Parallel Computation of Component Trees on Distributed Memory Machines

Markus Gotz, Gabriele Cavallaro, Thierry Géraud, Matthias Book, Morris Riedel

Parallel Computation of Component Trees on Distributed Memory Machines

Markus Götz, Gabriele Cavallaro, *Member, IEEE*, Thierry Géraud, *Member, IEEE*, Matthias Book, and Morris Riedel, *Member, IEEE*

Abstract—Component trees are region-based representations that encode the inclusion relationship of the threshold sets of an image. These representations are one of the most promising strategies for the analysis and the interpretation of spatial information of complex scenes as they allow the simple and efficient implementation of connected filters. This work proposes a new efficient hybrid algorithm for the parallel computation of two particular component trees—the max- and min-tree—in shared and distributed memory environments. For the node-local computation a modified version of the flooding-based algorithm of Salembier is employed. A novel tuple-based merging scheme allows to merge the acquired partial images into a globally correct view. Using the proposed approach a speed-up of up to 44.88 using 128 processing cores on eight-bit gray-scale images could be achieved. This is more than a five-fold increase over the state-of-the-art shared-memory algorithm, while also requiring only one-thirty-second of the memory.

Index Terms—Component-Trees, Threshold Decomposition, Max-Tree, Connected Component Labeling, High-Performance Computing, Hybrid Application, MPI, Multithreading.

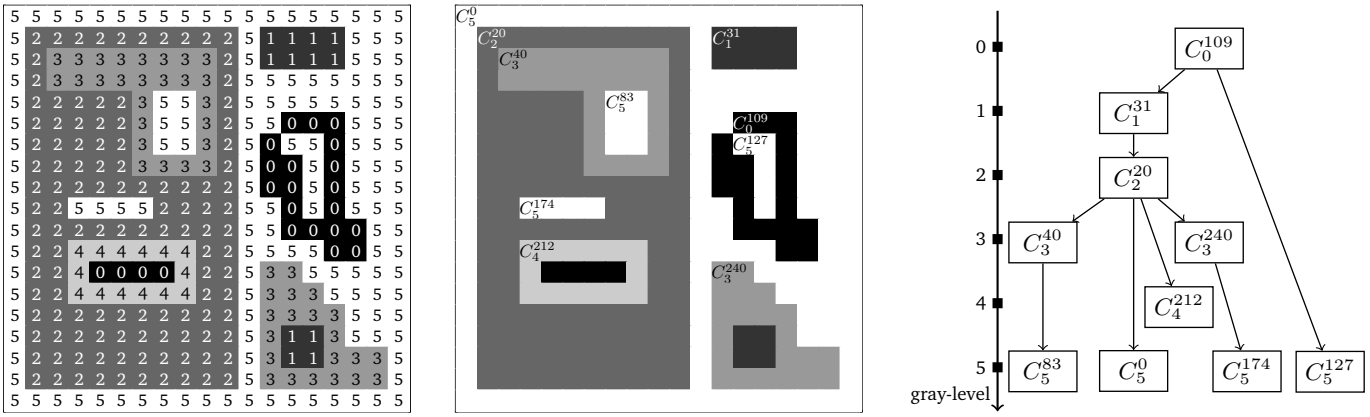
1 Introduction

Since the 1960s, mathematical morphology [1], [2] has become increasingly popular in the image processing community mainly due to its proven utility and rigorous mathematical description. The mathematical morphology framework provides a set of powerful operators for analyzing the spatial domain of images at the region-level—i.e., connected components—based on tree representations, called thresholds decompositions [3], [4]. These are based on tree representations of images which can be divided into two main groups [5]: hierarchies of segmentation—i.e., hierarchy of image partitions such as minimum spanning trees [6], alpha-trees [7], binary partition trees [8]—and threshold decompositions—i.e., hierarchy of regions such as component trees [4], [9], tree of shapes (ToS) [10] and multivariate tree of shapes [11]). Generally, tree structures are often considered richer in descriptive ability since they can be exploited for breaking down images into their fundamental elements which are easier to interpret with regards to the pixels. Component trees [4], [9], are thresholds decompositions that represent connected components [12] at every threshold level of an image in a hierarchical fashion, through parent relationships between nodes. The connected components organized in such trees can be filtered with different strategies [3], [4] and can model various types of connectivity [13].

Component trees (i.e., max- and min-tree) have been popularized by connected operators, such as attribute filters [2], [3], which have been extensively used for the modeling of spatial information in images from remote sensing [14], [15], astronomy [16], [17] and medical scanning [18], [19]. Attribute filters are edge-preserving and flexible operators due to the preservation of contours in the processed objects and rely on multiple spatial measures or attributes. The possibility to perform a multi-attribute analysis, like attribute filters built by employing different attributes, enriches the extraction of spatial arrangement and improves the discrimination between different structures. In the presence of scenes with high complexity and heterogeneity, e.g., densely populated urban area, a complete modeling of the spatial information can be achieved through a multi-level analysis. It implies the decomposition of the original gray-level image obtained by applying a sequence of attribute filters according to a set of filter thresholds [20]. The result of this operation are the so-called attribute profiles [14]. They have been exploited mainly in remote sensing, e.g., classification [21], [22], data fusion [23] and change detection [24], as well as in medical imaging processing for tomographic image segmentation [25]. Recently, attribute filters are utilized within a novel deep learning framework for the large-scale, unsupervised detection of objects in remote sensing image [26]. A set of attributes is automatically identified in order to extract a representative, high quality training data set.

Nowadays, image processing applications rely on very high resolution data due to the continuing technological improvements of the sensor instruments. For example earth observation platforms have led to the increasing volume, acquisition speed and variety of sensed images, e.g., the

- *M. Götz is with the Jülich Supercomputing Center, Wilhelm-Johnen-Straße 52428 Jülich, Germany, and the University of Iceland, 107 Reykjavik, Iceland.*
  E-mail: m.goetz@fz-juelich.de
- *G. Cavallaro is with the Jülich Supercomputing Center, Wilhelm-Johnen-Straße 52428 Jülich, Germany.*
  E-mail: g.cavallaro@fz-juelich.de
- *T. Géraud is with the EPITA Research and Development Laboratory (LRDE), Le Kremlin-Bicêtre, France.*
  E-mail: thierry.geraud@lrde.epita.fr
- *M. Book is with the University of Iceland, 107 Reykjavik, Iceland.*
  E-mail: book@hi.is
- *M. Riedel is with the Jülich Supercomputing Center, Wilhelm-Johnen-Straße 52428 Jülich, Germany, and the University of Iceland, 107 Reykjavik, Iceland.*
  E-mail: m.riedel@fz-juelich.de



(a) Original gray-scale image. (b) The iso-level, 4-connected regions. (c) The corresponding max-tree.

Fig. 1: Example of max-tree representation based on exemplary image and its components C_c^i, with the subscript c being the gray-level and the superscript i the canonical point uniquely identifying the component.

World-View-3 satellite sensor (spatial resolution of 0.31m), or the AISA Dual airborne sensor (500 bands with spectral resolution of 2.9nm). The performances of traditional serial and parallel algorithms for computing the component trees are strictly correlated to the size and the quantization of the data. The size of remote sensed images is usually in the order of several gigabytes due to the depiction of vast and complex scenes. Consequently, they can not be stored or processed by algorithms designed for a single shared-memory machine. Furthermore, due to the high sensitivity of the new sensors, e.g., radiometric resolution, these images are characterized by an ample domain of integers or floating point values, which directly affect the processing time.

In this paper, a novel shared- and distributed-memory hybrid algorithm for the efficient computation of exact component trees, specifically the min- and max-tree, of integral gray-scale and floating-point images, is presented. For this, the problem, i.e., the image is subdivided into equal-sized chunks that get assigned to all available distributing computing nodes. Then, each of the nodes computes a local, partial component tree of the assigned chunk. A modified version of the shared-memory parallelized, depth-first, flooding max-tree algorithm proposed by Ouzounis *et al.* [27] is employed. Finally, the obtained partial component trees need to be merged into a correct, monolithic global representation. This is achieved through the iterative resolution and rearrangement of the iso-level edges of the image chunks' boundary trees for each gray-level, marking the major algorithmic challenge. In the proposed approach, the level connections are expressed as tuples—a data structural design that has been used by Flick el al. [28] for the distributed resolution of genomic graphs. This is a novelty in the mathematical morphology framework and the distributed computation of component trees.

The remainder of this paper is organized as follows. Section 2 provides a brief introduction to component trees. The subsequent Section 3 presents an overview over existing algorithms proposed in the literature. In Section 4 the proposed algorithm for parallel and distributed computation of the component trees is laid out. Complexity considerations and implementation details are explained in Section 5. A study of the algorithms strong and weak scaling as well as comparative study to the current state-of-the-art algorithm is presented in the experimental evaluation in Section 6. Finally, Section 7 concludes the paper, discussing the findings of this work and presents opportunities for future work.

2 Component Trees

Component trees were introduced by Jones [9], [29] as efficient image representations that enable the computation of advanced morphological filters in a simple way. These trees are hierarchical structures that encode the threshold sets and their inclusion relationship. Thereby, each sub-tree is nothing different than all connected image components up until the given gray-threshold. As a result, one of the major advantages of components trees is their possibility to efficiently implement of connected filters.

More formally, let $f : \Omega \rightarrow E$ be a discrete two-dimensional gray-scale image, defined on a spatial domain $\Omega \subseteq \mathbb{Z}^2$ and taking values on a set of scalar values $E \subseteq \mathbb{Z}$. For any $\lambda \in \mathbb{Z}$, a lower $\mathcal{L}(f)$ and upper $\mathcal{U}(f)$ threshold set is defined by:

$$\mathcal{L}(f) = \{x \in \Omega, f(x) < \lambda\}, \tag{1}$$

$$\mathcal{U}(f) = \{x \in \Omega, f(x) > \lambda\}, \tag{2}$$

Let $\mathbb{P}(\Omega)$ be the power set of all the possible subsets of Ω. Given $X \in \Omega$, the set of connected components of X is denoted as $\mathcal{C}(X) \in \mathbb{P}(\Omega)$. Each connected component is represented by a unique point called the level root [30], or canonical element [16]. Considered two points $x, y \in \Omega$, and x_r the root of the tree, x is canonical if $x = x_r$ or $f(parent(x)) < f(x)$ (where parent is the image that encodes the inclusion relationship of the threshold sets [16])

If $\leq$ is a total relation, any two connected components $Y, Z \in \mathcal{C}(\mathcal{L}(f))$ are either disjointed or nested. The min-tree and max-tree structures represent the lower, respectively upper, threshold components in $\mathcal{L}(f)$ and $\mathcal{U}(f)$ as well as with their inclusion relations. For example, Fig. 1c shows the max-tree structure of the image in Fig. 1a. The arrows denote the parent relation between the nested connected

components that are identified in Fig. 1b. This is a simplified case that it is used for clarification purposes. In synthetic images that include more complex shapes (see the example in Section 4) or real scenarios the max-tree structure is less intuitive since its hierarchy is not driven by the inclusion relationship of connected components as it appears in Fig. 1. Furthermore, the notion of tuple as used in this work should be understood as a finite sequence of four elements of the form $\langle c_x, x, c_y, y\rangle$ with $x, y \in \Omega$ being two pixel coordinates and $c_x, c_y \in E \wedge f(x) = c_x \wedge f(y) = c_y$ the colors of these pixels.

3 Related Work

The selection of the most appropriate algorithm for computing the component trees shall be made according to the properties of the input image (i.e., size and pixel value quantization) and the processing resources available such as memory capacity and number of computing cores. Carlinet *et al.* [39] presented a comparative review of the state-of-the-art algorithms and provided detailed guidelines for selecting the most suitable algorithm. The algorithms are grouped into three main classes: immersion-, flooding- and merge-based. Algorithms that belong to the immersion and flooding class, may also be referred to as leaf-to-root merging and root-to-leaf flooding methods, respectively [37]. Since this section is not intended to repeat the review, Fig. 2 merely presents a timeline for each algorithm class, and how they have developed in the past years.

As explained in Section 1, the algorithm proposed in this work is of hybrid nature, entailing shared- and distributed-memory parallelization aspects simultaneously. The node-local parallelization is thereby based on a flooding strategy, while the distributed computation components follows a merge-based approach. Therefore, a detailed explanation of immersion algorithms is deliberately left out, and the reader is referred to Tarjan [31], Najman *et al.* [30], Berger *et al.* [16] and Carlinet *et al.* [39].

