non-recursive implementation by Hesselink [3] and later by Nistér and Stewénius [11] and Wilkinson [23]. These algorithms differ in only two points. First, [23] uses a pass to retrieve the root before flooding to mimic the original recursive version while Nistér and Stewénius [11] does not. Second, priority queues in [11] use an unacknowledged implementation of heap based on hierarchical queues while in [23] they are implemented using a standard heap (based on comparisons).

Merge-based algorithms. Merge-based algorithms consist in computing max-tree on sub-parts of images and merging back trees to get the max-tree of the whole image. Those algorithms are typically well-suited for parallelism since they adopt a map-reduce idiom [21]. 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. When blocks are image lines, a dedicated 1D max-tree algorithm can be used [7, 8]. Figure 2 shows an example of parallel processing using map-reduce idiom. 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 (number of chunks no greater than 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 library such as Intel's TBB [14].

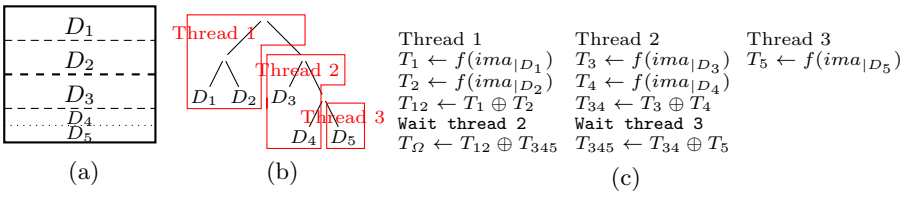


Fig. 2: Map-reduce idiom for max-tree computation. (a) Sub-domains of *ima*. (b) A possible distribution of jobs by threads. (c) Map-reduce operations where f is the map operator, $\oplus$ the merge operator, and $ima_{|D_n}$ denotes the image restricted to sub-domain D_n.

2.5 Algorithms implementation

Immersion algorithms An implementation of max-tree based on union-find without union-by-rank can be found in [1] and an implementation using union-by-rank in [10]. We adapted the latter to build *parent* and S without extra cost. Here, we only provide the principles of our new implementation of union-by-rank and a lighter that uses less memory called level compression.

Union-by-rank. Our implementation is similar to that of [1] but augmented with union-by-rank. The basis of the algorithm resides in two images *parent* and *zpar* representing two trees. *Parent* encodes the max-tree while *zpar* the underlying tree for tracking disjoint sets with union-find. Without *rank* balancing, root of any component in *zpar* matches the root of the corresponding max-tree in *parent*. When using union-by-rank to merge components from *zpar*, we loose this property. Therefore, we introduce an new image *repr* that keeps a connection between the root of the set in *zpar* and the root of the max-tree in *parent* updated. This method is slightly different from the one of Najman and Couprie [10]. They use two union-find structures, one of them dedicated to handle flat zones, while our implementation only uses a single one for *zpar*.

Level compression. Union-by-rank provides time complexity guaranties at the price of extra memory requirement. When dealing with huge images this results in a significant drawback (e.g. RAM overflow...) Without rank technique, the last point processed always becomes the root of the component, i.e. in most cases, we

merge a deep tree to a single node that tends to create a degenerated tree in flat zones. Level compression avoids this behavior by a special handling of flat zones. Let p be the point in process at level λ, 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. If z_p and z_n have a same gray level, they belong to the same node and we can choose any of them as a canonical element. Normally z_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. The remaining part of the algorithm stays unchanged *w.r.t.* [1].

Flooding algorithms The first recursive flooding algorithm was proposed in [15]. A non-recursive version implemented with hierarchical queues can be found in [11], and the one relying on a standard heap in [23]. Those algorithms have been slightly modified to use *parent* and S representation with no other modification.

Merge-based algorithms Algorithms used to merge two trees can be found in [21, 7]. As, implementation of the special 1D max-tree algorithm used when sub-domains are image lines has been proposed in [8]. Our implementations match the ones proposed in those papers. The major difference resides in a post-processing to ensure tree canonicalization and S construction. Indeed, once sub-trees have been computed and merged into a single tree, it does not hold any canonical property (because non-canonical elements are not updated during merge). In addition, the reduction step does not merge the S array corresponding to sub-trees (it would imply reordering S which is more costly than just recomputing it at the end). Algorithm 2 performs canonicalization and reconstructs the 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 point 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 into S and holds the canonical property, thus we can safely insert p back in S and canonicalize p.

