University of Groningen

Distributed Connected Component Filtering and Analysis in 2-D and 3-D Tera-Scale Data Sets

Gazagnes, Simon; Wilkinson, Michael H. F.

Published in:
IEEE transactions on image processing

DOI:
[10.1109/TIP.2021.3064223](https://doi.org/10.1109/TIP.2021.3064223)

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2021

[Link to publication in University of Groningen/UMCG research database](#)

Citation for published version (APA):

Gazagnes, S., & Wilkinson, M. H. F. (2021). Distributed Connected Component Filtering and Analysis in 2-D and 3-D Tera-Scale Data Sets. *IEEE transactions on image processing*, 30, 3664-3675.
<https://doi.org/10.1109/TIP.2021.3064223>

Copyright

Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license. More information can be found on the University of Groningen website: <https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment>.

Take-down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): <http://www.rug.nl/research/portal>. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Distributed Connected Component Filtering and Analysis in 2D and 3D Tera-Scale Data Sets

Simon Gazagnes¹ and Michael H. F. Wilkinson², *Senior Member, IEEE*

Abstract—Connected filters and multi-scale tools are region-based operators acting on the connected components of an image. Component trees are image representations to efficiently perform these operations as they represent the inclusion relationship of the connected components hierarchically. This paper presents DISCCOFAN (DISTRIBUTED Connected Component Filtering and ANALYSIS), a new method that extends the previous 2D implementation of the Distributed Component Forests (DCF) to handle 3D processing and higher dynamic range data sets. DISCCOFAN combines shared and distributed memory techniques to efficiently compute component trees, user-defined attributes filters, and multi-scale analysis. Compared to similar methods, DISCCOFAN is faster and scales better on low and moderate dynamic range images, and is the only method with a speed-up larger than 1 on a realistic, astronomical floating-point data set. It achieves a speed-up of 11.20 using 48 processes to compute the DCF of a 162 Gigapixels, single-precision floating-point 3D data set, while reducing the memory used by a factor of 22. This approach is suitable to perform attribute filtering and multi-scale analysis on very large 2D and 3D data sets, up to single-precision floating-point value.

Index Terms—Mathematical morphology, component-trees, connected filters, hierarchical image representation, multi-scale representation, hybrid algorithm, multithreading, MPI, high-performance computing.

I. INTRODUCTION

WHILE classical morphological operators work on a pixel-based representation, the development of connected filters [1] provided an efficient toolbox to perform segmentation and classification in applications working with the notion of region-based objects, i.e., organs or cells in medical analysis or infrastructures in remote-sensing. Component trees [1] are powerful hierarchical structures representing all connected components at all threshold sets of an image. They provide a suitable approach for region-based analysis, including various filtering strategies and multi-scale tools [2]. They have been applied in astronomy [3]–[5], remote-sensing [6], [7], or medical analysis [8], [9]. With the development of image acquisition techniques, data sets have become larger and more complex to process. Wilkinson *et al.* [10] proposed

the first concurrent algorithm to perform attribute filtering and multi-scale analysis on shared-memory machines, paving the way for applications with Giga-Scale data sets. However, these techniques fail to handle Tera-Scale images on modest computing machines because the component tree size becomes too large to be stored using a single compute node. To solve this, Kazemier *et al.* [11] introduced the notion of *Distributed Component Forests* (DCF). The latter approach performs area opening on low dynamic range data sets using distributed memory techniques. Gazagnes & Wilkinson [12] extended these techniques to 16 bit per pixel (bpp) data sets, and implemented multi-scale operators within the DCF framework. Recently, Götz *et al.* proposed a novel approach to compute component trees using both shared-memory and distributed memory techniques [13]. However, the latter does not include any attribute computation, filtering or multi-scale operations, limiting its application.

These developments laid the foundations for Tera-Scale data sets processing. Further improvements are still needed to handle larger dynamic range data sets (32-bpp or single precision floating point), or 3D volumes. In this paper, we introduce DISCCOFAN (DISTRIBUTED Connected Component ANALYSIS), a method that combines shared and distributed memory techniques and can perform a wide range of attribute filtering and multi-scale operators to analyze, extract or filter connected components in Tera-Scale 2D and 3D data sets with low to large dynamic ranges. The paper is organized as follows: Section II introduces the theoretical framework of attribute filtering and multi-scale operators. Section III presents the component tree representation and details the state of art of the sequential and parallel construction approaches. Section IV describes the implementation of DISCCOFAN, and Section V discusses the performance of our method. Finally, Section VI concludes this paper, summarizing the main assets of DISCCOFAN and considers the next steps for future improvements.

II. CONNECTED COMPONENT ANALYSIS

A. Connected Components

Discrete images can be represented in term of un-directed graphs (V, E) , with V the set of vertices (pixels) and E the set of edges. Graphs are *connected* if, for every $x_i \in V$ and any x and $y \in V$ such that $x = x_0$ and $y = x_n$, there exists a path $(x_0, \dots, x_i, \dots, x_n)$ with $(x_i, x_{i+1}) \in E$. A gray-level image I can be described using a function $f : V \rightarrow \mathcal{R}$, that assigns a gray value to all the vertices. The connected components characterize the flat zones of a data set, where f is constant.

Manuscript received September 29, 2020; revised January 27, 2021; accepted February 19, 2021. Date of publication March 11, 2021; date of current version March 17, 2021. The work of Simon Gazagnes was supported by the Centre for Data Science and Systems Complexity and this paper is based on research developed in the DSSC Doctoral Training Programme, co-funded through a Marie Skłodowska-Curie COFUND (DSSC 754315). The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Ming-Ming Cheng. (Corresponding author: Simon Gazagnes.)

The authors are with the Bernoulli Institute, University of Groningen, 9700 AK Groningen, The Netherlands (e-mail: s.r.n.gazagnes@rug.nl; m.h.f.wilkinson@rug.nl).

Digital Object Identifier 10.1109/TIP.2021.3064223

1941-0042 © 2021 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See <https://www.ieee.org/publications/rights/index.html> for more information.

In images or volumes, the connected components are usually defined in terms of groups of N-connected pixels, where N is the number of spatially adjacent neighbors of each pixel, at different threshold sets.

B. Attribute Filters and Multi-Scale Analysis

Attribute filters [14] are morphological tools that act on the connected components of I , using a criterion $T : \mathcal{P}(V) \rightarrow \{\text{false}, \text{true}\}$, using an attribute function $\mathcal{A} : \mathcal{P}(V) \rightarrow \mathbb{R}$. Applying these operators to gray-scale images consists of computing threshold images at all levels $l \in \mathbb{R}$ defined as

$$T_l(f) = \{x \in E \mid f(x) \geq l\}. \quad (1)$$

The attribute filter is then applied on each threshold set $T_l(I)$, and the resulting images are stacked up to obtain the filtered output. The complexity of the filtering process depends on the *increasing* or *non-increasing* characteristic of the attribute function $\mathcal{A}$. Different filtering approaches exist for *non-increasing* attribute functions such that they might preserve, remove, or modify C based on the nested relations of the components (see [1]). The use of region-based structures, such as *Component Trees* (see Section III), is particularly efficient to perform these attribute filtering operations. Additionally, Xu *et al.* [9] recently presented a new class of novel connected operators called *morphological shapings* where the filtering is performed in the space of shapes built from the image, extending the existing tree-based connected operators. These techniques are particularly efficient to perform robust object extraction.

