Scalable Algorithms for Hierarchical Image and Data Segmentation

Thèse de doctorat de l'Université Gustave Eiffel
Spécialité Informatique

École doctorale n°532, Mathématiques et STIC
Laboratoire d'Informatique Gaspard Monge, UMR 8049

Soutenue publiquement par
Josselin Lefèvre
le 30 juin 2026

Composition du jury

Rapporteurs

Camille Kurtz – Professeur, Université Paris Cité

Santiago Velasco-Forero – Chargé de recherches, MINES ParisTech, PSL Research University

Examinateurices

Bertrand Kerautret – Professeur, Université Lyon 2

Deise Santana Maia – Maîtresse de conférences, Université de Lille

Isabelle Sivignon – Chargée de recherches,CNRS, Université Grenoble Alpes

Direction de thèse

Jean Cousty – Professeur, ESIEE Paris, Université Gustave Eiffel

Benjamin Perret – Professeur, ESIEE Paris, Université Gustave Eiffel

Harold Phelippeau – Responsable R&D, Thermo Fisher Scientific

"When you have a watershed, everything is a topographic relief."

$\sim$ Manuel Tarjann

Contents

Notations **8**

Abstract **9**

1 Introduction **12**
1.1 Electron Microscopy 12
1.1.1 Concepts and Principles 12
1.1.2 Volume Electron Microscopy 13
1.2 From Imaging to Segmentation 16
1.3 Computational Challenges of Large Data 20
1.4 Objectives and Contributions 21

2 State of the Art on Morphological Hierarchies **26**
2.1 Introduction 26
2.2 Watershed Segmentation 27
2.2.1 Historical Background and Topographic Intuition 27
2.2.2 Frameworks and Principles 28
2.2.3 Marker-Based Watershed 30
2.3 Hierarchies of Segmentation 32
2.3.1 Component Trees 33
2.3.2 Hierarchies of Partition 34
2.3.3 Hierarchical Watershed 35
2.4 Prior Work on Scalable Graph and Tree Structures 38
2.4.1 Streaming Minimum Spanning Tree 38
2.4.2 Component Trees 40
2.4.3 Quasi-Flat Zone Hierarchy 42
2.5 Conclusion 43

3 Preliminary Mathematics **45**
3.1 Introduction 45

3.2 Graphs . . . 45
3.3 Hierarchies of Partition . . . 46
3.4 Binary Partition Hierarchy . . . 47
3.5 Watershed . . . 50
3.5.1 Watershed in Edge-Weighted Graphs: Watershed-Cut . . . 50
3.5.2 Watershed-cut from Markers . . . 52
3.5.3 Link with Binary Partition Hierarchy . . . 53
3.6 Hierarchical Watershed . . . 54
3.7 Conclusion . . . 55

4 Binary partition Hierarchy Out-of-Core Calculation 56
4.1 Introduction . . . 56
4.2 Distribution of Binary Partition Hierarchy . . . 57
4.3 An Algebraic Framework for Binary Partition Hierarchy Out-of-Core Computation . . . 59
4.4 High Level Calculi of Binary Partition Hierarchy Distribution . . . 60
4.4.1 Generic Partition . . . 61
4.4.2 Causal Partition . . . 66
4.5 Data Structures . . . 69
4.5.1 Representation of the BPH . . . 70
4.5.2 Graph Management . . . 72
4.6 Algorithms . . . 74
4.6.1 Select . . . 75
4.6.2 Join . . . 75
4.6.3 Insert . . . 80
4.7 Experiments . . . 82
4.8 Conclusion . . . 86

5 Out-of-Core Attribute and Watershed Algorithms 87
5.1 Introduction . . . 87
5.2 Distribution of Attributes . . . 89
5.3 Propagate Algorithm: a Fundamental Brick for Out-of-Core Attribute Computation . . . 90
5.4 Out-of-Core Attributes Algorithms . . . 94
5.4.1 Area . . . 95
5.4.2 Volume . . . 96
5.4.3 Height . . . 97
5.4.4 Minima . . . 97
5.5 Out-of-Core Watershed . . . 99
5.5.1 Marker-Based . . . 99
5.5.2 Rooted in Minima . . . 105

5.6 Out-of-Core Hierarchical Watershed . . . 108
5.6.1 In-Core Hierarchical Watershed . . . 108
5.6.2 Out-of-Core Persistence . . . 110
5.7 Conclusion . . . 112

6 Superpixel Based Out-of-Core Watershed 113
6.1 Introduction . . . 113
6.2 Overview . . . 114
6.3 Quasi Flat Zones for Marked Watershed . . . 116
6.3.1 Quasi Flat Zones and Hierarchies . . . 116
6.3.2 Marked Watershed on Superpixels . . . 117
6.4 Distribution of Region Adjacency Graph . . . 117
6.4.1 From Quasi-Flat Zones to Region Adjacency Graph . . . 118
6.4.2 Out-of-core Region Adjacency Graph . . . 120
6.5 Experiments . . . 122
6.5.1 Databases and Test Images . . . 122
6.5.2 Evaluated Methods . . . 122
6.5.3 Metrics . . . 123
6.5.4 Evaluation Results . . . 124
6.6 Conclusion . . . 125