3.1 Flooding Algorithms

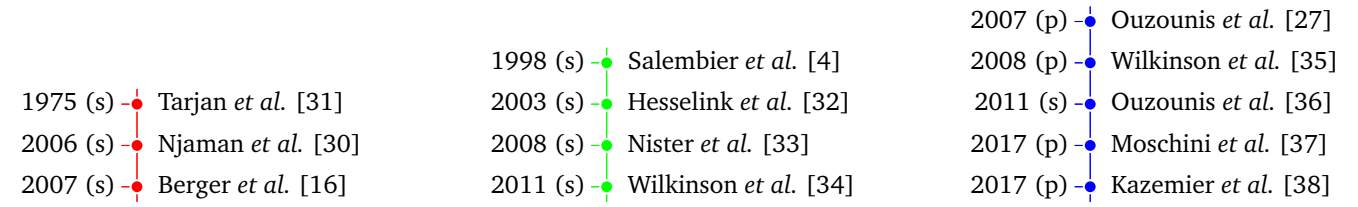
The first flooding algorithm was proposed by Salembier *et al.* [4]. It is an efficient algorithm which retrieves the pixel at the lowest gray-level, i.e., root, through a scanning step and then it performs a propagation by flooding the neighbor at the highest level, i.e., a depth-first traversal of the connected components at higher intensities. Pixels in the propagation front are stored in a hierarchical queue composed by as many First In First Out (FIFO) queues as the number of gray-levels. It allows to directly access any pixel in the FIFO queue at a given level. Salembier's *et al.* [4] algorithm was rewritten in a non-recursive implementation by Hesselink *et al.* [32], later also by Nister *et al.* [33] and Wilkinson *et al* [34]. The algorithm presented by Wilkinson aims at solving the limitation of Salembier, the linear scaling with the number of gray-levels. Wilkinson has proposed to use a priority queue and a stack, a combination of the algorithms of Salembier *et al.* and Hesselink *et al.*, instead of using only a hierarchical queue for handling the pixel values during the flooding.

Carlinet *et al.* [39] have proposed a non-recursive flooding algorithm variant of Salembier *et al.*, which has strong similarities with Wilkinson *et al.* and Nister *et al.* Due to the fact that the algorithm proposed in this work is based on it, the pseudo-code (40 lines only) is shown in Algorithm 1. The algorithm computes two structures that describe the tree: a parenthood image, such that $parent(p)$ is the parent pixel of pixel p in the tree, and an array of pixels S, where pixels are sorted such as the parent of any pixel is always stored before this pixel (so browsing S corresponds to a downward traversal of the tree). To that aim, two auxiliary structures are used: $pqueue$ is a hierarchical queue of pixels, and $levroot$ is a stack of pixels. The algorithm is divided into three stages: initialization, flooding and root fixing, respectively starting from lines 13, 19 and 32. In the initialization phase,

Algorithm 1 Non-recursive version of Salembier's algorithm as presented by Carlinet *et al.* [39].

```
1: procedure PROCESS-STACK(r, q)
2:     λ ← f(q)
3:     POP(levroot)
4:     while levroot not empty and λ<f(TOP(levroot)) do
5:         INSERT_FRONT(S, r)
6:         r ← parent(r) ←POP(levroot)
7:     if levroot empty or f(TOP(levroot)) ≠ λ then
8:         PUSH(levroot, q)
9:     ▷ Particular case of the last element:
10:    parent(r) ← TOP(levroot)
11:    INSERT_FRONT(S, r)

12: function MAX-TREE(f)
13:    ▷ 1. INITIALIZATION:
14:    for all p do parent(p) ← −1        ▷ meaning "unseen"
15:    start_pixel ← any point in Ω
16:    PUSH(pqueue, start_pixel)
17:    PUSH(levroot, start_pixel)
18:    parent(start_pixel) ← INQUEUE
19:    ▷ 2. FLOODING:
20:    loop
21:        flood:
22:        p ← TOP(pqueue); r ← TOP(levroot)
23:        for all n ∈ N(p) such that parent(p) = −1 do
24:            PUSH(pqueue, n)
25:            parent(n) ← INQUEUE
26:            if f(p) < f(n) then
27:                PUSH(levroot, n)
28:                goto flood
                                          ▷ p is done
29:        POP(pqueue)
30:        parent(p) ← r
31:        if p ≠ r then INSERT_FRONT(S, p)
32:    ▷ 3. ROOT FIXING:
33:    while pqueue not empty do
34:        ▷ all points at current level done?
35:        q ←TOP(pqueue)
36:        ▷ Attach r to its parent
37:        if f(q) ≠ f(r) then PROCESS-STACK(r, q)
38:    ▷ Particular case of the last element, the tree root:
39:    root ← POP(levroot)
40:    INSERT_FRONT(S, root)
```



(a) Immersion (leaf-to-root merging)

(b) Flooding (root-to-leaf flooding)

(c) Merge-based

Fig. 2: The main three classes of algorithms and their chronology. Each entry includes the publication year, the algorithm class (s - serial or p - parallel) and the corresponding reference.

a random point $start\_pixel$ is chosen as the flooding point. This pixel is now considered as a canonical element, i.e., the representative of the connected component, and it is pushed on the stack $levroot$. This stack stores the representative pixels of the visited components; these pixels are the roots of the sub-trees, so they represent components at different gray-levels. The main purpose of the flooding phase is to compare the gray-level of each pixel p with its neighboring pixels n, and to enqueue those that have not yet been seen. The first processed pixels are p and the canonical element r of its component. These have the highest priority in the queue, i.e., the highest gray-level, and are on top of $levroot$ (p is not removed from the queue). The neighboring pixels n are pushed on the stack only if $f(n) > f(p)$, line 24, which immediately triggers a jump (**goto**) to the `flood` label (in line 21). This jump thus emulates a recursive call, which actually corresponds to a *depth-first* discovery of the tree. At a certain point, all the neighboring pixels of p will be either in the queue or already processed, meaning that the analysis of p has terminated; we cannot progress deeper in the tree. Following this, p is removed from the queue (line 29), $parent(p)$ is set to r, i.e., the canonical element. In order to ensure that the canonical element will be the last one inserted, p is added to S when $r \neq p$ (line 31). After p is removed from the queue, the canonical element r is attached to its parent only when the level component has been fully processed (line 30). The last step (startging from line 32) aims at setting the parenthood relationship between components. The first element q of $pqueue$ is retrieved, and the PROCESS-STACK procedure is called (line 37) when q has a different level than p. It pops the stack, sets the parent relationship between the canonical elements, and inserts them in S until the top component has a level no greater than $f(q)$—lines 3 to 6. When the stack gets empty or the top level is lower than $f(q)$, then q is pushed on the stack as the canonical element of a new component—lines 7 and 8. The top element of the stack is the current root pixel—linelst:last and following. The algorithm ends when all points in queue have been processed, then S only misses the root of the tree, which is the single element that remains on the stack (line 38 and following).

3.2 Merge-based Algorithms

The natural way to implement a parallel algorithm is to divide the original image domain and compute the max-tree on each sub-image using any algorithm from Fig. 2. In order to compute this partition, the image should be split in Np connected disjoint regions, which is the union that forms the entire image domain. During this step, the image is split into a reasonable number of chunks, which reflects the underlining processing architecture, e.g., number of threads available. For instance, when the number of image chunks is lower than the number of threads the domain is not decomposed enough and the distribution of the computations is not yet optimal (load imbalance). Some threads will idle while having to wait for other threads to finish. Once all sub-trees are generated, they can be merged into a single global tree as proposed by Matas *et al.* [40], Wilkinson *et al.* [35] and Ouzounis *et al.* [27]. This is a non-trivial phase as it requires that the gray-levels of the connected components are merged and their parent relationships updated.

The merging strategy introduced by Matas *et al.* [40] follows the same principle used by Wilkinson. However, the algorithm starts by computing partial 1-D trees (i.e., a tree for each row of the image). Then, the trees that belong to neighboring rows are merged progressively until the global tree is obtained. The merging algorithm proposed by Wilkinson *et al.* [35] retrieves the global max-tree and its attribute values from multiple sub-trees that can be derived over any arbitrary image sections. Firstly, each thread computes a data structure (i.e., partial max-tree) that sets the parent pointers and accumulates attribute values. Afterward, all these sub-domains are merged through the use of a binary tree. This is achieved through a concurrent merging strategy which connects two partial max-trees step by step. A synchronized mechanism based on two binary semaphores defines when a thread is ready to accept the domain of its neighbor (i.e., the sender has to complete the max-tree computation). Once the last connection is computed by the thread 0, all the threads can resume and proceed with the filtering phase. Ouzounis *et al.* [27] proposed a max-tree algorithm implementation for attribute filtering based on the concurrent merging strategy in [35]. The hybrid algorithm proposed in this paper relies on the merging strategy used in [27], [35] and additional details can be found in Section 4.

3.3 Connected Component Labeling

The main problem of generating the max-tree can be split into two parts: find the connected components and establish their hierarchy. The task of grouping connected pixels within an image can be seen as the well-known Connected Component Labeling problem [41]. This is an important

step for a large number of applications and it relies firstly on finding which parts of an object, e.g., binary images, gray-levels images, data with higher dimensionality, etc., that are physically connected, based on a connectivity rule, and secondly, to label them. Iverson *et al.* [42] provide an evaluation of connected-component labeling algorithms in the context of distributed computing, when data that need to be processed in a given application usually require large processing power and distributed-memory machines. They conclude that there is an unavoidable compromise to find between memory and processing time. Most of the available parallel algorithms are problematic especially in terms of memory requirements. For instance, the merging step could end up on a single node resulting in an unbalanced scenario. However, algorithms that try to solve this problem provide poor scaling results in terms of processing time. At the same time Flick el al. [28] proposed a scalable distributed-memory algorithm to overcome this problems raised by Iverson. They have aimed to solve issues such as excessive memory usage, extra computation and communication of the processors, and load balancing. The main idea of the algorithm is similar to the Shiloach-Vishkin algorithm [43] in that it transforms the problem into finding weakly connected components within the Bruijn directed graph [44]. It will be shown in Section 4 that the proposed algorithm uses the notion of tuples and inverse doubling in order to connect the overlap zones between the split regions and resolves the connected components and their corresponding parenthood.

4 Distributed Component Tree Algorithm

In the following sections the algorithm for the parallel and distributed computation of the component trees will be introduced. To simplify the explanation, only the max-tree case will be presented. A corresponding min-tree algorithm can be inferred by reversing the order in which the gray-levels are processed. Furthermore, the explanations assume a homogeneous, distributed systems. This refers in particular to the workload distribution of the image into contiguous, equal-sized partitions. For heterogeneous, distributed systems a different, more appropriate strategy must be chosen, such as proposed by Qin *et al.* [45].

4.1 Definitions and Notation

A two-dimensional gray-scale image f can be seen as an undirected graph $\mathcal{G} = (V, E)$. V represents a set of vertices—the pixels of the image—and $n = |V|$ the total number of pixels. Then E, a number of edges or non-ordered pairs of vertices (v_i, v_j), with $i, j \in [0, n[$, which model the neighborhood relationship of the pixels. Classically, images are either four- or eight-connected [46], meaning the top, left, right and bottom neighbors, respectively including the diagonals, are considered connected neighbors. The entire graph $\mathcal{G}$ is said to be connected if, for any $p, q \in V$, there exists a path from p to q, which is a sequence of $s > 1$ vertices—i.e., $p = p_1, ..., p_s = q$—such that every $p_i \in V$, and any two successive pixels of the sequence are adjacent $e_{p_i, p_{i+1}} \in E$. Given this definition a connected component $\mathcal{CC}$ is a subgraph of $\mathcal{G}$ such that $V_{\mathcal{CC}} \subseteq V_{\mathcal{G}}, E_{\mathcal{CC}} \subseteq E_{\mathcal{G}}$, $\forall\, p \in V_{\mathcal{CC}} : f(p) = c$, that is maximal: $\forall\, e = (p, p') \notin E_{\mathcal{CC}}$ such that $p \in V_{\mathcal{CC}}$ and $p' \notin V_{\mathcal{CC}}$, $f(p') \neq c$. A connected component can be either weak or strong connected, depending on the path length s. Weak connected graphs can have an arbitrary path length, while for strong connected graphs $s = 2$ holds. Furthermore, if not stated otherwise, the following symbols are defined for the remainder of the document: h and w is the height and the width of the image f, respectively. The entire image has a gray-level depth d, i.e., the number of different gray-values c. Concerning parallelization, the number of available distributed compute nodes is p, while the local number of shared-memory threads is labeled with t. For the explanation of the distributed resolution, it is also necessary to introduce what is coined a tuples. These are essentially quartuples, mathematical tuples with four components, of the form $\langle c_i, p_k, c_j, p_l \rangle$ with c_i and c_j being two gray-values and p_k and p_l two vertices. They are used to explicitly express edges $e \in E$ of the image f of a canonical point with a certain gray-value to a different canonical point of a given other gray-value.