Multi-scale operators also provide an efficient approach to analyze the image based on the distribution of the components' attributes. One of these tools is the pattern spectrum [15], obtained by applying a set of consecutive filters and extracting the subset of image content that is removed at each step. An other approach is the *differential attribute profile* (DAP) [16] and its compact representation known as the Characteristic-Saliency-Leveling (CSL) model [2]. In [12], we described an efficient implementation of these tools for distributed tree representations. Our proposed method includes these tools, as well with different attribute functions and pruning strategies. For space-consideration, all the tests performed in this work only consider the simple morphological area opening [17].

III. THE COMPONENT TREE

Trivial implementations of connected operator applied to classical pixel-based representations are highly computationally expensive as the number of gray levels and the size of the data set increases. Component trees are hierarchical, region-based representations encoding the inclusion relationship of the connected components in the image and offer more flexibility to perform connected filtering or multi-scale analyses. Typical component tree representations are the Max-Tree and Min-Tree structures [1], where the leaves respectively represents the maxima and minima in the data set. Each node in the tree represents a connected component of I , and is represented by a single element, called *levelroot* or *canonical element*, which characterizes the entire node. The *levelroot*

Fig. 1. An example max-tree. (a) The input image. (b) The thresholded sets at each level. (c) The resulting Max-Tree representation. Figure adapted from [12].

points in turn to the canonical element of the parent node. The lower node in the component tree is the *root*, and does not point to any other node. Hence, a component tree encodes the parent-child relation, and attributes computation if any, of all the connected components of an image. Filtering, or analyzing the attribute distribution function of the latter can be simply achieved by manipulating the levelroots, which are usually in a much lower number than the number of pixels in the image in low and moderate dynamic range vast images. An example of max-tree for a gray-scale image is in Figure 1.

A. Tree Construction

Several sequential approaches to construct a Component Tree exist, typically grouped in two categories: *root-to-leaf flooding* and *leaf-to-root merging*. Leaf-to-root merging algorithms include the Najman and Couprie [18] algorithm, using a union-find approach originally introduced by Tarjan [19], and the Berger *et al.* [4] approach, which performs well for finely quantized images (>16 bpp to floating point).

By contrast, root-to-leaf flooding approaches are based on the idea of Salembier *et al.* [1]. The construction starts from the pixel with the lowest intensity and recursively propagates to its neighbors at the highest levels. The original implementation had a memory footprint scaling linearly with the number of grey levels G and the number of pixels S , hence was too expensive for high-dynamic range. Wilkinson [20] solved this caveat by combining a stack and a hierarchical queue such that the memory footprint does not depend on G . Carlinet *et al.* compared the performance state-of-the-art algorithms [21], and we refer the reader to the latter review for more details on sequential algorithms.

In [22], an improved version the algorithm in [20] was introduced, with a smaller memory footprint and a better time performance than the state-of-the art flooding algorithms ([4], [20], [23]) for images from 8-bit to 32-bit. Teeniga's main improvement arises from the use of a Trie priority queue instead of a heap priority queue. The pixel values are first mapped into unique ranks and the Trie priority queue operates on the prefixes of the ranks, which is faster than operating on the original pixel values. Algorithm 1 shows the construction procedure defined in [22], and a complete description of each step can be found there. In Section IV, we detail how we adapted and included Algorithm 1 in our proposed method to perform the construction of the local component trees in each MPI node.

B. Parallelization Strategies, Literature and Limitations

Sequential algorithms become computationally costly, both in time and memory, when the image size gets larger. Efficient parallelization strategies are required to improve their

Algorithm 1 Pseudo-Code of Algorithm [22]

```

1: procedure TREE_CONSTRUCTION( $f$ )
2:    $parents \leftarrow \text{ALLOC\_EMPTY\_ARRAY}(\text{size\_image})$ 
3:    $\text{trie\_prio\_queue} \leftarrow \text{INIT\_TRIE\_PRIORITY\_QUEUE}()$ 
4:   mark all nodes as unvisited
5:    $\text{node} \leftarrow$  node with index 0
6:   mark  $\text{node}$  as visited
7:   while true do
8:     while  $\text{node}$  has unvisited neighbors do
9:        $\text{neighbor} \leftarrow$  next unvisited neighbor
10:      mark  $\text{neighbor}$  as visited
11:      if  $f(\text{neighbor}) < f(\text{node})$  then
12:         $\text{TRIE\_PRIO\_QUEUE\_INSERT}(\text{neighbor})$ 
13:      else
14:         $\text{TRIE\_PRIO\_QUEUE\_INSERT}(\text{node})$ 
15:         $\text{node} \leftarrow \text{neighbor}$ 
16:      end if
17:    end while
18:    if  $\text{trie\_prio\_queue.empty}()$  then
19:      break
20:    end if
21:     $parents[\text{node}] \leftarrow \text{TRIE\_PRIO\_QUEUE\_TOP}()$ 
22:     $\text{node} \leftarrow \text{TRIE\_PRIO\_QUEUE\_TOP}()$ 
23:     $\text{TRIE\_PRIO\_QUEUE\_REMOVETOP}()$ 
24:  end while
25:   $parents[\text{node}] \leftarrow \text{root}$ 
26:  return  $parents$ 
27: end procedure

```

computation and make them suitable for applications which have high requirements in term of speed or storage. However, parallelizing connected operators is challenging because they are region-based procedures, thus global and non-separable. We first detail the shared-memory merging algorithms in Section III-B.1, and then focus on the distributed memory approaches in Section III-B.2.

1) *Shared-Memory Case*: The first shared-memory algorithm proposed by Wilkinson *et al.* [10] was based on a *divide-and-conquer* approach. The image is first divided in N_t smaller tiles, where N_t is the number of shared-memory threads. Each process builds the Component Tree representation of the tile assigned to it in a concurrent way. Because connected components can span over several independent tiles, the construction process requires a second stage, the merging phase, which is built on a synchronized mechanism that re-connects sub-trees 2 by 2. The fusion of two sub-trees is handled through a specific *merging* strategy which ensures that the parents and attributes are correctly reconnected when sub-trees are merged. This procedure is described in details in Algorithm 3 in [10]. At each step i , the number of sub-tree pairs being merged is $N_t / 2^i$, and the total number of concurrent steps needed to obtain the final tree is $N_t / 2$ if N_t is even, and $(N_t - 1) / 2 + 1$ if N_t is odd. The algorithm uses semaphores to ensure that each merging step only starts if the previous step is completed. The final step is handled by the process 0. All the individual sub-trees are merged into

the final component tree, which correctly represents the full image.

This approach has some limitations. Re-connecting two sub-trees requires going through the neighboring border nodes, which can become very costly if the length of the respective branches is large. For high dynamic range images, the length of the sub-tree branches can potentially be as long as the number of gray levels in the image, which limits this merging approach. Therefore, Moschini *et al.* [24] proposed a hybrid approach, where the image is divided in intensity levels, rather than in space. It used a *pilot-tree* which encodes a low-level quantization approximation of the image and is refined in parallel, such that each thread deals with a given quantization level of the tree. This approach gave a significant speed-up performance on double precision, floating-point data, providing a suitable approach to use multi-threading on extreme dynamic range data.

Wilkinson *et al.* [10] also implemented the parallelization of the post-processing of the tree (i.e attribute filtering operations), which can be done in a concurrent manner in the shared-memory case because each thread can access the full tree. Overall, the shared-memory parallelization strategies from [10] and [24] have been proved efficient, both for low and higher dynamic range images. However, they are limited to the size of the data itself, because the component tree of the entire image needs to fit in the memory of the shared-memory machine. The emergence of Tera-Scale data sets in a wide range of fields, from satellite imaging to astronomy, required a novel parallelization strategy using distributed memory techniques.