7 Incremental Interactive Watershed Segmentation 127
7.1 Introduction . . . 127
7.2 A New Computation of Watershed Cut Based on Binary Partition Hierarchy . . . 131
7.2.1 Tree-node Marking . . . 131
7.2.2 Pixel labeling . . . 133
7.2.3 Incremental Management of User Interaction . . . 133
7.3 Strategies for Minimum Spanning Forest Labeling . . . 135
7.3.1 Breadth-First Search and Heuristics . . . 135
7.3.2 A Parallel Strategy . . . 135
7.4 Experimental Results . . . 137
7.4.1 Tested Methods . . . 137
7.4.2 Experiment with user generated seeds . . . 139
7.4.3 Experiment with randomly generated seeds . . . 140
7.4.4 Experiment on 3D images . . . 142
7.5 Assessment of Parallel Breadth-First Search . . . 143
7.5.1 Parameter Tuning . . . 143
7.5.2 Discussion on Parallelization Rate . . . 145
7.6 Conclusion . . . 146

8 Conclusion **148**

Appendix: Proofs **154**
8.1 Proof of Property 4.4.1 154
8.2 Proof of Lemma 4.4.1 156
8.3 Proof of Property 4.4.2 157
8.4 Proof of Property 4.4.3 158

Bibliography **159**

List of Figures **172**

List of Algorithms **177**

Notations

x	a vertex
$\{x, y\}$	an edge
$w(e)$	the weight of an edge e
G	the graph (V, E)
$\mathbf{P}$	a partition
$\mathcal{H}$	a hierarchy
$\lvert\mathcal{H}\rvert$	the number of regions in a hierarchy
$\prec$	total ordering on the edges
$\mathcal{B}^{\prec}$	a binary partition hierarchy by altitude ordering
$R^{\star}$	the set of non-leaf regions of a hierarchy
$\gamma(X, Y)$	the set of edges between the subgraphs X and Y

Abstract

Recent advances in imaging technologies across diverse scientific fields such as astronomy, remote sensing, and notably electron microscopy have led to an important growth in image sizes. Modern imaging techniques like focused ion beam scanning electron microscopy can now produce high-resolution 3D volumes on the scale of tens of gigabytes. While this increase in data resolution offers unprecedented opportunities for structural and biological analysis, it also poses significant computational challenges in terms of both memory and processing requirements.

Among the fundamental tasks in image analysis, segmentation remains a cornerstone, enabling the identification and isolation of meaningful regions in images. Segmentation is crucial for downstream applications such as object recognition, metrology, and visualization. However, as image sizes grow, traditional segmentation workflows face two major limitations. First, processing large images often incurs extended computation times, which is particularly detrimental in interactive segmentation scenarios where rapid feedback is essential. Second, the memory footprint of segmentation algorithms can exceed the capacity of standard computing hardware, leading to complete workflow failure.

While this second challenge can be addressed through distributed architectures, concerning standard workstations, a complementary strategy lies in the design of out-of-core algorithms. These methods avoid memory overload by dividing the dataset into manageable chunks and processing each piece sequentially while saving intermediate results to disk. Out-of-core algorithms have demonstrated effectiveness across several domains.

This thesis presents a novel framework for performing hierarchical image analysis using out-of-core techniques on a single workstation. The focus is on the *Binary Partition Hierarchy*, a tree-based structure widely used in mathematical morphology and graph-based image analysis. This structure allows on one hand the computation of *watershed*, a widely used segmentation technique and on the other hand many other hierarchical representation including *hierarchical watershed*. The established algorithm for its computation is non-separable and requires global knowledge of the data, making it inherently memory-intensive. While offering quasi-linear time complexity, it fails to scale efficiently when data no longer

fit in memory.

The core contribution of this work is the development of an out-of-core framework for computing and processing the binary partition hierarchy. The proposed framework includes a strategy to construct a patchwork representation of the BPH, ensuring a limited memory usage at each computation step. This enables the processing of data that was then impossible to process. Moreover, the framework supports the out-of-core computation of region-based attributes and allows complex segmentation operations, such as watershed and hierarchical watershed, to be implemented in a scalable and efficient manner.

This work aims to bridge the gap between high-resolution data acquisition and practical analysis by enabling resource-efficient processing without sacrificing segmentation quality. The proposed methods are evaluated on real-world datasets, demonstrating their effectiveness. This work contributes to the broader effort of making high-resolution image analysis accessible on commodity hardware through out-of-core algorithmic design.

The manuscript is organized as follows. Chapter 2 presents the state of the art in watershed and hierarchical segmentation, with a particular focus on their graph-based formulations around the binary partition hierarchy and minimum spanning trees. It first addresses the watershed transform and its main variants, then reviews the principal families of hierarchical representations, including component trees, hierarchies of partitions, and hierarchical watersheds, highlighting their relationships and connections to BPH. The chapter concludes with an overview of scalable computation strategies of hierarchies of segmentation.

Chapter 3 lays the mathematical foundation for this research by introducing key concepts and formal definitions. It begins with fundamental graph theory notions that are essential for understanding hierarchical image representations. The binary partition hierarchy is then formally defined, providing a structured framework for hierarchical data analysis. The chapter also explores the watershed and hierarchical watersheds, emphasizing their formulation on edge-weighted graphs and their close relationship with the binary partition tree. These theoretical foundations serve as a basis for the methodologies developed in the subsequent chapters.

Chapter 4 introduces the notion of *distribution of the binary partition hierarchy*. An algebraic framework is defined to formalize out-of-core computation, and data structures tailored for efficient hierarchical processing are described. Based on three core operations, namely *select*, *join*, and *insert*, the chapter presents two high-level calculi to compute the distributed BPH together with efficient algorithms for these core operations. The first calculus supports generic partitions, allowing flexible data distribution, while the second focuses on causal partitions, ensuring strict out-of-core constraints. The advantages and limitations of these

approaches are analyzed through theoretical properties and an experimental comparison.