Algorithm 2 Canonicalization and S computation algorithm.

```
procedure CanonizeRec(p)
    dejavu(p) ← true
    q ← parent(p)
    if not dejavu(q) then                          ▷ Process parent before p
        CanonizeRec(q)
    if ima(q) = ima(parent(q) then                 ▷ Canonize
        parent(p) ← parent(q)
    InsertBack(S, p)

for all p do dejavu(p) ← False
for all p ∈ Ω such that not dejavu(p) do
    CanonizeRec(p)
```

Implementation details Algorithms have been implemented in pure C++ using an STL implementation of some basic data structures (heaps, priority queues), Milena[5] image processing library to provide fundamental image types and I/O functionality, and Intel TBB for parallelism. Specific implementation optimizations are the following. Sorting is optimized by switching to counting sort when quantization is lower than 18 bits. For large integers of q bits, it switches to 2^{16}-

based radix sort requiring $q/16$ counting sort. For immersion algorithms, 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. In our non-recursive implementation of Salembier (called non-recursive Salembier below) priority-queues are implemented with hierarchical queues (i.e., Nistér and Stewénius [11]'s implementation) and switches to the STL standard heap implementation[1], with a *pre-allocation* for data as well, when the number of bits exceeds 18 (i.e., Wilkinson [23]'s one). A y-fast trie can be used for large integers ensuring a better complexity (see Section 3.1) but no performance gain has been obtained. Finally, in parallel versions of the algorithms, all instructions that deal about S construction and *parent* canonicalization have been removed since S is reconstructed and *parent* canonicalized by Algorithm 2.

3 Algorithms comparison

3.1 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.

Immersion algorithms 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. [2]. 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.

Flooding algorithms 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 [15] or [11], 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, 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.

Merge-based algorithms complexity depends on $\mathcal{A}(k,n)$, the complexity of the underlying method used to compute max-tree on sub-domains. Let $s = 2^h$ the number of sub-domains. The map-reduce algorithms requires s mapping operations

[1] Please note that the authors have verified that choosing a particular implementation of STL (namely STLport 4.6 vs. gcc 4.7) does *not* impact the results presented here.

[2] $\alpha(n)$, the inverse of Ackermann function, is very low growing, $\alpha(10^{80}) \simeq 4$.

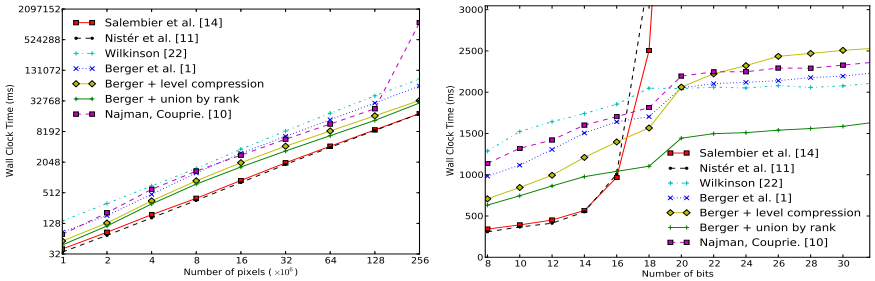
and $s-1$ merges. A good map-reduce algorithm would split the domain to form a full and complete tree so we assume all leaves to be at level h. Merging subtrees of size $n/2$ has been analyzed in [21] and is $O(k \log n)$ (we merge nodes of every k levels using union-find without union-by-rank). Thus, 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 in now dependent on n. This leads to the Matas et al. [7] 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 (`CanonizeRec` is called only once for each point) and only use an image of n elements to track already processed points.

Table 1: Time complexity of many max-tree algorithms compared. n is the number of pixels and k the number of gray levels.