2) *Distributed-Memory Case*: Parallelization strategies using distributed memory machines are more complex because one needs to make sure that the communication between the individual nodes does not become prohibitive during the tree construction. Two different approaches have emerged in the past few years: the *Distributed Component Forests* (DCF) [11], [12] and the distributed computation of the Component Tree [13]. Both use a similar *divide-and-conquer* approach, where the image is first divided in N_p tiles, assigned to N_p individual nodes. The local component trees are then concurrently built such that each node contains the sub-tree corresponding to its assigned tile. Sub-trees then need to be corrected through a procedure that avoids transmission of many, large messages across the communication interface. To do this, both methods use different procedures. The DCF approach uses the *Boundary Tree* structure. Boundary Trees only include the subset of the branches related to the connected components on the edges of the tiles that need to be corrected. These sub-trees have much smaller memory footprint, less than 1% in low dynamic range cases [12] than the local Component Trees, which makes them suitable for inter-node communication. The Boundary Trees are then merged similarly to the shared-memory strategy: Boundary Trees are fused 2 by 2, such that one process sends its Boundary Tree to the neighboring process who receives it and merges it with its own Boundary Tree stored locally. The number of merging steps required is $N_p / 2$ if N_p is even, and $(N_p - 1) / 2 + 1$ if N_p is odd. Once all the sub-trees have been merged,

the updated nodes need to be successively propagated back to each individual Boundary Tree. Hence, a second phase is required, which is the update phase introduced in [12]. The updated information from the last merging step $i + 1$ is propagated back to the merging step i until we reach the local boundary trees. The latter are used to correct the local Component Trees, such that, by the end of the updating phase, we have a forest of N_p Component Trees, each one encoding all the information it needs to perform attribute filtering or multi-scale operations.

In the approach of Götz *et al.* [13], the entire Component Tree is computed and stored in a distributed manner. Hence, each process contains a subset of the entire component tree, such that a tree node in process p might point to a parent in an other process. To perform the tree computation in a distributed manner, the authors define the *tuple* which contains information about the parents at a same level in different sub-trees. This approach is possible because each image chunk includes a 1-pixel redundant halo-zone, such that the parent merging can be done on the same pixel in the individual trees. One main advantage of this method is that it does not require the duplication of nodes, while it is necessary in the DCF-approach to be able to process all the local Component Trees individually. However, this method only computes the parent-child relations in the image, but does not include any attributes computation, or post-processing of the Component Tree. While attributes could in theory be added in the tuple structure and corrected during the tuple resolution stage, including connected filters or multi-scale tools would require a significant extension to avoid a communication overhead between the individual processes. Therefore, it might not be suitable to perform complex operations on the distributed tree structure once the latter has been built.

IV. DISTRIBUTED CONNECTED COMPONENT FILTERING AND ANALYSIS

This Section describes the implementation of DISCCOFAN (DIStributed Connected COmponent Filtering and ANalysis). It combines and improves the most efficient state-of-art flooding and merging algorithms to process higher dynamic range data sets than currently available algorithms. The structure of the algorithm can be summarized as:

- Load and split the images in N_p tiles, with a 1-pixel overlap, which are distributed to N_p MPI processes
- Build the local trees for the N_p tiles
- Extract, merge and combine the boundary trees of the individual tiles
- Propagate the updated attributes and parent-child relations from the last merging step to the local component trees
- (optional) Filter or apply multi-scale analysis tool on the Distributed Component Forest.

We detail each step in the next sub-sections.

A. Building the Local Component Trees

The data set is first divided into a grid of $N_p = H \times V \times D$ tiles, where N_p is the total number of MPI processes, and H , V and D represents the grid division in the horizontal, vertical, and depth direction (with $D = 1$ for 2D images).

Contrary to the original implementation [11], W , H and D can be any natural integer larger than 0. Additionally, each tile includes a redundant overlap of 1 pixel from its neighbors, to improve the merging approach of the connected components in the tiles border (detailed further in Section IV-B). Hence, a $100 \times 100 \times 100$ pixels data set divided into a $2 \times 2 \times 2$ grid lead to 8 tiles of size $51 \times 51 \times 51$ pixels.

Once the data set has been read and divided, the N_p processes concurrently build the component tree representation of their tile. The local tree structure contains three main arrays: *gval*, *parent* and *attribute* which respectively store the pixels intensity, the parent-child relations and the attributes of the connected components. The *attribute* array is initialized for all pixels before the flooding. Then, both the *parent* and *attribute* are computed using Algorithm 1. The propagation of the attributes is achieved by adding one additional step at line 22 in the Algorithm 1. It ensures that the attribute of the current node is merged with the attribute of its parent, using the function `MERGE_AUX_DATA(attribute[node], attribute[parents[node]])`. The latter handles the attribute propagation using a system of auxiliary data structures and functions, which allows to use a wide range of attributes.

Additionally, we added a shared-memory parallelization to Algorithm 1 based on the strategy from Wilkinson *et al.* [10]. This makes it possible to use several threads per MPI node to decrease the execution time of each local component tree when the dynamic range of the image is not too large ($\lesssim 20$ bpp [24]). The hybrid performance of DISCCOFAN is discussed in Section V.

The final local component tree representation is built using all the pixels belonging to that tile, including the 1-pixel overlap in the borders. However, we flag the latter during the initialization of the *attribute* array to make sure that they do not contribute to the attributes of the connected components, since these pixels belong to an other tile.

B. Optimal Selection of Border Nodes

At the end of the building stage, each tile is represented by a component tree that must be corrected such that filtering the forest of individual trees yields the same result as filtering the component tree of the full image. To do this, we use the Boundary Tree structures introduced in [11], [12]. Boundary trees consist of a subset of the local component trees which only includes the nodes in the tile border, and their parents. The `BOUNDARYTREE` structure has four main components. The *BoundaryNode* array contains all information about the levelroots of the connected components in the border, such as the intensity, index of the node in the local component tree, and its index in the Boundary tree. The arrays *attribute* and *border_par* encode respectively the attributes and parent-child relations of the components in the boundary tree. Additionally, *border_ori* and *border_lr* are two additional arrays used to keep track of the origin of each Boundary node, and of the levelroot pairs during the merging phase. We detail their use below.

Originally, boundary trees contained all nodes belonging to the 4 borders of the 2D tile. We extended it to handle 3D data using a mapping approach that allows extraction of the

2D border. Besides, the way the nodes are extracted from the local component trees has been modified to reduce the memory footprint and improve the merging strategy for large dynamic range data sets. The original merging strategy of the boundary trees in [12] was adapted from the shared-memory algorithm [10], and is limited to low and moderate dynamic range. We improved the merging approach by optimally selecting the border nodes to be merged. To show how this can be done, we consider the original merging procedure (Algorithm 3 in [10]). For two pixels x and y , with $f(x) \geq f(y)$, to be merged, the algorithm first traverses all ancestors of x until it finds x_m such that $f(\text{parent}(x_m)) < f(y)$. Hence, we have $f(x) \geq f(y) > f(\text{parent}(x_m))$, such that the parent of x_m needs to be reassigned to y , and its attribute merged with the attributed of y . Once this is done, x becomes y and y becomes the previous parent of x_m . This is repeated until we reach the root of both trees. During the first searching phase (finding x_m), no relevant operation is done as the algorithm simply browses the branch of x . In high dynamic range images, where the contrast between x and y can be large, finding x_m can require many iterations, thus, the computing time increases due to this searching phase. This issue can be solved if one knows a priori the merging point x_m . By including a 1-pixel overlap of the border in each tile, we provide a redundant halo in which a pixel x is included in both max-trees of the neighboring tiles. Two adjacent tiles have a border which is now is 2-pixels wide (the pixels from its original border and the pixels from the border of the neighbor tile). Hence, the optimal pair of nodes x and y to be merged consists of the two nodes representing the same pixel in the two component trees. Note that we could have chosen to have a single 1-pixel overlap (only one tile includes the pixels of its neighbor tile), however, in the low connectivity case, this additional redundancy helps to further improve the selection of boundary nodes since we can choose to perform the merging using only the nodes with the lowest intensities. This ensures that the length of the branches to be merged is minimal. For higher connectivity cases, this is less trivial because nodes in the tile borders have several neighbors. Hence, we need to perform the merging using the redundant nodes with maximal intensity, to ensure that the information is correctly propagated. This means that we maximize the length of tree branches to be merged, which is intuitively counter-productive. Nevertheless, the number of “long branches” to be covered is lower, and also improves the merging performance in 8 or 26 connectivity cases, as illustrated in Table I.