Building upon this framework, Chapter 5 extends the out-of-core computation scheme to watershed segmentation. The chapter first discusses the distribution of attributes, then addresses watershed computation and finally its hierarchical counterpart. Several out-of-core algorithms for attribute computation are introduced, along with the implementation of out-of-core watershed segmentation and its hierarchical extension.

Chapter 6 discusses the practical implementation of the proposed framework when considering superpixels as quasi-flat zones instead of pixels. It details the methodology required to apply the introduced calculi on a region adjacency graph rather than a pixel adjacency graph, and provides experimental results that evaluate the impact of this superpixel-based strategy on both computation time and memory consumption.

Chapter 7 presents an incremental in-core marker-based watershed algorithm that leverages the hierarchical representation to exploit the inherent causality of the interactive segmentation process. This method enables updating the segmentation in time proportional to the number of pixels affected when adding or removing markers. The chapter also introduces a parallel strategy for minimum spanning forest labeling. Both methods are evaluated experimentally, demonstrating their practical benefits.

Finally, Chapter 8 concludes this work by summarizing the main contributions and outlining several perspectives for future research, including possible extensions of the proposed framework and new application domains.

Chapter 1

Introduction

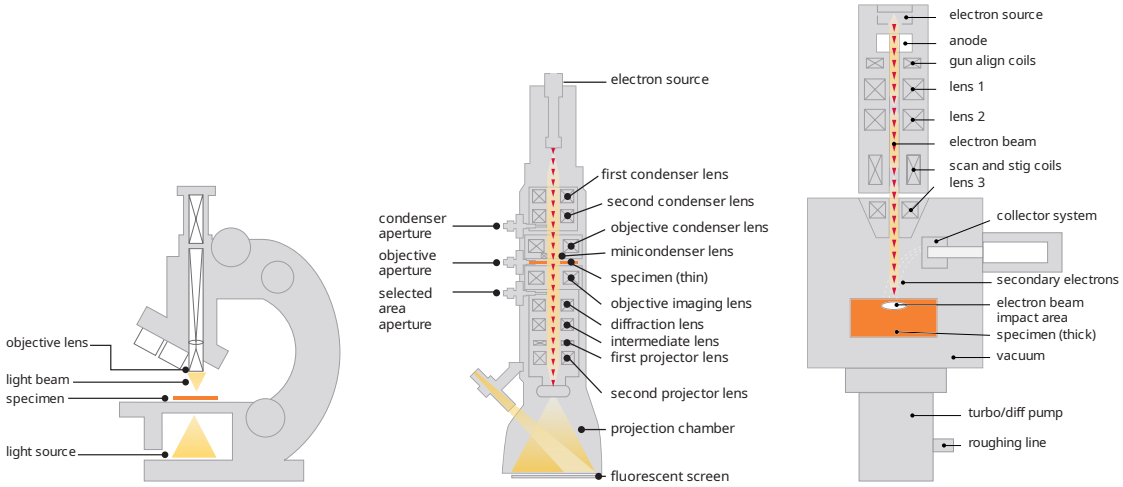
1.1 Electron Microscopy

1.1.1 Concepts and Principles

Classical light microscopy is limited by the wavelength of visible light, restricting its resolving power to a few hundred nanometers. Electron microscopy (EM), by contrast, uses accelerated electrons with wavelengths up to 100,000 times shorter than visible light, allowing imaging at nanometer and even sub-nanometer scales. This tremendous gain in resolution makes EM an essential tool across a wide range of scientific and industrial fields from discovery to production.

It underpins the characterization of nanomaterials, where linking structure to mechanical, electronic or optical properties is key to designing new functional materials. In energy applications, EM is central to the analysis of battery cathodes and other electrochemical materials, revealing degradation mechanisms that directly impact device lifetime and performance. In the semiconductor industry, EM-based metrology and defect analysis at the wafer level provide early access to critical process information, accelerating yield learning cycles and supporting faster time-to-market for advanced technologies. Finally, in the life sciences, EM transforms structural biology through atomic-scale visualization of macromolecular assemblies and cellular architecture, which is essential for understanding cellular function, disease mechanisms and drug design.

There exist two families of EM. The closest analogue to a light microscope is the Transmission Electron Microscope (TEM). In a TEM, a high-energy electron beam is transmitted through an ultra-thin specimen, typically below 100 nm for biological materials, under high vacuum. The transmitted electrons are focused by electromagnetic lenses to form an image down to atomic-scale resolution, even reaching sub-ångström details for state-of-the-art instruments. TEM enables the study of internal structures such as cellular organelles or crystalline lattices. This



(a) Light microscope (b) TEM (c) SEM

Figure 1.1: Comparison of the light microscope with SEM and TEM. Illustration used courtesy of Thermo Fisher Scientific [118].

performance comes at the cost of complex sample preparation, as the specimen must be electron-transparent.

The Scanning Electron Microscope (SEM) instead probes the surface of bulk specimens by scanning a focused electron beam across the sample and detecting emitted secondary or backscattered electrons. SEM produces high-resolution surface images, with modern instruments achieving resolutions down to approximately 0.7 nm. Its strength lies in revealing fine topographical details with a large depth of field. Compared to TEM, SEM typically requires less demanding sample preparation. Figure 1.1 compares the light microscope with TEM and SEM.

While TEM and SEM have historically been used to acquire two-dimensional images, many questions in both the material and life sciences require access to the three-dimensional organization of structures. This need has driven the development of *volume electron microscopy* (vEM), a family of techniques that extend TEM and SEM concepts to reconstruct continuous 3D volumes from series of EM images.