4.2 Concept

Algorithm 2 Pseudo-code of the proposed distributed max-tree algorithm.

```
1:  @parallel
2:  function DISTRIBUTED-MAX-TREE(f)
3:      p ← number of nodes
4:      r ← processor id in range [0, p[
5:      t ← number of threads
6:      f' ← LOAD-PARTIAL-IMAGE(f, r, p)
7:
8:      parents' ← LOCAL-MAX-TREE(f', t)
9:
10:     root_tuples ← HALO-TREE-EDGES(f', parents')
11:     area_tuples ← HALO-COMPONENTS(f', parents')
12:     tuples ← RESOLVE(area_tuples, root_tuples)
13:
14:     tuples' ← REDISTRIBUTE(tuples)
15:     parents ← APPLY(tuples', parents')
16:
17:     return parents
```

The general nature of the distributed max-tree algorithm can be described as divide-and-conquer. This means, that the entire problem, i.e., the image, is divided into sub-images for which the respective max-trees are computed and that are then successively merged along the division boundaries. The major algorithmic challenge lies in the latter stage. It requires to solve two demanding graph theory sub-problems—connected component labeling and graph canonicalization—in distributed memory environments. This work proposes an iterative, parallel merging algorithm based on explicit expression of the max-tree edges as directed tuples. For this, halo-zones are employed—one-pixel wide, redundant overlaps of the partial images of neighboring image chunks (see also Figure 3). Conflicts within the tuples signify the need to rearrange the edges of the boundary tree in order to obtain a globally correct view. Such a conflict could for example be, that the exact same gray-level component points to different parent components in the partial trees.

Analogous to the two graph problems to be solved, the proposed algorithm requires two kinds of tuples. First, the locally determinable edges of the boundary trees in the halo zone ($root\_tuples$), and, second, information about components that have been split due to the image division ($area\_tuples$) and their the canonical points. In the proposed resolution approach, the tuples are iteratively scanned for and remapped to the most optimal candidates. This entails merging split iso-level and determining the best parent or root each component and as a result remapping the tuples. Less optimal tuple candidates are replaced by transient edges to ensure a correct merge of the remaining trees. Subsequently, the computed changes need to be applied to the partial images. For this, the resolved tuples are send back to the respective sub-images of origin, and utilized to obtain the globally create max-tree. Algorithm 2 sketches the proposed strategy.

4.3 Local Max-Tree Algorithm

For the local computation one can in principle employ any correct max-tree algorithm. The proposed solution specifically utilizes a modified version of the recursive Salembier's depth-first flooding algorithm [4]. As observed by Carlinet *et al.* [39], Salembier's algorithm was rewritten in a nonrecursive implementation by Hesselink *et al.* [32] and later by Nister et al. [33] and Wilkinson *et al.* [34]. In [34], it was shown that replacing the hierarchical queue (used in [4]) by a priority queue to perform the recursive flooding strongly reduced the computation of component trees, especially for high-dynamic range images (i.e., with floating point). Due to the similarities of approach and the proved efficiency in terms of processing time in [34] and [33], Carlinet *et al.* [39] merged these solutions and suggested a novel non-recursive implementation, which is utilized in this work. Additionally, it has been enhanced to always use the minimal pixel index as canonical point for an iso-level and to yield better computational performance. Algorithm 3 presents the corresponding pseudo-code.

The first change can best be seen in line 26. In contrast to the original non-recursive variant, the canonical area point is not chosen at the beginning of an iso-level processing—as it may not yet be the canonical minimum—but rather constantly maintained throughout the process. This is done by keeping the current area minimum at a specific place, e.g., the front of the pixel vector, and compared to on insertion of new elements. Only after all pixels of the entire iso-level is found, the canonical point is assigned in the parent image, see also line 31, and thus minimality of the index guaranteed.

Moreover, when one considers the computational performance of the algorithm, the proposed modifications also allows for faster computation. Before, each gray-level had its one hierarchical queue in Salembier's original algorithm formulation or a singular in Carlinet's non-recursive reformulation. This approach scales logarithmically with both, the number of gray-levels as well as the number of pixels per channel. In the proposed variant the gray-levels are keys to a map, called $stacks$, that has vectors for the corresponding pixels as keys. Then, insertions only scale logarithmically with the number of gray-levels, for locating the respective vector in the map, but the actual push operation happens in constant

Algorithm 3 Pseudo-code of the modified version of Salembier's depth-first, flooding-based max-tree algorithm.

```
1:  function MAX-TREE(f)
2:      stacks ← {}                                   ▷ Initialization
3:      pixels ← {}
4:      children ← [ ]
5:      for all p ∈ f do
6:          parents(p) ← −1
7:          deja_vu(p) ← false
8:
9:      start_pixel ← any index in f                  ▷ Seed pixel
10:     start_grayv ← f(start_pixel)
11:     deja_vu(start_pixel) ← true
12:     PUSH(stacks(start_grayv), start_pixel)
13:     PUSH(pixels(start_grayv), start_pixel)
14:
15:     while not EMPTY(stacks) do                    ▷ Depth-first
16:         flood:
17:         grayv ← MAX-KEY(stacks)
18:         pixel ← POP(stacks(grayv))
19:         for all n ∈ N(p) do
20:             if deja_vu(n) then continue
21:             deja_vu(n) ← true
22:             PUSH(stacks(f(n)), n)
23:             PUSH(pixels(f(n)), n)
24:             if TOP(stacks) > BACK(stack) then
25:                 ▷ Ensure canonical point is in front
26:                 SWAP(TOP(stacks), BACK(stacks))
27:             if grayv < f(n) then
28:                 PUSH(stacks(grayv), pixel)
29:                 goto flood
30:
31:         if EMPTY(stacks(grayv)) then              ▷ Iso-level done
32:             c ← pixels(grayv)
33:             for all p ∈ pixels(grayv) do parents(p) ← c
34:             ▷ Remove the iso-level from the maps
35:             ERASE(pixels, grayv)
36:             ERASE(stacks, grayv)
37:
38:             if EMPTY(stacks) then                 ▷ Attach children
39:                 merge ← MAX-KEY(stacks)
40:             else
41:                 merge ← grayv
42:             while not EMPTY(children) and
                    ↪ BACK(children).grayv > merge do
43:                 child ← POP(children)
44:                 parents(p) ← child.pixel
45:             PUSH(children, ⟨merge, c⟩)
46:
47:     if not EMPTY(children) then                   ▷ Attach root children
48:         root ← BACK(children).grayv
49:         for all c ∈ children do parents(c) ← root
50:
51:     return parents
```

time. Especially in images with a large amount of pixels, this can drastically reduce computation time.

The strategy for shared-memory parallelization explained by Ouzounis *et al.* [27] has been chosen. Their max-tree algorithm was based on the parallel implementation proposed by Wilkinson *et al.* [35]. The authors largely proved the correctness and efficiency of the approach and they predicted an achievable (near) linear speed-up beyond 4 CPUs. In greater detail, the local image partition of the node distribution step, is again virtually partitioned in t equally sized horizontal chunks, without overlap, and assigned to one of the t threads. For each of the partitions the partial max-tree is computed using the introduced algorithm. The virtual images boundaries are realized by excluding respective pixels in the neighborhood searches. Finally, the partial max-trees are merged using the `connect` function, equally presented by Ouzounis *et al* [27]. Minor changes have been made to ensure that the canonical points of each iso-level is guaranteed to be minimal. This can be achieved by performing a look-ahead on the upcoming elements of the merge stacks and potentially swap them if required. After the local computation, one would obtain the partial max-trees depicted in Fig. 3e and Fig. 3f.

4.4 Tuple Generation

The tuple generation of the partial images is subdivided into two major steps. First, there is the generation of root tuples, representing the branches of the halo areas' partial max-trees, and, second, the area tuples—i.e., tuples that connect or stitch the gray-level components divided by the image partition.

The former set of tuples is generated by traversing the boundary max-trees recursively upwards towards the root. For each edge within the tree a corresponding tuple is created with the canonical point of a sub-ordered, child iso-level pointing to the canonical point of its parent. In line with the used notation it is the result of a HALO-TREE-EDGES function invocation. The performance of this step can be further optimized by skipping already visited branches also avoiding redundant tuples. Generating the root tuples can be performed fully local and does not require any data exchange with other processing nodes.

The former step, the connection of the split components, is achieved by identifying differences in the labeling of the canonical points of said components in the halo areas. It is the result of a HALO-COMPONENTS function invocation. This can achieved efficiently by performing two prefix-sums pixels in the halo area of the partial images. In the first run, the canonical points are broadcast downwards across the partial images, while the second prefix-sum, in reverse, back-propagates found differences. Each conflict implies that a merge of the two components is necessary. Therefore, an $area\_tuple$ needs to be created. It maps the higher canonical point x_h to the lower x_l, resulting in a tuple of the form $\langle c, x_h, c, x_l \rangle$. For a faster collapse of transitive chains in the resolution stage, the inverse tuple $\langle c, x_l, c, x_h \rangle$ is additionally generated and stored. This entire process of performing the prefix-sums can be efficiently realized using a logarithmic merge tree across all available nodes.

4.5 Distributed Tuple Resolution

The resolution of the tuples happens essentially in the same mode as the generation of tuples. First, the $area\_tuples$ are resolved, which requires the iterative resolution of the weakly connected area components into strong ones, and then, second, the resolution of the edges for the normalized components. The respective high-level pseudo-code is displayed in Algorithm 4. It makes use of the distributed communication primitive ALLREDUCE several times. In line with this text it should be understood as a function that reduces, i.e. combines, vector of values with the same length and arbitrary, but consistent data types, element-wise using an operator given as second parameter. The single, final result vector is broadcast to all participating distributed compute nodes. ALLREDUCE can be efficiently implemented using a logarithmic communications tree.

Algorithm 4 Pseudo-code of a single iterations of the distributed area tuple resolution.

```
1:  @parallel
2:  function RESOLVE(area, roots)
3:      tuples ← []
4:
5:      loop
6:          all_done ← ALLREDUCE(EMPTY(roots), And)
7:          if all_done then break
8:
9:          grayv ← ALLREDUCE(MAX-KEY(roots), Max)
10:         unresolved ← true
11:         while unresolved do
12:             GLOBAL-SORT(area)
13:             rules ← RESOLVE-COMPONENTS(area(grayv))
14:             unresolved ← REMAP(area(grayv), rules)
15:             unresolved ← ALLREDUCE(unresolved, Or)
16:
17:         resolved ← RESOLVE-ROOTS(grayv, area, roots)
18:         tuples ← CONCAT(tuples, resolved)
19:
20:     return tuples
```

In general, the area tuple resolution is inspired by the distributed connected component labeling algorithm presented by Flick *et al* [28]. The goal is to turn a weakly connected graph, here the components, into a strongly connected ones, meaning directly pointing to the correct canonical point of an area without intermediate, transitive connections. This is achieved by following the graph edges until the most optimal, i.e., smallest pixel point is found. However, in distributed memory environments it might not be possible to follow all paths directly as they might be residing on a different machine. To overcome this, the longest local partial graph paths are computed and iteratively shortened until the strong graphs are found. For this, all tuples are globally sorted, i.e., across all nodes, and locally linearly scanned for the most optimal candidate, remapped and saved. This process is repeated until convergence is achieved, which is equal to having no tuples remapped in the current iteration. Technically, there are two challenges involved in this.