We compare the performance of the merging approach using in DISCCOFAN with that in [12] using a 2D image of 5000×5000 pixels, where the random pixel intensities from a uniform distribution from 0 to $2^{32}-1$. We split the image using a 2×1 grid, and we compare the computational time needed to merge the two boundary trees. The total merging time t_t is divided along the two phases previously described, such that $t_t = t_s + t_c$, where t_s is the searching time (going through the parents of x until $f(\text{parent}(x)) < f(y)$), and t_c the re-connection time (propagating the parent and attributes from x to y). We additionally compare the performance gains for two 2D N-connectivity cases (4 and 8) since the pre-selection process is slightly different in both cases. The results of

TABLE I
COMPARISON OF THE MERGING PERFORMANCE IN DISCCOFAN COMPARED TO THE CODE FROM [12], USING A 5000×5000 , 32-BIT-
PER-PIXEL IMAGE. THE OPTIMAL SELECTION OF THE BOUNDARY
NODES IN DISCCOFAN IMPROVES BOTH THE MERGING TIME,
AND THE MEMORY SIZE OF THE BOUNDARY TREES

N^1	Alg ²	$t_s^3(s)$	$t_c^4(s)$	$t_t^5(s)$	S-U ⁶	M ⁷ (MB)	M-G ⁸
4	[12]	56.0	6.3	62.3	22	3.04	1.72
	TW	0.0	2.8	2.8		1.76	
8	[12]	99.5	17.0	116.5	20	3.04	1.29
	TW	0.0	5.8	5.8		2.34	

¹ N-Connectivity rule

² Algorithm used (TW: This Work)

³ Searching phase time ⁴ Connecting phase time

⁵ Total merging time ⁶ Speed-Up

⁷ Boundary-tree size on process 0 in MegaBytes

⁸ Memory Gain

this test are shown in Table I. By pre-selecting the relevant nodes to be merged, the new approach becomes optimal and greatly improves the performance of the merging phase. Note that Table I shows our new approach also lowers the t_c time because the pairs are optimally selected. We obtain a speed-up (ratio of the total time using the old approach over the total time in the new approach,) of 22 and 20 and a memory gain (ratio of the size of boundary tree stored in the process 0) of 1.72 and 1.23, for the 4 and 8-connectivity cases respectively. This improvement allows us to deal with higher dynamic range images than the state-of-the-art methods, as we show in the Section V.

Details of the new boundary-tree extraction function are shown in Algorithm 2. Once all border nodes have been optimally selected, their ancestors are added to the boundary tree using the function `ADD_PARENTS(local_tree, b_tree)`, directly adapted from [11].

C. Merging Boundary Trees

Algorithm 3 shows the procedure implemented in DISCCOFAN to merge two boundary trees. The procedure `MERGE_BOUNDARY` goes through all the pairs of nodes that need to be merged in the border, using the function `IDX_A` and `IDX_B` to recover the index of the nodes in the boundary trees a and b respectively. The procedure `MERGE_BRANCHES` merges two nodes x and y . This function is largely adapted from the Algorithm 3 in [10] and most of its changes have been previously detailed in [12]. `B_LEVELROOT`, `B_PARENT`, `B_GVAL`, `B_ATTR` return respectively the levelroot of x , the levelroot of the parent of x , the intensity and the attribute of x . Additionally, `B_TREE` returns the memory address of the boundary tree which x belongs to. The functions `MERGE_AUX_DATA` and `CLONE_AUX_DATA` are generic attribute functions that ensure the propagation or copy of any user-defined attributes. Additionally, we create levelroot pairs using the array `border_lr` to handle the cases where two levelroot x and y with $f(x) = f(y)$ are re-connected. In that case, one of them is not levelroot anymore, while the other one becomes the *global* levelroot of the updated connected component. If x and y belong to different boundary

Algorithm 2 Pseudo-Code of the Boundary Tree Extraction Algorithm

```

1: procedure CREATE_BOUNDARY_TREE(ComponentTree
   local_tree, int connectivity)
2:   BoundaryTree  $b\_tree \leftarrow$  ALLOC_BOUNDARY_TREE()
3:    $i \leftarrow 0$ 
4:   for all boundary nodes in the local tree do
5:      $node\_to\_merge \leftarrow$  next node in the tile boundary
6:      $node\_neighbor \leftarrow$  direct neighbor from the
       1-pixel overlap halo
7:     if  $f(node\_neighbor) < f(node\_to\_merge)$  and
        $connectivity < 8$  then
8:        $node\_to\_merge \leftarrow node\_neighbor$ 
9:     else if  $f(node\_neighbor) > f(node\_to\_merge)$  and
        $connectivity \geq 8$  then
10:       $node\_to\_merge \leftarrow node\_neighbor$ 
11:     end if
12:      $b\_tree.array[i] \leftarrow$  LEVELROOT( $node\_to\_merge$ )
13:      $i \leftarrow i + 1$ 
14:   end for
15:   ADD_PARENTS( $local\_tree, b\_tree$ )
16:   return  $b\_tree$ 
17: end procedure

```

trees, then x remains a *local* levelroot of its tree. Thus, the levelroot pairs keep track of the *local* levelroots, which are later useful when we have to disconnect the structures that have been merged into individual trees (Section IV-D). Note that a *global* levelroot is always a *local* levelroot in its original tree. Additionally, if a new node x' is merged with a node belonging to a levelroot pair, its attribute is propagated to the *global* levelroot in the pair, but its parent becomes the *local* levelroot that belongs to the same boundary tree. Hence, once two boundary trees have been merged, all the nodes that are not levelroots of a connected component in the tree point to a *local* levelroot in the same boundary tree. Similarly, all the *local* levelroots either point or are the *global* levelroots of the merged structure. Hence, propagating the information of the *global* levelroots to the *local* levelroots during the update phase can be done efficiently.

The pattern used to merge all the boundary trees is similar to the shared-memory implementation [10]: boundary trees are merged 2-by-2 and combined at each step, such that the total number of merging steps is $\log_2(N_p)$ if N_p is even or $\log_2(N_p - 1) + 1$ if N_p is odd. At each step, two merged boundary trees are combined into a single boundary tree that only includes the *global* levelroots, using the same COMBINE_BOUNDARY procedure implemented in [11], extended to handle 3D processing. The combined trees only include the *global* levelroots of the merged trees and hence is typically smaller than the cumulative size of the two boundary trees. The resulting structure is stored, and further combined with the remaining boundary trees from the neighboring processes. This pattern is repeated until all boundary trees have been merged.