1.1.2 Volume Electron Microscopy

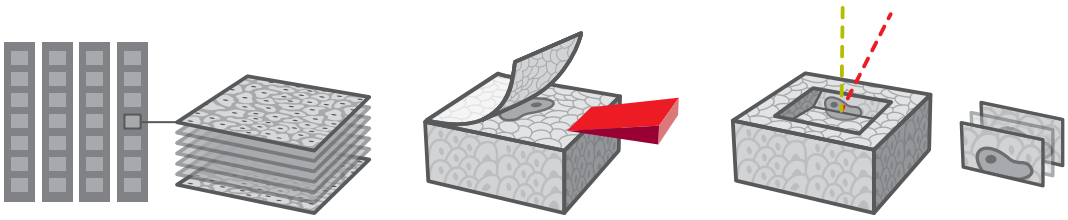
On the TEM side, volumetric information is most commonly obtained by *cryo-electron tomography* (cryo-ET) for biological samples. Thin vitrified lamellae or grids are imaged at multiple tilt angles, and tomographic reconstruction yields 3D volumes at nanometer or sub-nanometer resolution. This advancement was recognized with the 2017 Nobel Prize in Chemistry, awarded to Jacques Dubochet,

Joachim Frank, and Richard Henderson [47]. The transformative impact of cryo-EM has continued to be acknowledged, with the technique playing a crucial role in three subsequent Nobel Prizes: Chemistry 2020 for CRISPR-Cas9 gene editing, Physiology or Medicine 2021 for temperature and touch receptors, and Chemistry 2024 for protein design and structure prediction. However, the usable volume is constrained by the requirement for electron-transparent thickness, and preparing suitable cryo-lamellae is technically demanding. As a result, cryo-ET remains a highly specialized technique, primarily devoted to targeted nanoscale volumes in structural cell biology rather than the large, routinely acquired volumes considered in this work.

An alternative route to volumetric data relies on serial sections imaged by SEM, with three primary methods as illustrated in Figure 1.2. In *array tomography*, ultrathin sections are cut with a diamond knife and collected as ordered arrays on substrates such as silicon wafers. These arrays can then be imaged repeatedly: for instance, first in light microscopy to record fluorescence, and subsequently in SEM. Array tomography is particularly attractive when correlative imaging is required and when the ability to re-image the same physical sections is important. However, it introduces practical and computational challenges, including section handling, potential section loss or damage, and the need for robust 3D registration across large numbers of sections.

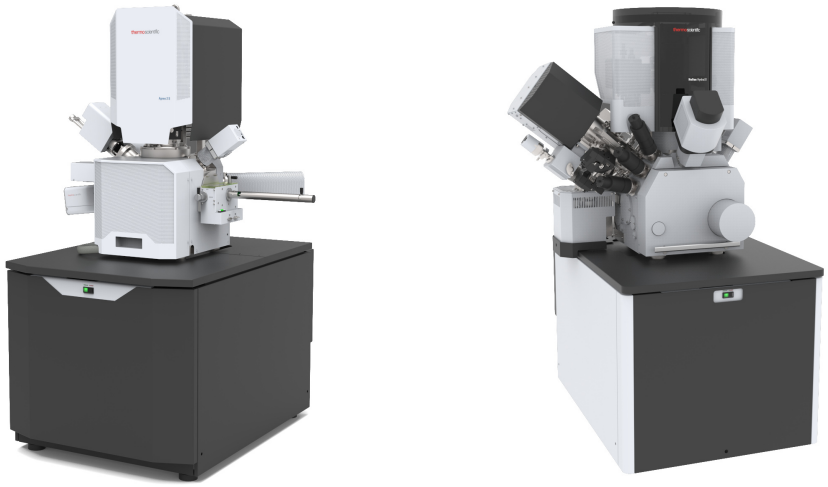
In this thesis, we are primarily concerned with vEM techniques based on SEM that naturally produce large 3D volumes on regular voxel grids and are widely used in both materials and life sciences. Two modalities dominate this landscape: *serial block-face SEM* (SBF-SEM) and *focused ion beam SEM* (FIB-SEM).

In SBF-SEM, a resin-embedded block is mounted inside the SEM chamber, and an integrated diamond knife removes thin slices from the block. After each cut, the freshly exposed block face is imaged by SEM. Because the specimen remains fixed with respect to the microscope, the resulting stack is intrinsically well registered along the cutting direction. SBF-SEM is well suited to acquiring large, contiguous



(a) Array tomography (b) Serial block-face (c) Focused ion beam

Figure 1.2: Schematic view of volume electron microscopy techniques. Images courtesy of Thermo Fisher Scientific.



(a) Volumescope 2 (b) Helios 5 Hydra

Figure 1.3: **(a)** Volumescope 2 SEM for serial block-face imaging system with multi-energy deconvolution. **(b)** Helios Hydra PFIB-SEM for large-volume milling and volumetric imaging. Images courtesy of Thermo Fisher Scientific

volumes up to $10^5 \mu m^3$ with moderate axial resolution, determined by the physical section thickness, typically greater than 15-20µm. Recent developments, such as multi-energy deconvolution [32], seek to overcome this limitation: for each physical slice, the block face is imaged at several accelerating voltages, each probing a different interaction depth. By deconvolving these energy-dependent signals, it becomes possible to infer a small stack of *virtual* sub-slices within each physical section and thereby improve the effective z-sampling beyond the mechanical section thickness, bringing SBF-SEM data closer to isotropy. Modern implementations such as the Volumescope 2 system (Figure 1.3a) integrate this multi-energy capability for enhanced axial resolution.