	Time Complexity		
Algorithm	Small int	Large int	Generic V
Berger [1]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Berger + rank	$O(n\, \alpha(n))$	$O(n \log \log n)$	$O(n \log n)$
Najman and Couprie [10]	$O(n\, \alpha(n))$	$O(n \log \log n)$	$O(n \log n)$
Salembier et al. [15]	$O(nk)$	$O(nk) \simeq O(n^2)$	N/A [3]
Nistér and Stewénius [11]	$O(nk)$	$O(nk) \simeq O(n^2)$	N/A [3]
Wilkinson [23]	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Salembier non-recursive	$O(nk)$	$O(n \log \log n)$	$O(n \log n)$
Menotti et al. [8] (1D)	$O(n)$	$O(n)$	$O(n)$
Map-reduce	$O(A(k,n))$	$O(A(k,n))+$ $O(k\sqrt{n} \log n)$	$O(A(k,n))+$ $O(k\sqrt{n} \log n)$
Matas et al. [7]	$O(n)$	$O(n) + O(k\sqrt{n}(\log n)^2)$	-

Table 2: Space requirements of many max-tree algorithms compared. n is the number of pixels and k the number of gray levels.

	Auxiliary space requirements		
Algorithm	Small int	Large int	Generic V
Berger et al. [1]	$n + k + stack$	$2n + stack$	$n + stack$
Berger + rank	$3n + k + stack$	$4n + stack$	$3n + stack$
Najman and Couprie [10]	$5n + k + stack$	$6n + stack$	$5n + stack$
Salembier et al. [15]	$3k + n + stack$	$2k + n + stack$	N/A[3]
Nistér and Stewénius [11]	$2k + 2n$	$2k + 2n$	N/A[3]
Wilkinson [23]	$3n$	$3n$	$3n$
Salembier non-recursive	$2k + 2n$	$3n$	$3n$
Menotti et al. [8] (1D)	k	n	n
Matas et al. [7]	$k + n$	$2n$	$2n$
Map-reduce	$\ldots + n$	$\ldots + n$	$\ldots + n$



(a) (b)

Fig. 3: (a) Algorithms comparison on a 8-bit image as a function of size; (b) Algorithms comparison on a 6.8 Mega-pixels image as a function of quantization.

3.2 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 6.8 MB 8-bit image which was 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. Figure 3 depicts performance of sequential algorithms w.r.t to image size and quantization. As a first remark, we notice that all algorithms are linear in practice. On natural images, the upper bound $n \log n$ complexity of the Wilkinson [23] and Berger et al. [1] algorithms is not reached. Let start with union-find based algorithms. Berger et al. [1] and Najman and Couprie [10] have quite the same running time ($\pm 6\%$ on average), however the performance of Najman and Couprie [10] algorithm drops significantly at 256 Mega-pixels. Indeed, at that size each auxiliary array/image requires 1 GB memory space, thus Najman and Couprie [10] 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 Najman and Couprie [10]. Level compression is an efficient optimization that provides 35% speedup on average on Berger et al. [1]. However, this optimization is only reliable on low quantized data. Figure 3b 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. 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, Salembier et al. [15] and Nistér and Stewénius [11] (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 Figure 3b. By using a standard heap instead of hierarchical queues, Wilkinson [23] does scale well with the number of bits and outperforms every algorithms except our implementation of union-by-rank. In [23], the algorithm is supposed to match

[3] Note always available because hierarchical queues requires V to be expressible as an index.

Salembier et al. [15]'s method for low quantized images, but in our experiments it remains 4 times slower. Since Najman and Couprie [10]'s algorithm is always outperformed by our implementation of union-find by rank, it will not be tested any further. Furthermore, because of the strong similarities of [11] and [23], 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.