Algorithm 3 Merging the Boundary Trees

```

1: procedure MERGE_BOUNDARY(BoundaryTree  $a$ ,
   BoundaryTree  $b$ , int  $length$ , Direction  $d$ )
2:    $i \leftarrow 0$ 
3:   for  $i < length$  do
4:     BoundaryNode  $x \leftarrow$  IDX_BTREE_A( $a, i, d$ )
5:     BoundaryNode  $y \leftarrow$  IDX_BTREE_B( $b, i, d$ )
6:     MERGE_BRANCHES( $x, y$ )
7:      $i \leftarrow i + 1$ 
8:   end for
9: end procedure

1: procedure MERGE_BRANCHES(BoundaryNode  $x$ ,
   BoundaryNode  $y$ )
2:    $x \leftarrow$  B_LEVELROOT( $x$ )
3:    $y \leftarrow$  B_LEVELROOT( $y$ )
4:    $attr \leftarrow$  null
5:    $attr1 \leftarrow$  null
6:   while  $x \neq y \wedge y \neq \text{root}$  do
7:      $z \leftarrow$  B_PARENT( $x$ )
8:     if  $z \neq \text{root}$  and  $f(z) \geq f(y)$  then
9:       MERGE_AUX_DATA(B_ATTR( $x$ ),  $attr$ )
10:       $z \leftarrow x$ 
11:     else
12:       if  $f(x) = f(y)$  and  $B\_TREE(x) \neq B\_TREE(y)$  then
13:         if levelroot pair already exists then
14:           Connect  $x$  and  $y$  to the previous levelroot pair
15:         else
16:           Create a levelroot pair  $x-y$ 
17:            $B\_PARENT(x) \leftarrow y$ 
18:         end if
19:       else
20:          $B\_PARENT(x) \leftarrow y$ 
21:       end if
22:        $attr1 \leftarrow$  ADD_AUX_DATA( $attr, B\_ATTR(x)$ )
23:        $attr \leftarrow$  CLONE_AUX_DATA( $B\_ATTR(x)$ )
24:        $B\_ATTR(x) \leftarrow$  CLONE_AUX_DATA( $attr1$ )
25:        $x \leftarrow y$ 
26:        $y \leftarrow z$ 
27:     end if
28:   end while
29:   if  $y == \text{root}$  then
30:     while  $x \neq \text{root}$  do
31:       MERGE_AUX_DATA( $B\_ATTR(x), attr$ )
32:        $z \leftarrow$  B_PARENT( $x$ )
33:     end while
34:   end if
35: end procedure

```

D. Updating the Trees

During the merging phase, the connected components are successively updated to obtain the correct parent-child relations and attributes. However, because each combined tree only includes the *global* levelroots at each step, the correct parent-child relation and attribute information need to be propagated back to the *local* levelroots following the inverse scheme of the merging phase. This update phase was introduced in the previous 2D implementation of DCF

from [12]. This phase is made possible by successively storing all the boundary trees and the combined structures at each merging step. During the update, each combined boundary tree is used to update the *local* levelroots in the individual boundary trees. The latter are, in turn, propagated back and successively used to update the boundary trees merged during the previous steps. The update strategy is shown in the Algorithm 4. The procedure UPDATE takes three inputs: the two merged boundary trees, referred as a and b , and their merger c . We first iterate through all the nodes x in c that are not levelroots. These nodes were the *global* levelroots of a and b , but their parent-child relations have been corrected in later merging steps, such that they are not levelroots anymore. We first find the origin of x in a or b , referred as $origin_x$. This is possible thanks to the array $border_ori$ in each merged boundary tree, which keep tracks of the boundary tree origin of each node. Similarly, we find the origin of the new levelroot z of x in c , referred as $origin_z$, and compare it with the original levelroot of $origin_x$ in a and b (referred as $origin_x_lr$). If both are the same, no action is needed. If not, we check if the new levelroot belongs to the same boundary tree, and change the parent of $origin_x$ to the latter if they do. If $origin_x$ and $origin_z$ do not belong to the same boundary tree, but $origin_z$ has a levelroot pair such that there exists a *local* levelroot node in the boundary tree of $origin_x$ at the same level, the latter becomes the new parent of $origin_x$. If $origin_z$ does not belong to the boundary tree of x and does not have a levelroot pair, then $origin_z$ and $origin_x$ form a new levelroot pair, and their parent-child relation will be corrected in the second phase.

During the merging phase, we ensure that every node that is not levelroot in a or b points to a *local* levelroot in the same boundary tree. Hence, the second step of the update phase only consists in updating all the *local* levelroots of a and b . All the later can be found for any nodes in a and b using the function $B_SAME_LEVELROOT$ which returns their *local* levelroot. We define z , the *global* levelroot (which is either x and or the levelroot of x), and s , the corresponding node in c if it exists, using the array $border_idx$ which returns the index of z in c . x and all its ancestors are corrected using the procedure $update_branch$. This updates the attribute of x using the most recent attribute in c , and re-defines the parent-child relations by comparing the most recent parents of s in c . If the latter belongs to the same boundary tree of x , or if it has a levelroot pair, the new parent of x becomes the corresponding node in its boundary tree. However, if the parent of s has no equivalent in a or b , it is duplicated in the boundary tree and becomes the new parent of x . Hence, all the missing information is successively added in the individual trees, to ensure that the local boundary trees have the correct information of the connected components that belong to their respective tile. This requires a dynamic reallocation of the boundary trees a and b as we successively add missing nodes. All the nodes that have been updated during the procedure $update_branch$ are flagged using a *visited* array such that we avoid going through the same branch several times.

Algorithm 4 The Update Phase Procedure in DISCCOFAN

```

1: procedure UPDATE(BoundaryTree  $a$ , BoundaryTree  $b$ ,
   BoundaryTree  $c$ )
2:   for all nodes not levelroot in  $c$  do
3:      $x \leftarrow$  next node
4:      $z \leftarrow B\_LEVELROOT(x)$ 
5:      $origin\_x \leftarrow c.border\_ori(x)$ 
6:      $origin\_z \leftarrow c.border\_ori(z)$ 
7:      $origin\_x\_lr \leftarrow B\_LEVELROOT(origin\_x)$ 
8:     if  $origin\_x\_lr == origin\_z$  then
9:       continue
10:    end if
11:    if  $B\_TREE(origin\_x\_lr) == B\_TREE(origin\_z)$  then
12:       $B\_PARENT(origin\_x) \leftarrow origin\_z$ 
13:    else if  $B\_LR\_PAIR(origin\_z) \neq -1$  then
14:       $B\_PARENT(origin\_x) \leftarrow B\_LR\_PAIR(origin\_z)$ 
15:    else
16:       $B\_LR\_PAIR(origin\_z) \leftarrow origin\_x$ 
17:       $B\_LR\_PAIR(origin\_x) \leftarrow origin\_z$ 
18:    end if
19:  end for

20:  for all nodes in  $a$  and  $b$  do
21:     $x \leftarrow B\_SAME\_LEVELROOT(\text{current node})$ 
22:    if  $visited(x)$  then
23:      continue
24:    end if
25:     $z \leftarrow B\_LEVELROOT(x)$ 
26:     $s \leftarrow c.border\_idx(z)$ 
27:     $UPDATE\_BRANCH(x, z, s)$ 
28:  end for
29: end procedure

```

The use of levelroot pairs, described in Section IV-B, leads to a reduction the memory footprint caused by the node duplication, because it ensures that we only duplicate nodes missing at certain threshold levels in the tree. It also guarantees path compression, because all nodes at a same intensity level point to the same *local* levelroot.

This procedure is repeated at each update step, until all individual boundary trees are updated. Finally, the updated information in the local boundary trees of each process is propagated to the local component trees. In [12], we showed the memory cost of storing the boundary trees and duplicating nodes was negligible for 2D images up to 16-bits per pixel (bpp) data sets. In Section V, we show that the impact increases at higher dynamic ranges, but is not prohibitive.

After the update phase, all individual local component trees hold all relevant information such that filtering or multi-scale analysis yields similar results as if one would have computed the full component tree of the dataset. DISCCOFAN is optimized to allow efficient implementation of several attribute functions, and incorporates various filtering strategies, including pattern spectra and DAP analysis, as before. We refer the reader to [12] for more details.