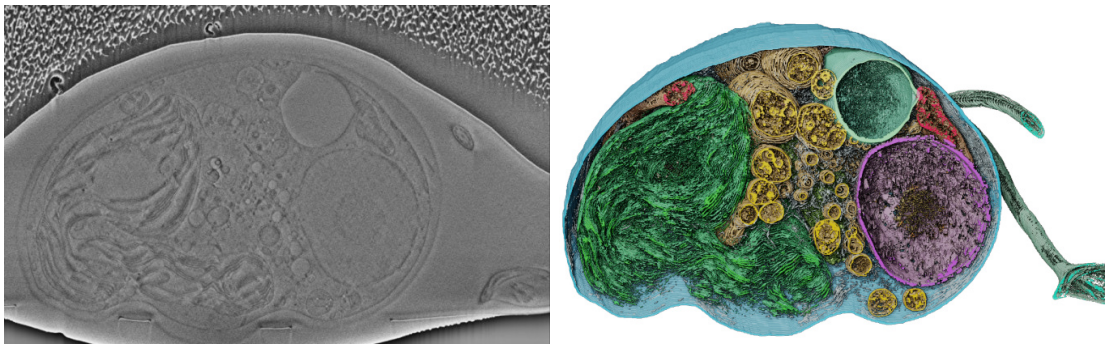
FIB-SEM implements a similar serial-imaging concept, but replaces mechanical sectioning with ion-beam milling. A focused ion beam removes a thin layer of material between successive SEM images. Unlike the virtual subslicing obtained by multi-energy deconvolution in SBF-SEM, FIB-SEM exposes a new physical surface at each milling step, providing direct control over the true slice thickness. This strategy avoids compression and crevasse artefacts and can deliver nearly isotropic voxels with nanometer-scale resolution in all three dimensions. FIB milling has become the standard thinning method for many high-resolution structural studies. More recently, alternative ion sources, plasmas or lasers have been introduced with the dual aim of increasing milling speed and reducing damage, further improving throughput. Systems such as the Helios 5 Hydra PFIB-SEM (Figure 1.3b) exem-

plify these advances, combining plasma ion sources with enhanced automation and instrument stability to enable routine vEM of significantly larger volumes than was previously feasible [127].

Across these SEM-based modalities, vEM now routinely produces volumetric datasets with nanometer-scale voxels over volumes ranging from subcellular regions to large tissue blocks. The corresponding data sizes frequently reach tens of gigabytes. Extracting useful information from such datasets, for the sake of visualization or measurement, critically depends on *segmentation*, the process of partitioning images or volumes into meaningful regions.

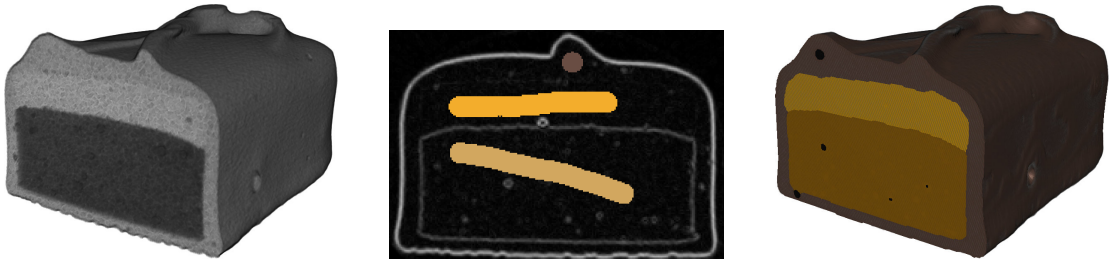
1.2 From Imaging to Segmentation

Segmented images or volumes underpin most quantitative image analysis workflows. Formally, image segmentation consists of partitioning an image or volume into meaningful regions that facilitate downstream tasks such as object detection, tracking, recognition or metrology. A wide variety of methods have been proposed over the years, ranging from simple thresholding to region-growing approaches, graph cuts, random walker algorithms, and more recently deep learning.



(a) Cryo-PFIB/SEM grayscale slice (b) Segmented volume

Figure 1.4: Reconstructed cryo-PFIB image and deep learning-based segmentation of *Chlamydomonas reinhardtii* [94].



(a) Original volume (b) User-defined markers (c) Segmented volume

Figure 1.5: Marker-based watershed segmentation of a chocolate bar CT volume.

In contemporary practice, deep learning has become the dominant approach to segmentation in many domains. Supervised networks trained on pixel-wise annotations can learn rich, task-specific features and yield highly accurate, dense predictions. This paradigm is illustrated by Figure 1.4 with a cryo-PFIB/SEM dataset of *Chlamydomonas reinhardtii*, where a U-Net-like architecture is used to produce a high-quality semantic segmentation of intracellular structures from grayscale volume data. Such learning-based approaches are particularly relevant for modalities with low and spatially variable contrast, where purely gradient-driven methods may be unstable.

However, such supervised approaches rely on the availability of large, carefully annotated training datasets, which are expensive and time-consuming to generate. In many practical situations, no labelled data are initially available. This is the case for new materials, novel imaging protocols, or proprietary industrial samples. In these cases, classical segmentation methods remain highly relevant. They can be used directly to obtain a usable segmentation, or as a tool to generate initial labels that will later be used to train a deep neural network.