First, there is the problem of globally sorting (GLOBAL-SORT in Alg. 4) the tuples. This means that all tuples need

to be partially ordered, so that the smallest element is on the node with the smallest rank and the maximal tuple on the node with the highest rank. For this, the distributed max-tree algorithm uses an enhanced version of the parallel sorting by regular sampling algorithm [47]. In the variant that is proposed here, the number of tuples are additionally balanced after sorting them, in order to keep the workload equal on each node.

Algorithm 5 Pseudo-code of the component resolution algorithm mapping weakly connected tuples into stronger ones.

```
1:  function RESOLVE-COMPONENT(area_tuples)
2:      rules ← {}
3:
4:      for all tuple ∈ area_tuples do
5:          from ← tuple.from
6:          to ← tuples.to
7:          if to > from then SWAP(to, from)
8:          canonical ← CANONIZE(rules, from)
9:          min ← MIN(canonical, to)
10:         max ← MAX(canonical, to)
11:         rules[from] = min
12:         rules[max] = min
13:
14:     ends ← [FRONT(area_tuples), BACK(area_tuples)]
15:     LEFT-PREFIX-SUM(ends, rules, Min)
16:     RIGHT-PREFIX-SUM(REVERSE(ends), rules, Min)
17:
18:     return rules
```

Second, there is the problem of finding the strongest connected components local to each node. The approach proposed here, is sketched in Algorithm 5. Each of the $area\_tuples$ is linearly scanned and the most optimal, i.e. smallest canonical point memorized in an associate map called $rules$. These can then be applied to the tuples by scanning them once again and modifying the During the transitive solution the entire rule chain needs to always be canonized and the currently known minimum pointed to (lines 8–12). If the tuples are balanced during the global sorting step, the resolution of the partial component graphs, i.e., all the tuples with the same origin, may additionally be fragmented across the memory of multiple nodes. Therefore, the distribute max-tree algorithm must connect these partial graphs in each iteration. This can be achieved by exchanging the start and end of the sorted tuple chain including the found canonical point with the direct neighbors. Given that the neighboring node proposes a better canonical point, it is adopted instead of the one found locally. Transitivity is achieved by logarithmically merging these chains across all available machines. For this two prefix-sums (sometime also cumulative sum or scan, lines 15–16), first from left to right across the ranks, and then in reverse to propagate potentially better canonical points back.

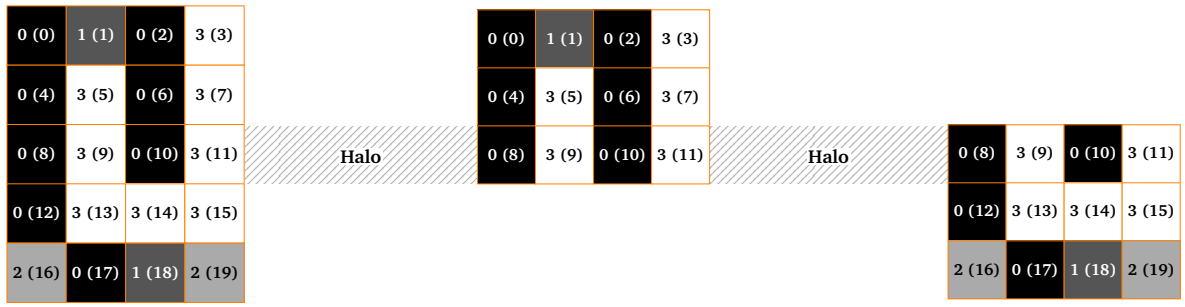
Furthermore, each tuple exists twice. Once in the "correct direction" pointing from the larger pixel index to the lower index and its inverse, pointing from low to high. Whenever a tuple is remapped, the tuple is flipped, i.e., the direction is changed, for the next iteration in order to back propagate this change to its inverse. The reason behind this is, that the inverse tuple might have found an even more optimal canonical point coming from the other side of the chain, due to say half circular structures on the image. Both tuples are then updated and the transitive chain collapsed much quicker. As an effect of this, the number of iterations heavily reduces, as introduced by Flick *et al* [28].

After the canonicalization of the weak connected components into strong connected components, the roots for each of the canonical points must be found. The corresponding approach is sketched in Algorithm 6. For this, the $area\_tuples$ and $root\_tuples$ are merged first in order to normalize of the components' canonical points (lines 2–4). Only then can the most optimal root for each component be determined. This is achieved by once again linearly scanning the tuples and memorizing the best candidates in an associative map called $best\_roots$. Similarly to the $area\_tuples$ fragmentation, root candidates of a single component may be scattered across multiple nodes. Therefore, they must be connected two prefix-sum operations (lines 7–10).

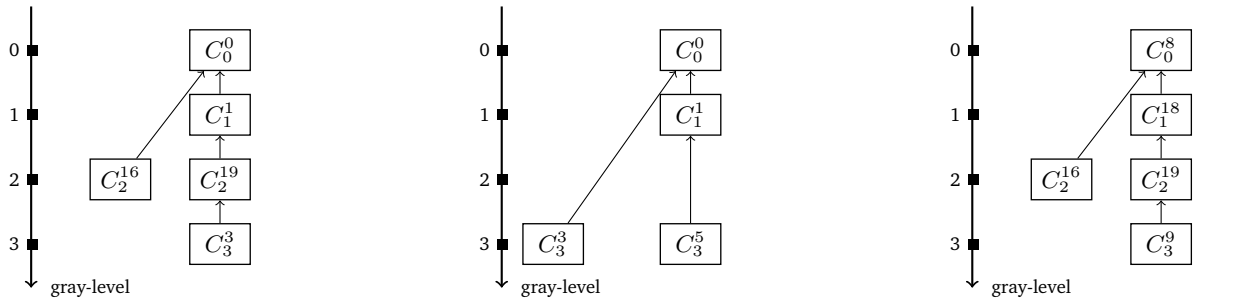
In a third and final linear scan over the combined tuples the found best roots are evaluated. There are four possible options. First, the tuple's root gray-value is smaller, i.e. it is further up in the tree, than the best root. In this case, a tuple needs to be created that connects the best root transitively with the one from the tuple (see case 2). Second, the gray-value of the tuple and the best root match, but the root has a smaller canonical point (see case 3) This means that the root and its connected component as well as the one of the neighbor are weakly connected via the area of the current tuple. An area remapping tuples needs to be created, including its inverse, and added to the $area\_tuples$. Third, the current tuple already points correctly to the best root (see case 4), then the tuple is inverted, essentially pointing the "wrong" way around from the root to the lower area and pushed into the root's gray-valued tuple bucket. The reason behind this is, that the canonical point of the root might still be changed during the area resolution phase of its gray-value. Therefore, it may not be marked as finished yet, but kept until the respective gray-level of the roots has been resolved. Finally, the fourth condition (see case 1) is meant for inverted root tuples. After their normalization, they are flipped yet again into the "correct" order, i.e., pointing from high gray-values to low gray-values and send back to its respective tuple bucket.

4.6 Obtaining the Global Parent Image

After the tuples have been resolved the global parent image can be obtained by redistributing the tuples back to their original sub-image. For this, the each tuple is send back to the node, that contains the pixel with the index of the second tuple component, independent whether it is a area or edge tuple. Each of the area tuples is then stored in an associative container, mapping from the original points to the canonical point. Subsequently, while iterating over the image, each of the pixels is normalized to its correct canonical points using said data structure. The root tuples are handled slightly differently. Each tuples is visited once and the pixel index at the from part of the tuples is simply set to the destination of the tuple. After this step, each node posses the correct partial parent image representing the global max-tree.



(a) Unpartitioned image (b) Partial image of p_0 (c) Partial image of p_1



(d) Unpartitioned image (e) Max-tree of p_0's partial image. (f) Max-tree of p_1's partial image.

1) Identify partitioned components. Utilize conflicts in canonical point labeling within the halo areas ($area\_tuples$). Tuples in this step are doubled, from large to small and inverted.

HALO COMPONENTS	$\{3:[\langle 3,5,3,3\rangle, \langle 3,3,3,5\rangle]\}$	$\{0:[\langle 0,8,0,0\rangle, \langle 0,0,0,8\rangle]$ $3:[\langle 3,9,3,3\rangle, \langle 3,3,3,9\rangle]\}$

2) Determine the edges of the boundary max-trees and represent them as tuples ($root\_tuples$).

HALO-TREE-EDGES	$\{3:[\langle 3,3,0,0\rangle, \langle 3,5,1,1\rangle]$ $1:[\langle 1,1,0,0\rangle]\}$	$\{3:[\langle 3,9,2,19\rangle]$ $2:[\langle 2,19,1,18\rangle]$ $1:[\langle 1,18,0,8\rangle]\}$

3) Iteratively resolve the generated the tuples beginning from the highest to the lowest gray-level—component first, then edges. Tuples with $\downarrow$ are dropped (incorrect or inverted), $\lightning$ mark conflicts during remapping and applied changes are bold.

RESOLVE

C_3

RESOLVE-COMPONENTS	$[\langle 3,3,3,5\rangle\downarrow, \langle 3,3,3,9\rangle\downarrow]$	$[\langle 3,5,3,3\rangle, \langle 3,9,3,3\rangle]$
NORMALIZE	$[\langle 3,3,0,0\rangle, \langle 3,5,1,1\rangle\lightning, \langle 3,5,3,3\rangle\lightning]$	$[\langle 3,9,2,19\rangle\lightning, \langle 3,9,3,3\rangle\lightning]$
RESOLVE-ROOTS	$[\langle 3,3,0,0\rangle\lightning, \langle \mathbf{3},\mathbf{3},\mathbf{1},\mathbf{1}\rangle\lightning]$	$[\langle \mathbf{3},\mathbf{3},\mathbf{2},\mathbf{19}\rangle\lightning]$
Generated tuples	$[\langle \mathbf{2},\mathbf{19},\mathbf{0},\mathbf{0}\rangle, \langle \mathbf{2},\mathbf{19},\mathbf{1},\mathbf{1}\rangle]$	[]

C_2

RESOLVE-COMPONENTS	[]	[]
NORMALIZE	$[\langle 2,19,0,0\rangle, \langle 2,19,1,1\rangle]$	$[\langle 2,19,1,18\rangle]$
RESOLVE-ROOTS	$[\langle 2,19,0,0\rangle\lightning, \langle 2,19,1,1\rangle\lightning]$	$[\langle 2,19,1,18\rangle\lightning]$
Generated tuples	$[\langle \mathbf{1},\mathbf{1},\mathbf{0},\mathbf{0}\rangle, \langle \mathbf{1},\mathbf{1},\mathbf{1},\mathbf{18}\rangle]$	$[\langle \mathbf{1},\mathbf{18},\mathbf{1},\mathbf{1}\rangle]$

C_1

RESOLVE-COMPONENTS	$[\langle 1,1,1,18\rangle\downarrow]$	$[\langle 1,18,1,1\rangle]$
NORMALIZE	$[\langle 1,1,1,18\rangle]$	$[\langle 1,18,0,8\rangle\lightning, \langle 1,18,1,1\rangle\lightning]$
RESOLVE-ROOTS	$[\langle 1,1,0,0\rangle\lightning]$	$[\langle \mathbf{1},\mathbf{1},\mathbf{0},\mathbf{8}\rangle\lightning]$
Generated tuples	$[\langle \mathbf{0},\mathbf{0},\mathbf{0},\mathbf{8}\rangle]$	$[\langle \mathbf{0},\mathbf{8},\mathbf{0},\mathbf{0}\rangle]$

C_0

RESOLVE-COMPONENTS	$[\langle 0,0,0,8\rangle\downarrow, \langle 0,0,0,8\rangle\downarrow]$	$[\langle 0,8,0,0\rangle, \langle 0,8,0,0\rangle\downarrow]$
NORMALIZE	$[\langle 0,8,0,0\rangle]$	[]
RESOLVE-ROOTS	[]	[]
Generated tuples	[]	[]

Fig. 3: Toy example demonstrating the tuple resolution of the distributed max-tree algorithm.

TABLE 1: Overview of the worst-case time, space and message complexity of the distributed max-tree algorithm steps.