V. PERFORMANCE TESTING

In this section, we assess the performance of DISCCOFAN on 2D and 3D data sets. All the following tests have been run on the Peregrine cluster of the Center for Information Technology of the University of Groningen. This cluster contains 159 standard nodes, with 24 Intel Xeon 2.5 GHz cores and 128 GB of RAM each, and 7 high memory nodes with 48 Intel Xeon E7 4860v2 2.6 GHz cores and 1024 or 2048 GB memory. We used the high-memory nodes, because our data sets are too large to be handled on the standard nodes. To compensate for the low number of nodes, we use several number of cores by node as MPI processes. We checked with several tests that this has no significant impact on the timing reported. The libraries used in all tests were gcc/g++ 8.2.0, HDF5 1.10.5, OpenMPI 3.1.3. All codes were compiled with an optimization level O3. All timings reported are the minimum obtained out of 10 different runs.

A. 2D Random Images

We first test DISCCOFAN as a function of the dynamic range using 2D, random-valued images. For a given bit depth b , the values are uniformly distributed from 0 to $2^b - 1$. These are the worst possible cases, as there are possibly many spanning connected components. We generate images of 100000×100000 pixels (10 Gpixels) and compare the performance of DISCCOFAN, the 2D DCF implementation from [12] (later referred as DCF-2D), and the algorithm from [13] (GÖTZ-2D). As mentioned before, the latter code does not compute any attributes, and implementing connected filters and multi-scale analysis would require a significant addition. Hence, for a fair comparison of the three methods, we implemented two versions of DCF-2D and DISCCOFAN which only compute the parent-child relations without any attributes (referred as DCF-2D-PAR and DISCCOFAN-PAR, respectively). In this experiment, we compare the gain of using 2 processes compared to a sequential run.

The results are shown in Figure 2, where the top panel shows for each algorithm the processing rate, in megapixels per second, for 1 or 2 processes, as a function of the dynamic range, and the bottom panel details the corresponding speed-up. The top panel highlights that DISCCOFAN has the highest processing rates, except in the 8-bpp case, where DCF-2D performs slightly better. This is because the latter algorithm uses Salembier's algorithm [1] to handle 8-bit images, which shows better performance than our modified Teeninga & Wilkinson algorithm at this low dynamic range. For higher dynamic range, DISCCOFAN performs better, and using it in sequentially already performs better than using 2 processes with DCF-2D or GÖTZ-2D. Data are missing the 32-bit case for GÖTZ-2D and DCF-2D because they required more than 9 days of computation on the Peregrine cluster.

The bottom panel shows similar outcomes, highlighting the better performance of DISCCOFAN at higher dynamic range. The performance of GÖTZ-2D slowly decreases with the bit depth. The original DCF implementation has a speed-up ≥ 2 up to 20-bpp images but the latter drops after. Both algorithm can process dynamic ranges up to 24-bpp but fail after. By contrast, DISCCOFAN can handle random images

Fig. 2. Top: Processing rate using 1 or 2 processes, for each algorithm, in megapixels per second, as a function of the dynamic range of the 2D, 10 billion pixels, images filled with random values. Bottom: the corresponding speed-up.

up to 28-bpp, and is the only method to have a reported speed-up at 32-bit images, because its computation ended in a reasonable time scale (≈ 15 hours). From 12 to 20 bpp, DCF-2D shows better speed-up than DISCCOFAN. However, DISCCOFAN is 2 to 5 times faster for these cases. Hence, we show here that the improved merging approach discussed in Section IV-B helps to handle higher dynamic range images. Additionally, we show in the next section that it improves the performance when dealing with realistic data sets, for cases where the two other approaches fail.

In Figure 3, we detail the fraction of total execution time spent in the different processing steps of DISCCOFAN, as a function of the dynamic range of the random images. For this experiment, we compute the morphological area opening [17], and report the percentage of total time for the following steps: building the local max-trees, merging and combining the boundary trees, successively updating the trees, and applying the area opening filter. This figure shows that building the local tree is the most computationally expensive operation up to 24-bpp. At 28 and 32-bpp images, the merging of the boundary trees is respectively ≈ 40 and 90 % of the total time to process the data. As we mentioned before, this behavior is expected because the merging approach becomes more costly as these high dynamic ranges. In the future, this

Fig. 3. Percentage of the total execution time for each processing step as a function of the dynamic range of images filled with random values. The steps are: building local trees; merging and combining boundary trees; updating boundary trees and local component trees; and filtering the resulting DCF using an area opening.

caveat could be solved using GPUs to more efficiently handle these operations. Finally, Figure 3 highlights that the cost of the update stage is negligible (always $< 1\%$) and that the filtering stage remains always $\approx 10\%$ of the total run time.

B. 2D Realistic Data Sets

We now assess the performance on real 2D data sets. Similarly to [13], DISCCOFAN includes a multi-threading implementation of the local tree computation. We first compare the performance of the hybrid implementation of GÖTZ-2D and DISCCOFAN (we omit DCF-2D because it does not have multi-threading). Note that we test two versions of DISCCOFAN, one only computes the parent-child relations (referred as DISCCOFAN-PAR), while the second computes morphological area opening (referred as DISCCOFAN-AREA), hence includes attribute computation and filtering. We used the ESO image of the Milky Way taken at the Paranal Observatory in Chile using the Visible and Infrared Survey Telescope for Astronomy (VISTA) [25]. This image is 81503 by 108199 pixels (≈ 9 Gpx) and is shown in RGB in Figure 4. As in [24], we converted the three RGB channels into a luminance image, where the intensity of each pixel is given by $L = 0.2126R + 0.7152G + 0.0722B$. In order to test the performance as a function of the dynamic range, we used two different quantization versions: an 8-bit (unsigned) image and the floating point luminance channel. Because the luminance channel of the ESO image is a linear combination of three 8-bpp channels, it is effectively a 24-bpp data set.

We follow the approach of [13]: The number of threads (N_t) and MPI processes (N_p) is successively doubled, such that we first have $N_p = 1$ and $N_t = 1$, $N_p = 1$ and $N_t = 2$, $N_p = 2$ and $N_t = 2$, etc. Figure 5 shows the comparison for the 8-bit case. DISCCOFAN-PAR is faster and scales almost linearly compared to GÖTZ-2D in this low dynamic range case. Nevertheless, this comes with a moderate increase in the maximum memory use. Overall, this excess vanishes with the use of several MPI processes. As expected, DISCCOFAN-AREA is slightly

Fig. 4. Images from the test data used.

more computationally expensive in time and memory than DISCCOFAN-PAR because it includes attribute computation and filtering. However, it remains faster than GÖTZ-2D, and scales the same way as DISCCOFAN-PAR.

Figure 6 shows similar outcomes for the floating point luminance channel. DISCCOFAN is faster, scales better, but uses more memory. This memory load should not be prohibitive because we aim to using several MPI nodes. Because we are dealing with higher dynamic range, the speed-up does not scale as linearly as before, and tends to flatten after 16 processes. One can note that using more than 4 threads does not improve the performance. This is because both shared-memory implementations are adapted from [10], hence their performance is limited at this dynamic range.

Figure 7 compares the speed-up of DISCCOFAN and GÖTZ-2D on the ESO floating point luminance data, in a fully distributed manner. It shows that the distributed approach of DISCCOFAN deals better with these high dynamic range cases, as we obtained almost linear speed-up up to 64 processes, while GÖTZ-2D reaches a maximum speed-up of 3 at 8 processes, and its performance decreases after.