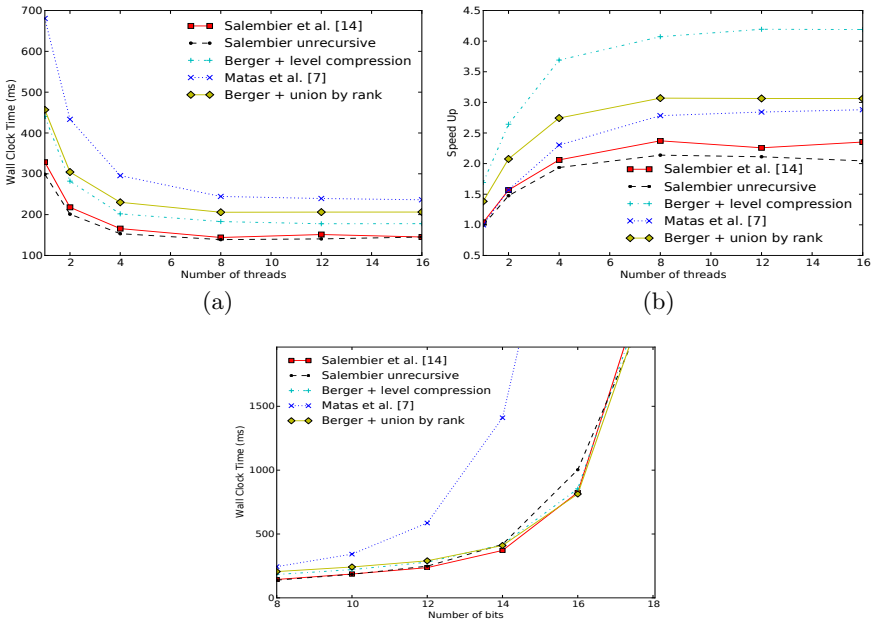


Fig. 4: (a,b) Parallel algorithms comparison on a 6.8 Mega-pixels 8-bits image as a function of number of threads. (a) Wall clock time; (b) speedup w.r.t the sequential version; (c) Parallel algorithms comparison using 8 threads on a 6.8 Mega-pixels image as a function of quantization.

Figure 4 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 ($\times 4.2$) at 12 threads and union-find by rank a $\times 3.1$ speedup with 8 threads. 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 balance tree merges overheads. On the other hand, flooding

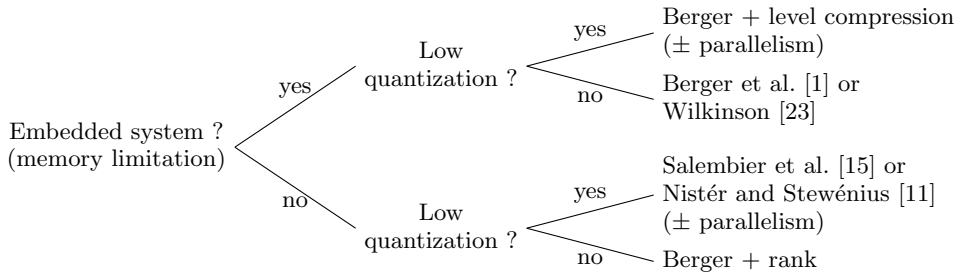


Fig. 5: Decision tree to choose the appropriate max-tree algorithm.

algorithms do not scale as well because they are limited by canonicalization and S reconstruction post-process (that is going to happen as well for union-find algorithms on architectures with more cores). In [21] and [7], 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. Figure 4c 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, Figure 5 provides some guidelines on how to choose the appropriate max-tree algorithm *w.r.t.* to image types and architectures.

4 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 [10]. Furthermore, we proposed a second one that uses level compression for systems with strict memory constraints. As further work, we shall include image contents as a new parameter of comparison, for instance images with large flat zones (e.g. cartoons) or images having strongly non-uniform distribution of gray levels. Extra-materials including algorithm pseudo-codes and descriptions can be found in the appendix of this paper [2] and a "reproducible research" code, intensively tested, is available on the Internet at `http://www.lrde.epita.fr/Olena/maxtree`.