	Time	Space	Messages
Image chunking	$\mathcal{O}(\frac{n}{p})$	$\mathcal{O}(\frac{n}{p})$	$\mathcal{O}(1)$
Local max-tree	$\mathcal{O}(\frac{n}{p\times t} \times log(d) + w \times log(t))$	$\mathcal{O}(\frac{n}{p} + t \times w)$	$-$
Root tuple generation	$\mathcal{O}(w \times d)$	$\mathcal{O}(w \times d)$	$-$
Area tuple generation	$\mathcal{O}(w \times log(p))$	$\mathcal{O}(w)$	$\mathcal{O}(log(p))$
Tuple resolution	$\mathcal{O}(k \times d \times \frac{w}{d} \times log(w \times d) \times log(p))$	$\mathcal{O}(w \times d)$	$\mathcal{O}(k \times d \times log(p))$
Redistribution	$\mathcal{O}(w \times d)$	$\mathcal{O}(w \times d)$	$\mathcal{O}(log(p))$
Application	$\mathcal{O}(w \times d + \frac{n}{p} \times log(w \times d)))$	$\mathcal{O}(w \times d)$	$-$

Algorithm 6 Pseudo-code of the distributed resolution of the root tuples.

```
1:  function RESOLVE-ROOTS(grayv, area, roots)
2:      combined ← CONCAT(area(grayv), roots(grayv))
3:      GLOBAL-SORT(combined)
4:      NORMALIZE(combined)
5:
6:      ▷ Determine best root for each components globally
7:      best_roots ← FIND-BEST-ROOTS(combined)
8:      ends ← [FRONT(combined), BACK(combined)]
9:      LEFT-PREFIX-SUM(ends, Min)
10:     RIGHT-PREFIX-SUM(REVERSE(ends), Min)
11:
12:     for all tuple ∈ combined do
13:         root ← best_root[tuple.from]
14:         if tuple.grayv > tuple.n_grayv then        ▷ Case 1
15:             PUSH(roots(tuple.grayv), INVERT(tuple))
16:         else if tuple.grayv < root.grayv then      ▷ Case 2
17:             PUSH(roots(root.grayv), ⟨root.grayv,
                    ↪ root.from, tuple.n_grayv, tuple.to⟩
18:         else if tuple.n_grayv = root.grayv and
                ↪ root.pixel < tuple.to then           ▷ Case 3
19:             t_to ← CANONIZE(tuple.to)
20:             r_to ← CANONIZE(root.to)
21:             n_c ← tuple.n_grayv
22:             min_to ← MIN(t_to, r_to)
23:             max_to ← MAX(t_to, r_to)
24:             PUSH(area[n_c], ⟨n_c, tuple.to, n_c, min_to⟩)
25:             PUSH(area[n_c], ⟨n_c, min_to, n_c, tuple.to⟩)
26:             PUSH(area[n_c], ⟨n_c, max_to, n_c, min_to⟩)
27:             PUSH(area[n_c], ⟨n_c, min_to, n_c, max_to⟩)
28:         else                                       ▷ Case 4
29:             PUSH(roots(root.grayv), ⟨root.grayv,
                    ↪ root.from, tuple.grayv, tuple.from⟩
30:
31:     return tuples
```

5 IMPLEMENTATION

The proposed parallel algorithm has been implemented in C++ and is available on the open-source code repository Github [48]. The coarse-grained parallelization across multiple nodes has been realized using the Message Passing Interface (MPI) [49]. For the shared-memory implementation C++11 native threads have been used. The algorithm accepts data loaded from files in the Hierarchical Data Format 5 (HDF5) format [50], which it will also store the resulting parent image to.

5.1 Complexity

In this section the time and space complexity for the algorithm steps of the distributed max-tree computation are laid out. A summary can be found in Table 1. The used symbols are explained in Section 4.1. All formulas are given for the worst-case scenario.

The time and space complexity for loading the sub images can be straight-forward inferred and amount to the number of total pixels divided by the amount of available processing nodes, as each of the receives an equally size chunk of the entire problem. This requires to potentially exchange one message in which the image dimensions are broadcast. For the local max-tree computation each of the t thread needs to allocated the part of parent image, that is equal in size to the processed raw image, plus and additional area remapping that is solely dependent on the image width, explaing the space complexity. The computational complexity consists of the linear image-scan for each thread and sub-image, which in turn need to do look-ups into the associative container for the $stacks$, resulting in the first summand. However, each thread needs to be merged with its direct neighbors, which can be done in logarithmic fashion as explained in Section 4.3, along the virtual split boundaries, i.e., the width of the image.

The next two steps involve the generation of the tuples. In the worst case, for each of the boundary pixels—again the width of the image—a tuple needs to created, resulting in according space and time complexity. Although, in practice the number will most of the times be much lower, because the boundary zones mostly consists of connected flat zones with shared parents nodes, resulting in early outs. In fact, the average complexity should therefore be closer to $\mathcal{O}(w * log(\frac{d}{2}))$, but is for obvious reasons dependent on the analyzed data. Contrary to root tuples, which can be generated entirely local, area tuples needs to be stitched together across all processing nodes, resulting in the additional factor for the time complexity. Using a two prefix-sum operations this information can be exchanged in a logarithmic number of communication steps, explaining the messaging complexity.

The main resolution consists of k iterations of sorting, linearly scanning and remapping the tuples for each of the available gray-levels of the gray depth d. It is assumed here that the number of tuples per gray-level is more or less evenly distributed, resulting in the purposefully chosen term $\frac{w}{d}$ (it would actually cancel out with d). The sorting adds both logarithmic components, over the number of tuples and nodes, as it requires reordering them across all machines. One of the major uncertainty factors is the iteration constant k, which is dependent on the data. In the worst case, k

is equal to the number of processing nodes p as upper bound, given a single tuple needs to visit each single machine due to transitivity. However, in practice, the iteration count will remain low, typically only one or two, even for a high number of nodes, due to the area stitching step and tuple inversions, effectively minimizing the canonicalizations. The corresponding message complexity can be explained in a similar fashion. For each of the d gray-levels, k iterations are performed requiring the communication of the tuples using a logarithmic communication primitive across all cores. Thereby, the tuples can be exchanged in whole, not requiring to break them down into individual messages.

Finally, the resolved tuples need to be send back to the partial image of origin and locally applied. The former step, requires to exchange messages between each of the nodes. Using a logarithmic communication tree, this can be done ad hoc, explaining the message complexity. The later step requires iterating over the whole image and normalizing each of the pixels using an associative data structure with logarithmic look up time. This is the reasons for the second summand in the time complexity equation. The first can be explained by the changing roots by directly assigning them while iterating through the tuples.

Generally, the algorithm has complexity classes that are either linear or linear-logarithmic, supporting good scalability overall. The only bottleneck seems to be the iterative constant k that could potentially degrade into the number of used compute nodes p as upper bound. However, this is rarely the case in practical use and presents an opportunity for future research.

5.2 Implementation Details

For the algorithm implementation the Message Passing Interface (MPI) [51] programming framework has been used. It provides low-level network communication primitives to exchange one-to-one and many-to-many messages between the participating distributed nodes. Efficient algorithms usually rely on the later category of operations, the so-called *collectives*, due to possibility of achieving a logarithmic scaling, in time and number of messages, across the number of cores. The proposed implementation makes use of two concrete function. On the one hand this is `MPI_Allreduce` and on the other hand this is MPI_Scan. The latter is a concrete realization of the prefix-sum operation, accepting a vector of input values and element-wise calculating the left partial sum with respect to the passed binary operator. In the proposed algorithm implementation is used at various points, e.g. connecting the components in the halo areas or to exchange the canonical points of the partial strong connected components (see Section 4.5).

Specifically for the generation of the $area\_tuples$ that connects the halo zones, two prefix-sums, alternating from left to right, are necessary. The first operations identifies conflicts in the canonical point labeling and the second propagates them back. In order to retain the complete information about all remapping rules for all nodes, the prefix-sum function (`MPI_Scan`) would require an exchange buffer with a worst case memory complexity of $\mathcal{O}(p * w * 2)$—i.e. two halo zones $2 * w$ for each of the p nodes. This is highly undesirable, as it scales both, with the number of processing nodes as well as the width of the image. One can realize this operation more efficient, if only the outer boundaries of the already merged images are communicated and the intermediary remapping rules are memorized in a data structure, e.g., a map, across the two `MPI_Scan` calls. Then, the memory complexity of the sent buffer simply becomes $\mathcal{O}(w * 2)$.

The MPI framework does allow the registration of custom functions for collective calls, such as `MPI_Scan`. These must be associative and optionally commutative, which is satisfied by the above operation. In practice, however, such a reduction function additionally needs to have static linkage, or in other words, it must be a singleton. Furthermore, due to the definition of the MPI API standard, it is also not possible to pass any context or state, say an object pointer to the aforementioned map, to the prefix-sum. For this reason, a straight forward realization of the stateful `MPI_Scan` is not possible. Yet, there is the alternative of working around this limitation by accessing static data structures, e.g. a map modeling the local execution context, within the scope of the custom reduction operation. This could be a globally defined variable or static class member.

Algorithm 7 Pseudo-code of a thread-safe, stateful MPI reduction operation and subsequent usage by a prefix-sum.

```
1: mutex ← CREATE_MUTEX()
2: rules ← {}
3:
4: procedure REDUCTIONOPERATION(in, out)
5:     LOCK(mutex)
6:     local_rules ←FIND(rules, thread_id)
7:     UNLOCK(mutex)
8:     MERGE(in, out, local_rules)                ▷ actual work
9:
10: procedure CAPTURESTATE(local_rules)
11:     LOCK(mutex)
12:     PUT(rules, local_rules)
13:     UNLOCK(mutex)
14:
15: local_rules ← {}
16: op ←MPI_OP_CREATE(REDUCTIONOPERATION)
17: CAPTURESTATE(local_rules)
18: MPI_SCAN(..., op)
```

In this case, though, the whole distributed max-tree implementation effectively also becomes a singleton and may not be used in multi-threaded environments, which is suboptimal for a number of analysis uses cases. Therefore, it has been chosen an approach as sketched in Algorithm 7. A set of potential remapping data-structures from different threads is stored in a global associative container, here $rules$, guarded by a $mutex$. Before calling an `MPI_Scan` the remapping data-structure must be stored and during the custom reduction operation retrieved. In order to be able to correctly retrieve the remapping data-structure a unique, shared key must be chosen, for example the current thread identifier.

One enhancement to and possibility for future research on the MPI standard versions could be, to directly allow passing context pointers to every API call that utilizes reduction operations. This pointer is simply forwarded to the custom reduction operation on each invocation and then used to

realized stateful behaviour. In case a context is not needed, it may be set to null. As a result, this would remove the locking overhead and the code becomes cleaner and more understandable.

6 Experimental Evaluation

6.1 Environment

The experiments have been performed on the JURECA system [52] at the Juelich Supercomputing Centre. The system consists of 1884 compute nodes with each having two *Intel® Xeon® E5-2680 v3 Haswell* CPUs with 12 cores at 2,5 GHz and Hyperthreading. 1604 compute nodes have 128 GiB, 128 nodes 256 GiB, 74 node 512 GiB and two nodes 1024 GiB DDR5 RAM. For our experimental evaluation the following software libraries have been used—`HDF5 1.8.18 parallel` and `ParaStation MPI 5.1.9`. Source code has been compiled with `g++ 5.4.0` optimization level `03`. The available benchmark for the experiments relies on a maximum of 32 nodes and 24 threads.

6.2 Datasets

As for the used data, the tests have been performed on two real-world images depicted in Figure 4. The first dataset is a Pléiades Ortho Product [1] that was acquired over the Naples metropolitan area (Italy) in 2013. It includes four Pan-sharpened images with spatial and radiometric resolution of 0.5m and 8 bpp, respectively. This dataset was selected due to its free availability, its sufficient size and spatial resolution, which are relevant to the needs of the remote sensing community for scalable and accurate methods that allow to classify rapidly and accurately objects of interest over vast areas, as was discussed in Section 1 (e.g., produce very high resolution land cover mapping at the European scale). The second dataset is an image that was taken at the ESO Paranal Observatory in Chile by the Visible and Infrared Survey Telescope for Astronomy (VISTA). It portraits more than 84 million stars in the central regions of the Milky Way [53]. The Figure 4a and Figure 4b show the true-color image of both datasets. For the Naples dataset, experiments are performed only using the first channel [54]. For the ESO, the RGB image is simplified to a singular luminance channel, similarly to how it was done by Moschini *et al.* [37] in order to obtain similar conditions for benchmarking the proposed algorithm. The luminance image is obtained through weighing and summing the channels, so that $L = 0.2126R + 0.7152G + 0.0722B$. However, in order to show that the algorithm scales regardless of the domain size of the gray-levels, three different quantization levels are derived from the luminance channel: 8-uint bpp, 16-uint bpp and 32-float bpp [55]. Contrary to [37], the original size of the image is preserved (≈9Gpx), since the JURECA system provides node with large memory.