As mentioned above, the luminance channel of the ESO image is effectively a 24-bpp data set. Therefore, we included a second astronomical data set from the Fornax Deep Survey (FDS) [26] with real, single precision, floating point values. We combined 4 *i-band* observation of the central field into a 83991 by 83990 pixels image (≈ 7 billion pixels), and one section of it is shown in Figure 4 (b). This data set is composed of point sources, and more extended structures in a fairly dark, smooth background. However, because of instrumental effects such as noise, the number of intensity levels is close to the size of the image itself. We compare the performance of GÖTZ-2D, DCF-2D-PAR, and DISCCOFAN-PAR in Figure 8. The execution time of GÖTZ-2D and DCF-2D increases with the number of processes, and suggests that

Fig. 5. Hybrid performance of DISCCOFAN-PAR, DISCCOFAN-AREA and GÖTZ-2D on the 2D 8-bpp quantization of the ESO luminance channel (≈ 9 Gpx). The size of the tiles decreases as the number of MPI processes increases. DISCCOFAN has a small memory overhead, but is faster and scales almost linearly up to 64 processes.

Fig. 6. Same as Figure 5 for the 2D, single-precision floating point, ESO luminance channel (≈ 9 Gpx). Similarly to the 8-bpp case, DISCCOFAN has a small memory overhead, but is faster and scales better than GÖTZ-2D.

Fig. 7. Speed-up of DISCCOFAN-AREA and GÖTZ-2D on the 2D, floating point, ESO luminance channel (≈ 9 Gpx) when using only MPI processes without threads.

these approaches fail to process real floating point data sets. Note that using more than 4 processes with GÖTZ-2D method returns an MPI error that we have not been able to solve. Hence we do not report timing values for $N_p > 4$, but the trend up to 4 processes is already explicit. We obtained a maximum speed-up of ≈ 1.5 for 4 MPI processes with DISCCOFAN-PAR. This is relatively modest, but DISCCOFAN is the only method for which using multiple MPI processes improves the sequential case. Additionally, the right panel of

Figure 8 shows that the memory usage decreases with the number of processes. Hence, DISCCOFAN shows promise to lower the memory footprint for processing Tera-Scale floating point data sets using multiple nodes.

C. 3D Data Sets

We assess DISCCOFAN's ability to handle 3D data. We use a different approach than in Section V-B. Rather than cutting the images in several tiles as we increase the number of processes, we keep constant the size of the tiles, such that the size of the volume processed increases with the number of processes. We performed this experiment on a data set that consists of a 6 hours observation collected with the LOFAR¹ radio-telescope. In total, the observation is made of 300 frequency slices of 15000×15000 single precision floating point pixels each. A single frequency slice is shown in Figure 4 (c). We fixed to 15 the number of frequency slices that a single node can process, such that the characteristic 3D volume tile is $15000 \times 15000 \times 15$ pixels (3.375 Gpx, not accounting for the overlapping pixels). For N_p MPI nodes, the size of the whole data set processed is $15000 \times 15000 \times 15 \times N_p$.

Figure 9 shows the results of this experiment. The speed-up is computed as $N_p \times t(1)/t(N_p)$ where $t(1)$ is the time obtained on 1 process. The memory gain is computed as $N_p \times m(1)/m(N_p)$ where $m(1)$ is the maximum memory allocated

¹Low-Frequency Array, <http://www.lofar.org>

Fig. 8. Distributed memory performance of DISCCOFAN-PAR, DCF-2D and GÖTZ-2D on the 2D, real single precision floating point, FDS data (≈ 7 Gpx). The size of the tiles decreases as the number of MPI processes increases.

Fig. 9. Execution time (left), speed-up (middle) and memory gain (right) on the LOFAR 3D observation, with single precision floating point values. The 3D tile size remains constant ($15000 \times 15000 \times 15$ pixels), such that the volume size increases linearly with the number of processes used. With 48 processes, the total volume processed is 162 Gvoxels.

when using 1 process. The latter measures the method's ability in scaling down the memory usage per node. Due to hardware limitations, we could only use at maximum 48 processes, which corresponds to an image cube of 162 Gigapixels. To our knowledge, this is the largest data set ever handled using component trees. There is a slight increase in execution time with the number of process, such that the speed-up corresponds roughly to $0.5 \cdot Np$ up to 16 processes, and flattens after. The processing rate goes from 1.63 to 18.3 Mpx per second when going from 1 to 48 processes. The memory gain is almost linear up to 4 processes, and slightly decreases after, up to 16 for 32 processes. Hence, despite of the non negligible size of the boundary trees when dealing with high dynamic range data sets, the results show that this approach still reduces the memory footprint over several nodes. This supports that DISCCOFAN can handle 3D Tera-scale real floating-point image cubes in a distributed manner.

VI. CONCLUSION

In this work, we presented DISCCOFAN,² a new method suitable to perform attribute filtering and multi-scale operations on Tera-Scale 2D and 3D images. This method is inspired by the *Distributed Component Forests* algorithm developed in our group [11], [12]. Here, we extended and improved the previous approaches, by adapting the recent flooding techniques to build the components trees [22], and by implementing a new, optimal merging approach to extend the computation of the Distributed Component Forest to Tera-Scale data sets with high

dynamic ranges. Overall, DISCCOFAN uses shared-memory and distributed-memory techniques to efficiently build max and min-trees and perform attribute filtering or multi-scale operation using any attribute function and several pruning strategies.

We reported in Section V-B that DISCCOFAN is always faster than currently available 2D distributed algorithms from [12], [13] when using only MPI processes or when combining MPI nodes and threads. Especially, DISCCOFAN is the only method able to improve the execution time of the parallel processing of a 2D real astronomical floating point image. Additionally, in Section V-C, we used DISCCOFAN to compute the parent-child relations on a 162 Gpx LOFAR observation with single precision floating-point and obtained a speed-up of 11.2 while reducing the theoretical memory footprint by 22 using 48 MPI processes. Thus, DISCCOFAN is the first approach suitable to process real Tera-Scale 2D and 3D data sets up to single precision floating-point complexity. Nevertheless, we report that DISCCOFAN uses slightly more memory compared to the approach from [13], but we assume that this excess is not prohibitive as it scales down with the number of processes. Because the merging strategy becomes computationally costly as we increase the bit depth, this approach is unlikely to handle extreme dynamic range data sets (64 bits or double precision floating point value). Hence, new approaches are needed, and one could think of extending the extreme dynamic range merging strategy from Moschini *et al.* [24] to distributed memory machines. Additionally, our algorithm focuses on a CPU-based parallelization while the use of a

²The code is available at <https://github.com/sgazagnes/disccofan>

GPU-based parallelization has become a standard in several applications (e.g. with deep learning [27]) as GPUs more efficiently handle parallel operations. We plan to explore this possibility to extend our algorithm using GPUs to improve its efficiency.

Finally, the method discussed in this work could be extended to work with *Tree-Based Shape-Spaces* which are particularly promising for object extraction [9].

ACKNOWLEDGMENT

The authors would like to thank the referees for their constructive comments that helped improve the quality of this manuscript and the Center for Information Technology of the University of Groningen for their support and for providing access to the Peregrine high performance computing cluster.