Among these classical techniques, *watershed segmentation* is one of the most widely used methods. Introduced in the 1970s [34] as a segmentation tool within the framework of mathematical morphology and popularized in the 1990s by efficient algorithms [122, 14], it is now a standard component of many image analysis pipelines. Based on a topographic analogy, it interprets a gray-level image as a topographic surface: bright pixels form hills and dark pixels form valleys. A drop of water falling on such a surface flows along the path of steepest descent toward a regional minimum. The set of points that drain toward a single minimum forms a *catchment basin*. The set of points from which water could flow toward two or more distinct minima defines the *watershed lines*.

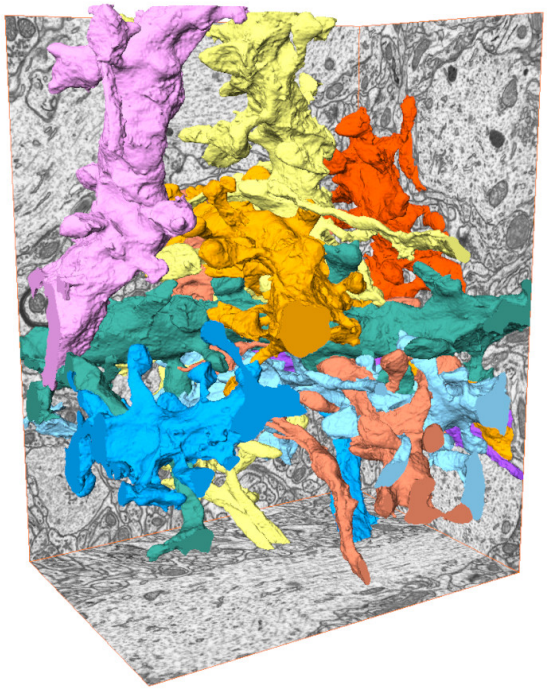


Figure 1.6: Example of interactive marker-based watershed segmentation of a nearly isotropic FIB-SEM mammalian cerebral cortex volume [62].

In practice, the so-called *marker-based watershed* is widely used. It consists of substituting the natural minima with user-defined markers thus associating one catchment basin with each marker. This allows prior knowledge about the objects of interest and background to be injected into the segmentation process. The watershed is typically computed on a gradient image, where edges between distinct regions correspond to high gradient values, while markers are placed to indicate the regions of interest. Figure 1.5 illustrates this idea on a simple X-ray CT volume of a chocolate bar: a few markers placed interactively in the different phases (nougat, chocolate and caramel) are sufficient to guide the watershed on the gradient and obtain a clean 3D partition.

In the example of Figure 1.6, a nearly isotropic FIB-SEM volume of mammalian cerebral cortex is segmented using interactive marker-based watershed: neurons are delineated by manually placing markers and propagating them with watershed. More generally, among the vEM modalities discussed above, marker-based watershed is typically not the method of choice for cryo-ET, where low signal-to-noise ratio and less predictable contrast can make gradient-based region growing unstable. In such settings, watershed is nevertheless frequently used as a post-processing step, for instance to refine probability maps and separate touching structures. This is also the case for array tomography, where strong anisotropy

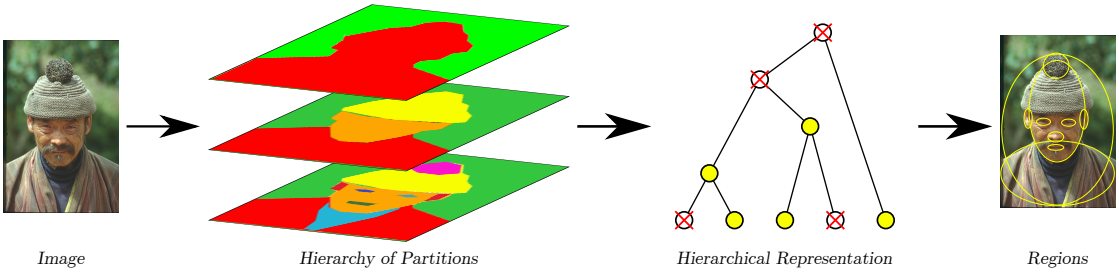


Figure 1.7: Example of a hierarchy of partitions of an image from the BSDS500 dataset [5], represented as a series of nested segmentations and an associated tree of regions.

and section-to-section variability can favor leakage or inconsistent basin boundaries. By contrast, watershed is often very effective with SBF-SEM and FIB-SEM. The near-isotropic resolution of produced volumes allows watershed to be applied in three dimensions without severe anisotropy artefacts leading to basin leakage between two physical sections. This use case is representative of such workflows on FIB-SEM data, where experts iteratively refine markers to obtain anatomically consistent 3D segmentations.

Beyond such interactive scenarios, marker-based watershed is also widely used in fully automatic or semi-automatic pipelines. In materials science in particular, markers are often derived algorithmically, for example from local maxima of distance or regional minima of smoothed images. These automatically generated markers are then used to drive watershed with minimal user intervention.

In modern mathematical morphology, watershed is embedded within the framework of *hierarchical segmentation* through an intermediate representation called the *Binary Partition Hierarchy* (BPH) [28]. Rather than producing a single partition, hierarchical methods represent an image as a series of nested partitions, organized from the finest to the coarsest. This yields a multiscale description in which regions can be selected or merged according to application-dependent criteria, avoiding the need to commit to a single segmentation scale. These hierarchical representations allow for multi-scale analysis and facilitates the selection of specific regions across scales eliminating the need for choosing a specific scale, as illustrated in Figure 1.7.