6.3 Experimental Setup

As discussed in Section 3 there are a number of other serial and parallel versions of the algorithm. Most of them report different value permutations for the computation time, memory consumption, speed-up and scalability of their implementations. Carlinet *et al.* [39] provide their used benchmarks, datasets and the source codes in C++ for many different serial and parallel algorithms[2]. In order to compare results achievable by using serial and parallel computing, the Berger *et al.* [16] algorithm has been selected. Moschini *et al.* [37] proved that Berger is the fastest sequential algorithm for images with high quantization values and even floating. However the algorithms depend on the MILENA image processing library [56] (i.e., provide fundamental image types and I/O functionality) which was not designed to handle very large images and floating values. For these reasons it was necessary to re-write a new C++ implementation of the algorithm which is library independent. For the parallel processing case, the hybrid shared-memory parallel max-tree algorithm developed by Moschini *et al.* [37] was considered. The algorithm has been implemented in C using POSIX threads and the source code is available publicly[3]. Contrary to the MILENA library, this algorithm has been proposed with the purpose of dealing with large-scale and high-dynamic range images, and was therefore ready to be used out-of-the-box. It may be argued that the comparison is not entirely fair due to the different nature of the algorithms—i.e., shared-memory and distributed-memory—but can very well be investigated for the same number of utilized cores. The expectation naturally is that distributed memory implementation, as the one proposed here, are naturally going to have more overhead compared to shared-memory versions. To the best of our knowledge the only distributed max-tree algorithm has been proposed recently by Kazemier *et al.* [38], but the source code was not obtainable at the time of writing as it is not yet released.

TABLE 2: Hybrid: multithreading+MPI.

Nodes	1	1	2	2	4	4	8	8
Threads	1	2	2	4	4	8	8	16
Cores	1	2	4	8	16	32	64	128

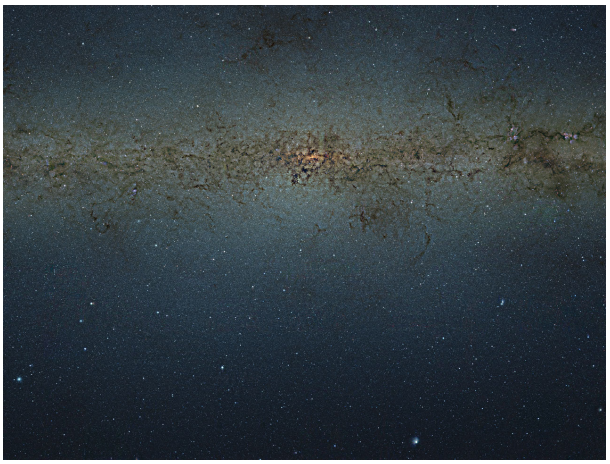
The performance assessment of the algorithms proposed by Berger and Moschini against the algorithm proposed in this paper is conducted with two kinds of benchmarks. The first type is focused on the computation time and speed-up, while the second measures memory consumption. Each benchmark configuration, meaning a particular node and core count, is executed five times and the following statistics are reported: mean μ, standard deviation σ, minimum, maximum, and coefficient of variation (CV), defined as $\nu = \frac{\sigma}{\mu}$ [57]. The use of the multithreading/MPI hybrid features of the algorithm allows to span the MPI process on each node available and to parallelize it locally using multithreading. For this reason, both types of benchmarks are performed on each number of cores, as shown in Table 2. The strategy is to evaluate first the performance on one core of one node. Afterwards the number of threads are doubled alternating with doubling the number of nodes, until the maximum of 128 cores across nodes and threads is reached.

1. http://www.intelligence-airbusds.com/en/23-sample-imagery

2. https://www.lrde.epita.fr/wiki/Publications/carlinet.14.itip

3. http://www.cs.rug.nl/ michael/ParMaxTree/

(a) Pan-sharpened, true-color image of Naples, Italy.



(b) Central Milky Way, captured by the ESO, Chile.

Fig. 4: Benchmark images used in the experimental evaluation of the algorithm.

6.4 Speed-up and Memory Consumption

The Figure 5 depicts the experimental results related to the processing time. For each dataset, the plot of the mean execution time and the plot of the speed-up for increasing number of cores is reported. In order to make a fair comparison with the state-of-the-art results (i.e., the shared-memory algorithm [37]), the proposed algorithm is first run on a single node. The algorithm's execution time measures the beginning and end of the `main()` function of the process with the MPI rank 0 and the thread number 0. The speed-up coefficient is computed as $t_p = t_1/t_c$. the fraction of the execution time with a single core and the execution time with multiple processing cores. Generally it can be said that the proposed algorithm is able to gain a substantial speed-up for both data sets and the different gray-levels quantizations.

For the 8bpp case, the algorithm shows a constant, near linear speed-up curve. In both datasets, the speed-up shows an increasing behavior for up to 256 cores with no reason to doubt its consistency for a higher number of cores, with an execution time of 9.38 and 62.35 seconds for Naples and ESO, respectively. However, for ESO 16bpp and 32bpp the speed-up flattens sooner, stabilizing at 64 and 16 cores, respectively. With a high gray-level depth, the number of tuples is increasing sharply, resulting in larger merge time. The effect observable here is the Amdahl speed-up boundary for a constant workload.

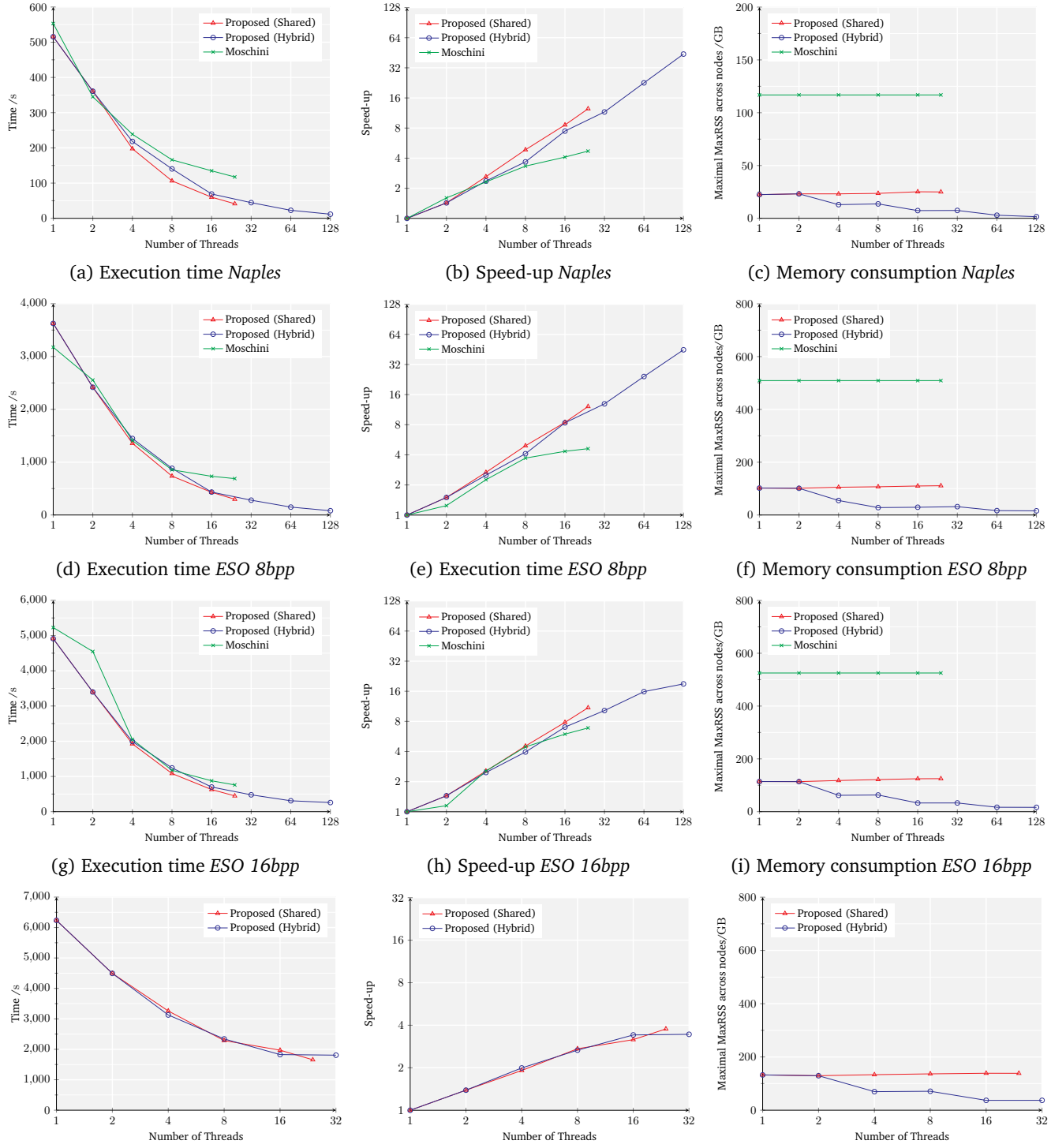
TABLE 3: Processing times (mean values in minutes and statistics) of the sequential Berger algorithm.

Images	Mean μ	StDev σ	CV	Min	Max
Naples	13.54	0.484	0.001	13.53	13.55
ESO 8bpp	113.76	6.011	0.001	113.62	113.91
ESO 16bpp	184.56	9.877	0.001	184.35	184.73
ESO 32bpp	185.67	19.457	0.002	185.34	186.21

When these results are compared with the Moschini algorithm, the proposed algorithm always provides faster execution times. Unfortunately, for the dataset ESO 32bpp it was not possible to derive any conclusions since the Moschini algorithm did not terminate. A more detailed analysis of the execution time for the different phases of the algorithm (see the Algorihtm 2) is depicted in Figure 6. The results are related only to a single dataset case (ESO 16bpp) because of space considerations. Each set of rows depicts a a specific number of nodes (i.e., 1 node, 2 nodes, 4 nodes and 8 nodes). The time distribution for increasing number of threads is shown in each row. For the single node case, the computation of the local max tree is the most time-consuming phase. This is a shared-memory scenario where the three phases concerning the management of the tuples do not take place. When the number of threads increases, the second most costly phase is the local apply. The local merge needs to be considered only beyond four threads. The same conclusions can be derived for the remaining nodes configuration. However, since it is a distributed memory environment, the phases connected with tuple handling are also present.

In the two-node case the tuple generation phase and the global apply are mostly present, the weight of the tuple resolution becomes more pronounced the higher the node count. This behavior can best be explained by the algorithm complexity, explained in Table 1, showing a logarithmic scaling with the number of processing cores. As has been shown here, the parallel implementation allows to achieve a significant processing time gain when compared with serial processing. In Table 3 the processing times for the different datasets are presented. When considering ESO 32bpp, which is the more challenging dataset used in this work, the proposed algorithm computes the max-tree in ≈27 minutes (with 24 cores) while Berger converges only after ≈3 hours.

Last but not least, comments should be made regarding the memory consumption of the proposed algorithm. Considering the usual trade-off between memory consumption and computational time, the experiments show the proposed algorithm is more memory efficient and takes shorter computational time than Berger and Moschini. Table 4 and Table 5 scrutinize the memory consumption (in GB) for the different algorithms with Naples and ESO 16bpp images, respectively, when computing on a single node. For each given number of threads, it can be noticed that the average and the maximum memory usage of all the tasks in the job are always lower for



(a) Execution time *Naples* (b) Speed-up *Naples* (c) Memory consumption *Naples*

(d) Execution time *ESO 8bpp* (e) Execution time *ESO 8bpp* (f) Memory consumption *ESO 8bpp*

(g) Execution time *ESO 16bpp* (h) Speed-up *ESO 16bpp* (i) Memory consumption *ESO 16bpp*

(j) Execution time *ESO 32bpp* (k) Speed-up *ESO 32bpp* (l) Memory consumption *ESO 32bpp*

Fig. 5: Execution time, speed-up and memory consumption curves of the proposed and Moschini's algorithm for increasing number of threads. The thread count for Moschini and Proposed (Shared) is increased locally on a single node. In case of the hybrid setting the number of threads and number of nodes are doubled alternately (refer to Table 2).