REFERENCES

- [1] P. Salembier, A. Oliveras, and L. Garrido, "Antiextensive connected operators for image and sequence processing," *IEEE Trans. Image Process.*, vol. 7, no. 4, pp. 555–570, Apr. 1998.
- [2] M. Wilkinson, M. Pesaresi, and G. Ouzounis, "An efficient parallel algorithm for multi-scale analysis of connected components in gigapixel images," *ISPRS Int. J. Geo-Inf.*, vol. 5, no. 3, p. 22, Feb. 2016.
- [3] P. Teeninga, U. Moschini, S. C. Trager, and M. H. F. Wilkinson, "Statistical attribute filtering to detect faint extended astronomical sources," *Math. Morphol.-Theory Appl.*, vol. 1, no. 1, p. 100, Jan. 2016.
- [4] C. Berger, T. Geraud, R. Levillain, N. Widynski, A. Baillard, and E. Bertin, "Effective component tree computation with application to pattern recognition in astronomical imaging," in *Proc. IEEE Int. Conf. Image Process.*, San Antonio, TX, USA, vol. 4, Sep./Oct. 2007, pp. 41–44.
- [5] C. Haigh *et al.*, "Optimising and comparing source extraction tools using objective segmentation quality criteria," Sep. 2020, *arXiv:2009.07586*. [Online]. Available: <https://arxiv.org/abs/2009.07586>
- [6] M. H. F. Wilkinson, P. Soille, M. Pesaresi, and G. K. Ouzounis, "Concurrent computation of differential morphological profiles on gigapixel images," in *Proc. Int. Symp. Math. Morphol. (ISMM)*, in Lecture Notes in Computer Science, vol. 6671, 2011, pp. 331–342.
- [7] G. Cavallaro, M. D. Mura, E. Carlinet, T. Geraud, N. Falco, and J. A. Benediktsson, "Region-based classification of remote sensing images with the morphological tree of shapes," in *Proc. IEEE Int. Geosci. Remote Sens. Symp. (IGARSS)*, Jul. 2016, pp. 5087–5090.
- [8] M. A. Westenberg, J. B. T. M. Roerdink, and M. H. F. Wilkinson, "Volumetric attribute filtering and interactive visualization using the max-tree representation," *IEEE Trans. Image Process.*, vol. 16, no. 12, pp. 2943–2952, Dec. 2007.
- [9] Y. Xu, T. Geraud, and L. Najman, "Connected filtering on tree-based shape-spaces," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 38, no. 6, pp. 1126–1140, Jun. 2016.
- [10] M. H. F. Wilkinson, H. Gao, W. H. Hesselink, J. E. Jonker, and A. Meijster, "Concurrent computation of attribute filters using shared memory parallel machines," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 30, no. 10, pp. 1800–1813, Aug. 2008.
- [11] J. J. Kazemier, G. K. Ouzounis, and M. H. F. Wilkinson, "Connected morphological attribute filters on distributed memory parallel machines," in *Proc. Int. Symp. Math. Morphol. (ISMM)*. Cham, Switzerland: Springer, 2017, pp. 357–368.
- [12] S. Gazagnes and M. H. F. Wilkinson, "Distributed component forests in 2-D: Hierarchical image representations suitable for tera-scale images," *Int. J. Pattern Recognit. Artif. Intell.*, vol. 33, no. 11, Oct. 2019, Art. no. 1940012, doi: [10.1142/S0218001419400123](https://doi.org/10.1142/S0218001419400123).
- [13] M. Gotz, G. Cavallaro, T. Geraud, M. Book, and M. Riedel, "Parallel computation of component trees on distributed memory machines," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 11, pp. 2582–2598, Nov. 2018.
- [14] E. J. Breen and R. Jones, "Attribute openings, thinnings, and granulometries," *Comput. Vis. Image Understand.*, vol. 64, no. 3, pp. 377–389, Nov. 1996.
- [15] P. Maragos, "Pattern spectrum and multiscale shape representation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 11, no. 7, pp. 701–715, Jul. 1989.
- [16] J. A. Benediktsson, M. Pesaresi, and K. Arnason, "Classification and feature extraction for remote sensing images from urban areas based on morphological transformations," *IEEE Trans. Geosci. Remote Sens.*, vol. 41, no. 9, pp. 1940–1949, Sep. 2003.
- [17] L. Vincent, "Morphological area openings and closings for grey-scale images," in *Shape in Picture: Mathematical Description of Shape in Grey-Level Images*, Y.-L. O, A. Toet, D. Foster, H. J. A. M. Heijmans, and P. Meer, Eds. Brussels, Belgium: NATO, 1993, pp. 197–208.
- [18] L. Najman and M. Couprie, "Building the component tree in quasi-linear time," *IEEE Trans. Image Process.*, vol. 15, no. 11, pp. 3531–3539, Nov. 2006.
- [19] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *J. ACM*, vol. 22, no. 2, pp. 215–225, Apr. 1975.
- [20] M. H. F. Wilkinson, "A fast component-tree algorithm for high dynamic-range images and second generation connectivity," in *Proc. 18th IEEE Int. Conf. Image Process.*, Sep. 2011, pp. 1041–1044.
- [21] E. Carlinet and T. Geraud, "A comparative review of component tree computation algorithms," *IEEE Trans. Image Process.*, vol. 23, no. 9, pp. 3885–3895, Sep. 2014.
- [22] P. Teeninga and M. H. Wilkinson, "Fast and memory efficient sequential max-tree construction using a trie-based priority queue," *Pattern Recognit. Lett.*, 2020.
- [23] E. R. Urbach, J. B. T. M. Roerdink, and M. H. F. Wilkinson, "Connected shape-size pattern spectra for rotation and scale-invariant classification of gray-scale images," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 2, pp. 272–285, Feb. 2007.
- [24] U. Moschini, A. Meijster, and M. H. F. Wilkinson, "A hybrid shared-memory parallel max-tree algorithm for extreme dynamic-range images," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 40, no. 3, pp. 513–526, Mar. 2018.
- [25] R. K. Saito *et al.*, "Milky way demographics with the VVV survey: I. The 84-million star colour-magnitude diagram of the galactic bulge," *Astron. Astrophys.*, vol. 544, p. A147, Aug. 2012.
- [26] E. Iodice *et al.*, "The fornax deep survey with VST. I. The extended and diffuse stellar halo of NGC 1399 out to 192 kpc," *Astrophys. J.*, vol. 820, no. 1, p. 42, Mar. 2016.
- [27] S. Shi, Q. Wang, P. Xu, and X. Chu, "Benchmarking state-of-the-art deep learning software tools," in *Proc. 7th Int. Conf. Cloud Comput. Big Data (CCBD)*, Nov. 2016, pp. 99–104.

Simon Gazagnes received the dual M.S. degree in electrical engineering and signal and image processing from the National Institute of Applied Sciences of Lyon, France, in 2016, and the M.Sc. degree in astrophysics from the University of Toulouse, France, in 2017. He is currently pursuing the Ph.D. degree with the Centre for Data Science and Systems Complexity, University of Groningen, where he aims at developing efficient methods based on connected morphology to meet the needs of astrophysical and particle physics applications. He first worked on biological imaging with the MORPHEME Team, Sophia Antipolis, and also with the STARBURST Team, Geneva Observatory, to work on modeling and data analysis of galaxy spectroscopic observations.

Michael H. F. Wilkinson (Senior Member, IEEE) received the M.Sc. degree in astronomy from the Kapteyn Astronomical Institute, University of Groningen, in 1993 and the Ph.D. degree from the Institute of Mathematics and Computing Science, Groningen, in 1995. He worked on image analysis of intestinal bacteria with the Department of Medical Microbiology, University of Groningen. He was appointed as a Researcher at the Centre for High Performance Computing, Groningen, working on simulating the intestinal microbial ecosystem on parallel computers. After this, he worked as a Researcher on image analysis of diatoms with the Johann Bernoulli Institute for Mathematics and Computer Science. He is currently an Associate Professor with the Bernoulli Institute for Mathematics, Computer Science and Artificial Intelligence, working on morphological image analysis and especially connected morphology and models of perceptual grouping. An important research focus is on handling giga- and tera-scale images in remote sensing, astronomy, and other digital imaging modalities.