The continuous increase in resolution and field of view offered by modern vEM leads to ever larger volumetric datasets. Segmentation in this context has traditionally relied on experts manually tracing regions slice by slice; however, such manual approaches are clearly not scalable at these data volumes. Many algorithmic methods face similar limitations. In particular, watershed and its hierarchical extensions, while powerful for segmenting volumetric data and for bootstrapping annotations for deep learning, are inherently global and non-separable, and thus

become increasingly demanding in terms of memory and computation time as volumes grow.

1.3 Computational Challenges of Large Data

While advances in electron microscopy have revolutionized structural analysis, they have also accelerated the rate at which data is produced and its size, introducing significant computational challenges. The increasing sophistication of acquisition workflows directly contributes to this growth. For example, serial-section imaging pipelines such as the one described in the patent of Phelippeau et al. [92], require multiple high-resolution acquisitions per section and subsequent 3D alignment, dramatically amplifying storage and processing demands. Similar pressures emerge in downstream analysis steps: object detection in large tomograms has motivated the development of more scalable approaches such as *tensorial template matching* [3, 67], designed to overcome the rotational complexity bottleneck inherent to classical template matching. These examples illustrate a broader trend across the field: as data size expand, both acquisition and analysis pipelines increasingly depend on computational methods capable of scaling well beyond the limits of traditional processing.

A simple order-of-magnitude estimate illustrates how quickly these datasets reach the limits of standard workstations. A single 8k × 8k 8-bit slice contains about 64 million pixels, *i.e.*, $\approx 64\,\text{MB}$. At an acquisition rate of one slice per minute, this corresponds to $\approx 92\,\text{GB/day}$ of raw images. Even when working on a curated few gigabytes subset of such data, memory becomes a limiting factor on a typical workstation (64-256 GB RAM). Moreover, segmentation pipelines rarely operate on raw 8-bit intensities alone, but also require additional volumes such as gradients, probability maps, or label images, often stored in 16-bit or 32-bit floating point formats. As a result, the working memory footprint can easily grow by several times the raw data size.

When considering segmentation within an image analysis workflow, two primary challenges emerge as image sizes continue to increase:

1. **Extended computation time:** As the data size grows, the computation time required for processing can significantly increase. For example, in interactive segmentation tasks, refining markers for segmenting a tomogram can lead to delays of up to several minutes before the user sees the impact of their adjustments on the segmentation result. Such delays hinder iterative refinement and the overall user experience.

2. **Resource limitations and workflow failure:** The processing of such datasets often exceed the memory capacity of standard workstations, leading

to complete workflow failure without producing any segmentation result. This issue underscore the need for memory-efficient algorithms and scalable computational solutions.

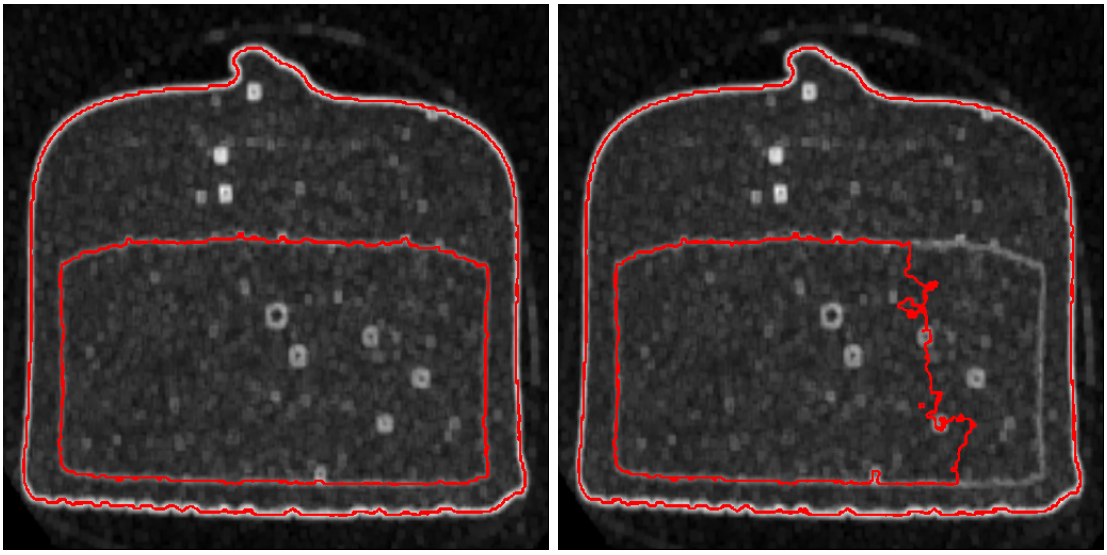
The first challenge, efficient computation for large data, has been extensively addressed in the literature through the development of new algorithms and parallelization/distributed strategies.

The second challenge, scalability in memory-constrained environments, can be tackled through distributed computing which distributes data across multiple computation nodes. In a single machine, the problem can be addressed through *out-of-core* algorithms, also referred as external memory algorithms. These methods are explicitly designed for situations where the data or intermediate objects cannot fit into main memory. They proceed by dividing the dataset into smaller, manageable chunks that are processed sequentially: each chunk is loaded into memory, computations are performed, and intermediate results are written back to disk. Finally, the stored partial results are aggregated to produce a result equivalent to the one obtained with the regular, *in-core* algorithm.

The challenges posed by memory-constrained processing of large-scale data have driven the development of specialized algorithmic approaches across multiple domains [123]. In image processing, methods such as the random walker algorithm have been adapted to large datasets [124]. In topographic analysis, scalable techniques for calculating watersheds on elevation data have been developed [7, 31]. Further contributions have come from terrain rendering [61] and linear algebra [119], demonstrating the widespread need for innovative solutions to handle massive datasets. These collective efforts highlight the importance of developing memory-efficient and scalable approaches, which are critical for processing the increasingly large volumes of data generated in modern workflows.