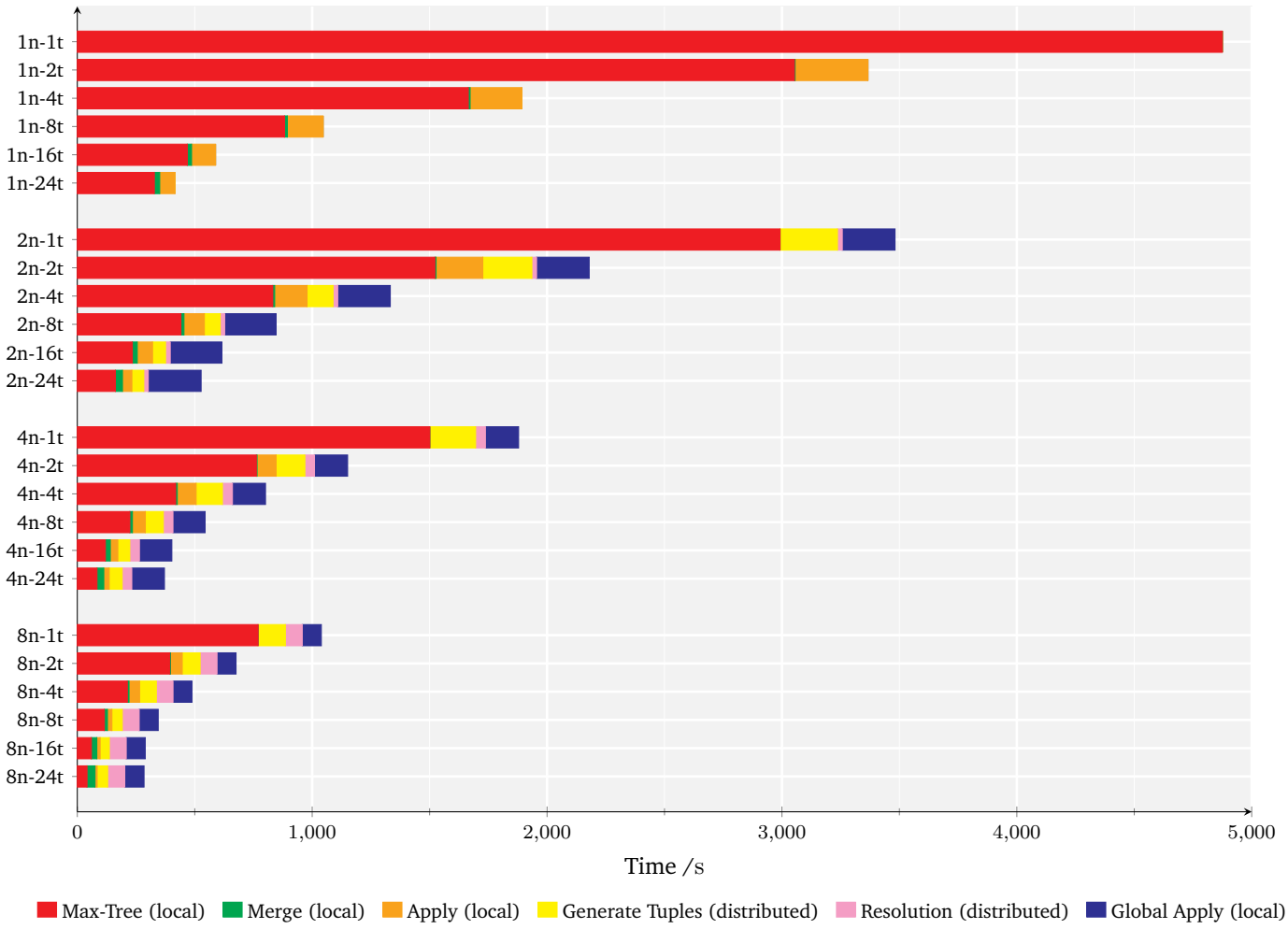


Fig. 6: Execution time distribution of the proposed algorithm for the *ESO 16bpp* dataset. n signifies the number of utilized nodes and t the number of threads for on each node.

TABLE 4: Memory consumption (mean values in GB and statistics) for the different algorithms with Naples image when using a single node. For each threads setup, the average and the maximum resident set size of all the tasks in the job are reported, respectively.

Algorithm	Threads	Mean μ	StDev σ	CV	Min	Max	
Berger *et al.* [16]	1	59.65	0.327	0.005	59.35	60.19	Average
		66.89	6.429	0.096	63.29	78.13	Maximum
Moschini *et al.* [37]	1	91.99	1.955	0.021	89.64	93.52	Average
		116.83	0.000	0.000	116.83	116.83	Maximum
	2	89.03	0.590	0.007	88.10	89.64	Average
		116.83	0.000	0.000	116.83	116.83	Maximum
	4	82.82	1.988	0.024	81.19	86.09	Average
		116.83	0.000	0.000	116.83	116.83	Maximum
	8	79.33	3.161	0.040	73.99	81.80	Average
		116.84	0.000	0.000	116.84	116.84	Maximum
	16	74.23	5.571	0.075	64.96	79.14	Average
		116.84	0.000	0.000	116.84	116.84	Maximum
	24	62.45	4.593	0.074	58.83	70.03	Average
		116.84	0.000	0.000	116.84	116.84	Maximum
Proposed	1	18.93	0.340	0.018	18.43	19.32	Average
		22.36	0.106	0.005	22.22	22.45	Maximum
	2	18.40	0.657	0.036	17.81	19.12	Average
		23.13	0.106	0.005	22.94	23.18	Maximum
	4	18.18	0.107	0.006	18.06	18.33	Average
		22.92	0.178	0.008	22.68	23.17	Maximum
	8	16.23	1.159	0.071	14.79	17.19	Average
		23.07	0.544	0.024	22.44	23.64	Maximum
	16	14.28	2.082	0.146	10.66	15.63	Average
		24.24	0.686	0.028	23.24	25.15	Maximum
	24	23.11	1.872	0.081	20.42	25.00	Average
		22.35	0.099	0.004	22.18	22.41	Maximum

TABLE 5: Memory consumption (mean values in GB and statistic) for the different algorithms with ESO 16bpp image when using a single node. For each threads setup, the average and the maximum resident set size of all the tasks in the job are reported.

Algorithm	Threads	Mean μ	StDev σ	CV	Min	Max	
Berger *et al.* [16]	1	292.30	0.217	0.001	291.93	292.48	Average
		296.70	0.001	0.000	296.70	296.70	Maximum
Moschini *et al.* [37]	1	457.71	0.777	0.002	456.98	458.61	Average
		525.21	0.001	0.000	525.21	525.21	Maximum
	2	472.53	0.898	0.002	471.54	473.76	Average
		525.21	0.001	0.000	525.21	525.21	Maximum
	4	436.67	1.761	0.004	434.06	437.89	Average
		525.21	0.001	0.000	525.21	525.22	Maximum
	8	409.87	3.323	0.008	406.41	414.84	Average
		525.21	0.001	0.000	525.21	525.22	Maximum
	16	383.99	2.466	0.006	381.68	387.10	Average
		525.22	0.001	0.000	525.22	525.22	Maximum
	24	380.92	5.482	0.014	376.22	388.25	Average
		525.22	0.001	0.000	525.22	525.22	Maximum
Proposed	1	102.39	0.091	0.001	102.32	102.54	Average
		114.13	0.052	0.000	114.08	114.20	Maximum
	2	103.04	0.307	0.003	102.61	103.43	Average
		113.84	0.028	0.000	113.81	113.88	Maximum
	4	102.34	0.528	0.005	101.65	103.05	Average
		117.99	0.036	0.000	117.95	118.03	Maximum
	8	100.80	0.458	0.005	100.25	101.46	Average
		121.55	0.068	0.001	121.48	121.62	Maximum
	16	94.86	1.264	0.013	93.58	96.30	Average
		124.43	0.185	0.001	124.27	124.72	Maximum
	24	94.27	1.714	0.018	92.95	96.19	Average
		124.82	0.241	0.002	124.46	125.05	Maximum

the proposed algorithm. This means that the algorithm is able to scale in terms of memory consumption and communication cost with respect to large datasets and the number of parallel cores. This is an important factor, considering most of the time the main constraint lies in the memory size.

7 Conclusion

In this work a new parallel and distributed algorithm for the computation of the max-tree of an image has been presented. The parallelization strategy consists of splitting the entire problem, i.e., the image, into equal-sized sub images, for which the partial max-trees are computed that are subsequently merged at the split boundaries. Using this algorithm, substantial speed-ups and scalability could be achieved in computing the max-tree on large real-world images, outperforming the state-of-the-art shared memory implementation. In particular, faster execution time and significantly less memory consumption can be achieved. The proposed algorithm allows to process gray-scale image of arbitrary gray-level depth including floating point values. This makes it suitable for the usage in large-scale image classification task, such as land cover type prediction, which is one of the major practical application domains.

In future work, the equivalent min-tree algorithm including distributed attribute filter are going to be implemented. This will set a solid foundation for the next research goal, the massive parallelization of the tree of shapes [58]—a contrast independent component tree representation of images.

Acknowledgments

The authors would like to thank Igancio Toledo and Martin Kornmesser for making the ESO/VVV Survey/D. Minniti image with the id *eso1242a* publicly available.