Bibliography

[1] Berger, C., Géraud, T., Levillain, R., Widynski, N., Baillard, A., Bertin, E.: Effective component tree computation with application to pattern recognition in astronomical imaging. In: Proc. of ICIP. vol. 4, pp. IV–41 (2007)

[2] Carlinet, E., Géraud, T.: Appendix of the present paper (2013), `http://www.lrde.epita.fr/Olena/maxtree`

[3] Hesselink, W.H.: Salembier's min-tree algorithm turned into breadth first search. Information Processing Letters 88(5), 225–229 (2003)

[4] Jones, R.: Connected filtering and segmentation using component trees. Computer Vision and Image Understanding 75(3), 215–228 (1999)

[5] Levillain, R., Géraud, T., Najman, L.: Why and how to design a generic and efficient image processing framework: The case of the Milena library. In: Proc. of ICIP. pp. 1941–1944, http://olena.lrde.epita.fr (2010)
[6] Matas, J., Chum, O., Urban, M., Pajdla, T.: Robust wide-baseline stereo from maximally stable extremal regions. IVC 22(10), 761–767 (2004)
[7] Matas, P., Dokladalova, E., Akil, M., Grandpierre, T., Najman, L., Poupa, M., Georgiev, V.: Parallel algorithm for concurrent computation of connected component tree. In: Adv. Concepts for Intelligent Vis. Sys. pp. 230–241 (2008)
[8] Menotti, D., Najman, L., de Albuquerque Araújo, A.: 1D component tree in linear time and space and its application to gray-level image multithresholding. In: Proc. of Intl. Symp. on Math. Morphology. pp. 437–448 (2007)
[9] Monasse, P., Guichard, F.: Fast computation of a contrast-invariant image representation. IEEE Trans. on Image Processing 9(5), 860–872 (2000)
[10] Najman, L., Couprie, M.: Building the component tree in quasi-linear time. IEEE Transactions on Image Processing 15(11), 3531–3539 (2006)
[11] Nistér, D., Stewénius, H.: Linear time maximally stable extremal regions. In: Proc. of European Conf. on Computer Vision. pp. 183–196 (2008)
[12] Ouzounis, G.K., Wilkinson, M.H.F.: Mask-based second-generation connectivity and attribute filters. IEEE Transactions on Pattern Analysis and Machine Intelligence 29(6), 990–1004 (2007)
[13] Perret, B., Lefevre, S., Collet, C., Slezak, É.: Connected component trees for multivariate image processing and applications in astronomy. In: Proc. of International Conference on Pattern Recognition. pp. 4089–4092 (2010)
[14] Reinders, J.: Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Incorporated (2007)
[15] Salembier, P., Oliveras, A., Garrido, L.: Antiextensive connected operators for image and sequence processing. IEEE Transactions on Image Processing 7(4), 555–570 (1998)
[16] Salembier, P., Serra, J.: Flat zones filtering, connected operators, and filters by reconstruction. IEEE Trans. on Ima. Proc. 4(8), 1153–1160 (1995)
[17] Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. Journal of the ACM 22(2), 215–225 (1975)
[18] Urbach, E.R., Wilkinson, M.H.F.: Shape-only granulometries and grey-scale shape filters. In: Proc. of ISMM. pp. 305–314 (2002)
[19] Vincent, L.: Grayscale area openings and closings, their efficient implementation and applications. In: Proc. of EURASIP Workshop on Mathematical Morphology and its Applications to Signal Processing. pp. 22–27 (1993)
[20] Westenberg, M.A., Roerdink, J.B.T.M., Wilkinson, M.H.F.: Volumetric attribute filtering and interactive visualization using the max-tree representation. IEEE Trans. on Image Processing 16(12), 2943–2952 (2007)
[21] Wilkinson, M.H.F., Gao, H., Hesselink, W.H., Jonker, J.E., Meijster, A.: Concurrent computation of attribute filters on shared memory parallel machines. IEEE Trans. on PAMI 30(10), 1800–1813 (2008)
[22] Wilkinson, M.H.F., Westenberg, M.A.: Shape preserving filament enhancement filtering. In: Proc. of MICCAI. pp. 770–777 (2001)
[23] Wilkinson, M.H.F.: A fast component-tree algorithm for high dynamic-range images and second generation connectivity. In: ICIP. pp. 1021–1024 (2011)
[24] Xu, Y., Géraud, T., Najman, L.: Morphological filtering in shape spaces: Applications using tree-based image representations. In: Proc. of International Conference on Pattern Recognition. pp. 1–4 (2012)