References

[1] G. Matheron, *Random Sets and Integral Geometry*. New York: John Wiley & Sons, 1975.
[2] J. Serra, *Image Analysis and Mathematical Morphology*. London: Academic Press, 1982.
[3] E. J. Breen and R. Jones, "Attribute Openings, Thinnings, and Granulometries," *Computer Vision and Image Understanding*, vol. 64, no. 3, pp. 377–389, 1996.
[4] P. Salembier, A. Oliveras, and L. Garrido, "Antiextensive Connected Operators for Image and Sequence Processing," *IEEE Transactions on Image Processing*, vol. 7, no. 4, pp. 555–570, 1998.
[5] L. Najman and J. Cousty, "A Graph-based Mathematical Morphology Reader," *Pattern Recognition Letters*, vol. 47, pp. 3–17, 2014.
[6] J. B. Kruskal, "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem," *Proceedings of the American Mathematical Society*, vol. 7, pp. 48–50, 1956.
[7] G. K. Ouzounis and P. Soille, "The Alpha-Tree Algorithm," *Publications Office of the European Union*, 2012.
[8] P. Salembier and L. Garrido, "Binary Partition Tree as an Efficient Representation for Image Processing, Segmentation, and Information Retrieval," *IEEE Transactions on Image Processing*, vol. 9, no. 4, pp. 561–576, 2000.
[9] R. Jones, "Component Trees for Image Filtering and Segmentation," in *Procedings of the IEEE Workshop on Nonlinear Signal and Image Processing (NISP)*, E. Coyle, Ed., 1997.
[10] V. Caselles, B. Coll, and J. M. Morel, "Topographic Maps and Local Contrast Changes in Natural Images," *International Journal of Computer Vision*, vol. 33, no. 1, pp. 5–27, 1999.
[11] E. Carlinet and T. Géraud, "MToS: A Tree of Shapes for Multivariate Images," *IEEE Transactions on Image Processing*, vol. 24, no. 12, pp. 5330–5342, 2015.
[12] P. Salembier and J. Serra, "Flat Zones Filtering, Connected Operators, and Filters by Reconstruction," *IEEE Transactions on Image Processing*, vol. 4, no. 8, pp. 1153–1160, 1995.
[13] G. K. Ouzounis and M. H. F. Wilkinson, "Mask-Based Second-Generation Connectivity and Attribute Filters," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 29, no. 6, pp. 990–1004, 2007.
[14] M. Dalla Mura, J. A. Benediktsson, B. Waske, and L. Bruzzone, "Morphological Attribute Filters for the Analysis of Very High Resolution Remote Sensing Images," in *IEEE International Geoscience and Remote Sensing Symposium (IGARSS '09)*, vol. 3, 2009, pp. 2–3.
[15] J. A. Benediktsson, L. Bruzzone, J. Chanussot, M. Dalla Mura, P. Salembier, and S. Valero, "Hierarchical Analysis of Remote Sensing Data: Morphological Attribute Profiles and Binary Partition Trees," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6671 LNCS, 2011, pp. 306–319.
[16] 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 *IEEE International Conference on Image Processing*, 2007, pp. IV – 41–IV – 44.
[17] P. Teeninga, U. Moschini, S. C. Trager, and M. H. F. Wilkinson, "Improved Detection of Faint Extended Astronomical Objects Through Statistical Attribute Filtering," in *Mathematical Morphology and Its Applications to Signal and Image Processing (ISMM): 12th International Symposium*, J. A. Benediktsson, J. Chanussot, L. Najman, and H. Talbot, Eds. Springer International Publishing, 2015, pp. 157–168.
[18] I. K. E. Purnama, K. Y. E. Aryanto, and M. H. F. Wilkinson, "Non-Compactness Attribute Filtering to Extract Retinal Blood Vessels in Fundus Images," *International Journal of E-Health and Medical Communications (IJEHMC)*, vol. 1, no. 3, pp. 16–27, 2010.
[19] F. N. Kiwanuka and M. H. F. Wilkinson, "Automatic Attribute Threshold Selection for Morphological Connected Attribute Filters," *Pattern Recognition*, vol. 53, no. C, pp. 59–72, 2016.
[20] G. Cavallaro, N. Falco, M. D. Mura, and J. A. Benediktsson, "Automatic Attribute Profiles," *IEEE Transactions on Image Processing*, vol. 26, no. 4, pp. 1859–1872, 2017.
[21] B. Song, M. Dalla Mura, P. Li, A. Plaza, J. M. Bioucas-Dias, J. A. Benediktsson, and J. Chanussot, "Remotely Sensed Image Classification Using Sparse Representations of Morphological Attribute Profiles," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 52, no. 8, pp. 5122–5136, 2014.
[22] N. Falco, J. A. Benediktsson, and L. Bruzzone, "Spectral and Spatial Classification of Hyperspectral Images Based on ICA and Reduced Morphological Attribute Profiles," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 53, no. 11, pp. 6223–6240, 2015.
[23] M. Pedergnana, P. R. Marpu, M. Dalla Mura, J. A. Benediktsson, and L. Bruzzone, "Classification of Remote Sensing Optical and LiDAR Data Using Extended Attribute Profiles," *IEEE Journal of Selected Topics in Signal Processing*, vol. 6, no. 7, pp. 856–865, 2012.
[24] N. Falco, M. Dalla Mura, F. Bovolo, J. A. Benediktsson, and L. Bruzzone, "Change Detection in VHR Images Based on Morphological Attribute Profiles," *IEEE Geoscience and Remote Sensing Letters*, vol. 10, no. 3, pp. 636–640, 2013.
[25] G. K. Ouzounis, M. Pesaresi, and P. Soille, "Differential Area Profiles: Decomposition Properties and Efficient Computation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 8, pp. 1533–1548, 2012.
[26] N. Aldeborgh, G. K. Ouzounis, and K. Stamatiou, "Unsupervised object detection on remote sensing imagery using hierarchical image representations and deep learning," in *Proceedings of BiDS17 (Conference on Big Data from Space 2017)*, Toulouse, France, 2017, pp. 275–278.
[27] G. Ouzounis and M. Wilkinson, "A parallel dual-input max-tree algorithm for shared memory machines," in *Mathematical Morphology and Its Applications to Signal and Image Processing (ISMM): 8th International Symposium*, 2007, pp. 449–460.
[28] P. Flick, C. Jain, T. Pan, and S. Aluru, "A Parallel Connectivity Algorithm for De Bruijn Graphs in Metagenomic Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '15. New York: ACM, 2015, pp. 1–11.
[29] R. Jones, "Connected Filtering and Segmentation Using Component Trees," *Computer Vision and Image Understanding*, vol. 75, no. 3, pp. 215–228, 1999.

[30] L. Najman and M. Couprie, "Building the component tree in quasi-linear time," *IEEE Transactions on Image Processing*, vol. 15, no. 11, pp. 3531–3539, 2006.
[31] R. E. Tarjan, "Efficiency of a good but not linear set union algorithm," *ACM Journal*, vol. 22, no. 2, pp. 215–225, 1975.
[32] W. H. Hesselink, "Salembier's min-tree algorithm turned into breadth first search," *Information Processing Letters*, vol. 88, no. 5, pp. 225 – 229, 2003.
[33] D. Nistér and H. Stewénius, "Linear time maximally stable extremal regions," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5303 LNCS, no. PART 2, 2008, pp. 183–196.
[34] M. H. F. Wilkinson, "A fast component-tree algorithm for high dynamic-range images and second generation connectivity," in *Proceedings - International Conference on Image Processing, ICIP*, 2011, pp. 1021–1024.
[35] M. H. F. Wilkinson, H. Gao, W. H. Hesselink, J. E. Jonker, and A. Meijster, "Concurrent Computation of Attribute Filters on Shared Memory Parallel Machines," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30, no. 10, pp. 1800–1813, 2008.
[36] G. K. Ouzounis and L. Gueguen, "Interactive collection of training samples from the max-tree structure," in *18th IEEE International Conference on Image Processing*, 2011, pp. 1449–1452.
[37] U. Moschini, A. Meijster, and M. Wilkinson, "A hybrid shared-memory parallel max-tree algorithm for extreme dynamic-range images," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PP, no. 99, pp. 1–1, 2017.
[38] J. J. Kazemier, G. K. Ouzounis, and M. H. F. Wilkinson, *Connected Morphological Attribute Filters on Distributed Memory Parallel Machines*. Springer International Publishing, 2017, pp. 357–368.
[39] E. Carlinet and T. Géraud, "A Comparative Review of Component Tree Computation Algorithms," *IEEE Transactions on Image Processing*, vol. 23, no. 9, pp. 3885–3895, 2014.
[40] P. Matas, E. Dokládalová, M. Akil, T. Grandpierre, L. Najman, M. Poupa, and V. Georgiev, "Parallel algorithm for concurrent computation of connected component Tree," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5259 LNCS, 2008, pp. 230–241.
[41] H. Samet and M. Tamminen, "Efficient component labeling of images of arbitrary dimension represented by linear bintrees," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 10, no. 4, pp. 579–586, 1988.
[42] J. Iverson, C. Kamath, and G. Karypis, "Evaluation of connected-component labeling algorithms for distributed-memory systems," *Parallel Computing*, vol. 44, pp. 53–68, 2015.
[43] Y. Shiloach and U. Vishkin, "An o(logn) parallel connectivity algorithm," *Journal of Algorithms*, vol. 3, no. 1, pp. 57 – 67, 1982.
[44] N. G. de Bruijn, "A Combinatorial Problem," *Koninklijke Nederlandsche Akademie Van Wetenschappen*, vol. 49, no. 6, pp. 758–764, 1946.
[45] X. Qin and H. Jiang, "A Dynamic and Reliability-driven Scheduling Algorithm for Parallel Real-time Jobs Executing on Heterogeneous Clusters," *Journal of Parallel and Distributed Computing*, vol. 65, no. 8, pp. 885–900, 2005.
[46] P. Soille, *Morphological Image Analysis: Principles and Applications*, 2nd ed. Springer-Verlag Berlin Heidelberg New York, 2004.
[47] H. Shi and J. Schaeffer, "Parallel Sorting by Regular Sampling," *Journal of Parallel and Distributed computing*, vol. 14, no. 4, pp. 361–372, 1992.
[48] M. Goetz, "Distributed Max Tree Implementation," 2017. [Online]. Available: https://bitbucket.org/markus.goetz/
[49] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable parallel programming with the message-passing interface*, 2000, vol. 40, no. 2-3.
[50] HDF Group, "Hierachical Data Format 5." [Online]. Available: http://www.hdfgroup.org/HDF5
[51] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.
[52] Jülich Supercomputing Centre, "JURECA: General-purpose supercomputer at Jülich Supercomputing Centre," *Journal of large-scale research facilities*, vol. 2, no. A62, 2016.
[53] R. K. Saito, D. Minniti, B. Dias, M. Hempel, M. Rejkuba, J. Alonso-García, B. Barbuy, M. Catelan, J. P. Emerson, O. A. Gonzalez, P. W. Lucas, and M. Zoccali, "Milky Way demographics with the VVV survey," *Astronomy & Astrophysics*, vol. 544, 2012.
[54] Pleiades Ortho Product Dataset: Pan-Sharpened Green Band Channel. [Online]. Available: https://b2share.eudat.eu/records/f2a45218461842f5a4f886ee461d4bcc
[55] ESO Dataset: 3 Luminance Channels with Different Quantization Levels. [Online]. Available: https://b2share.fz-juelich.de/records/46f12d8f0d294928a9e7baf4ab56e95b
[56] R. Levillain, T. Géraud, and L. Najman, "Why and how to design a generic and efficient image processing framework: The case of the Milena library," in *Proceedings of the IEEE International Conference on Image Processing (ICIP)*, Hong Kong, 2010, pp. 1941–1944.
[57] H. Abdi, "Coefficient of variation," *Encyclopedia of research design*, pp. 169–171, 2010.
[58] S. Crozet and T. Géraud, "A first parallel algorithm to compute the morphological tree of shapes of nD images," in *Proceedings of the 21st IEEE International Conference on Image Processing (ICIP)*, Paris, France, 2014, pp. 2933–2937.

Markus Götz received his Bachelor of Science degree in Software Engineering from Hasso-Plattner-Institute, University of Potsdam, Potsdam, Germany in 2010. In 2014 he has been awarded with a Master of Science degree in Software Engineering from the same institution. During this time has gathered experience in data analysis, image processing and data mining during his stays at Blekinge Tekniska Högskola, Sweden, the European Organization for Nuclear Research (CERN), Switzerland and mental images GmbH, Germany. Currently, he is with the Juelich Supercomputing Center, Germany and the University of Iceland in line with his Ph.D. studies. His research interests include high-performance computing, parallel algorithms, machine learning as well as time series and data analysis.

Gabriele Cavallaro received the B.S. and M.S. degrees in telecommunications engineering from the University of Trento, Italy, in 2011 and 2013, respectively. He holds a Ph.D. degree in Electrical and Computer Engineering from the University of Iceland, obtained in 2016. At present he is a postdoctoral research assistant at the Juelich Supercomputing Centre, Juelich, Germany. At this institute, he is part of a scientific research group focused on high productivity data processing within the Federated Systems and Data Division. His research interests include remote sensing and analysis of very high geometrical and spectral resolution optical data with the current focus on mathematical morphology and high performance computing. He was the recipient of the IEEE GRSS Third Prize in the Student Paper Competition of the 2015 IEEE International Geoscience and Remote Sensing Symposium 2015 (Milan, Italy, July 2015). He serves as a reviewer for IEEE Geoscience and Remote Sensing Letters and IEEE Journal of Selected Topics in Earth Observations and Remote Sensing.

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

Matthias Book Matthias Book is professor for software engineering at the University of Iceland. After receiving his doctoral degree from the University of Leipzig, he worked as Research Manager for the German software company adesso AG, led the Mobile Interaction group at the University of Duisburg-Essen's Ruhr Institute for Software Technology (paluno), and served as acting head of the Software Engineering and Information Systems Chair at Chemnitz University of Technology. His research focus is on facilitating collaboration between domain experts and technology experts in complex software projects.

Morris Riedel received his Ph.D. degree from Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, and started working in parallel and distributed systems in the field of scientific visualization and computational steering of e-science applications on large-scale HPC resources. He is an Adjunct Associate Professor with the School of Engineering and Natural Sciences, University of Iceland, Reykjavik, Iceland. He previously held various positions at the Juelich Supercomputing Centre, Juelich, Germany. At this institute, he is also the head of a scientific research group focused on high productivity data processing as a part of the Federated Systems and Data Division. The given lectures in universities such as the University of Iceland, University of Applied Sciences of Cologne, Cologne, Germany, and the University of Technology Aachen (RWTH Aachen), Aachen, Germany include High Performance Computing and Big Data, Statistical Data Mining, and Handling of large Datasets and Scientific and Grid Computing. His research interests include high productivity processing of big data in the context of scientific computing applications.