1.4 Objectives and Contributions

General out-of-core strategies provide a foundation for handling massive data volumes but the specific characteristics of some algorithms require tailored solutions. This is the case for watershed segmentation. Unlike many conventional image processing operations, such as median or Gaussian filtering, both watershed and binary partition hierarchy current algorithms are inherently non-separable. The current formulation rely on Kruskal's minimum spanning tree algorithm which requires a holistic view of the data. As a consequence, these methods are not amenable to straightforward divide-and-conquer strategies. Thus, any watershed-based workflow would then fail to produce a result when the image size or processing exceeds the available memory.



(a) Expected (b) Block-wise algorithm

Figure 1.8: **(a)** Comparison between the expected watershed outcome and **(b)** the naive block-wise algorithm result.

This limitation is illustrated in Figure 1.8, which shows an attempt to compute the watershed segmentation of a chocolate bar image using a naive block-wise strategy. Three markers were manually placed: one for the background, one for the chocolate and caramel and one for the nougat. While the expected result should clearly delineate the three distinct phases, the block-wise computation produces visible inconsistencies. These artifacts underscore the non-separable nature of the watershed and highlight the need for dedicated out-of-core strategy yielding results consistent with in-memory algorithms.

In this thesis, we address the scalability limitations of watershed segmentation by developing an out-of-core framework for its computation. Rather than working directly on the watershed formulation, we adopt the more general perspective of hierarchical representations and, in particular, the binary partition hierarchy. The BPH provides a unifying structure for a wide range of segmentation tools such as hierarchical watershed, quasi-flat zone hierarchies, constrained connectivity, etc. Then computing it in an out-of-core manner not only unlocks memory scalable watershed algorithms but also extends to this broader family of methods.

Our objective is therefore twofold. First, we aim to formalize and design an out-of-core algorithm for BPH construction on edge-weighted graphs under strict memory constraints. This requires rethinking classical in-memory algorithms in a way that they can operate on data that are partitioned into tiles, processed sequentially, and stored on disk. Second, we propose an out-of-core post-processing

of the BPH for scalable watershed-based segmentation, including watershed cuts and hierarchical watershed, so that these methods can be applied to images whose processing exceeds the available main memory.

The approach we propose is inspired by strategies developed in domains closely related to our problem. In particular, we build on the classical idea of emulating parallel algorithms sequentially in an out-of-core setting [23], and on the tile-based framework of Kazemier et al. [53], where both input and output are decomposed into chunks. By adapting and formalizing these ideas in the setting of BPH computation, we leverage a divide-and-conquer paradigm that respects out-of-core constraints at each step of the hierarchy construction. This leads to a practical and theoretically grounded framework for processing large-scale images within the limits of a single machine.

The organization of the manuscript is as follows:

Chapter 2 provides a state of the art in watershed and hierarchical segmentation, with a particular focus on their graph-based formulations around the binary partition hierarchy and minimum spanning tree. We first address the watershed transform and its main variants, then review the principal families of hierarchical representations, including component trees, hierarchies of partitions, and hierarchical watersheds, highlighting their relationships and their connections to BPH. It then examines existing work on scalable computation, such as streaming, parallel, and distributed algorithms for minimum spanning trees, component trees, and quasi-flat zone hierarchies.

Chapter 3 lays the mathematical foundation for this research by introducing key concepts and formal definitions. It begins with fundamental graph theory notions, which are essential for understanding hierarchical image representations. The binary partition hierarchy is then formally defined, providing a structured framework for image partitioning. The chapter also explores the (hierarchical) watershed transform, emphasizing its formulation on edge-weighted graphs and its close relationship with BPH. These theoretical foundations serve as a basis for the methodologies developed in the subsequent chapters.

Chapter 4 introduces the notion of *Distribution of the Binary Partition Hierarchy*. We define an algebraic framework to formalize out-of-core computation and describes data structures tailored for efficient hierarchical processing. Based on three core operations, namely *select*, *join*, and *insert*, we give two high level calculi to compute the distribution of the BPH alongside with efficient algorithms for the core operations. The first method supports generic partitions, allowing flexible data distribution, while the second focuses on causal partitions, ensuring strict out-of-core constraints. The advantages

and limitations of these approaches are analyzed through theoretical properties and an experimental study comparing both computation schemes.

Chapter 5 builds upon the previous chapter to extend the out-of-core computation framework to watershed segmentation. It begins by discussing the distribution of attributes, then watershed computation and finally its hierarchical counterpart. Several out-of-core attribute computation algorithms are then presented alongside with the implementation of out-of-core watershed segmentation and its hierarchical extension.

Chapter 6 discusses the practical way to implement this framework considering superpixels, defined as quasi-flat zones, instead of pixels. We describe the methodology to apply the introduced calculi to a region adjacency graph instead of pixel adjacency graphs, and we provide experimental results that evaluate the impact of this superpixel-based strategy on time and memory consumption.

Chapter 7 presents an incremental in-core marker-based watershed algorithm that leverage the hierarchical representation to efficiently exploit the inherent causality of the interactive segmentation process. This method enables updating the segmentation in a time proportional to the number of pixels updated when adding or removing markers. The chapter also introduces a parallel strategy for minimum spanning forest labeling. Both methods are evaluated through experimental results.

Chapter 8 concludes this work giving a synthesis of the contributions before detailing perspectives.

Publications

The research contributions presented in this thesis have been disseminated through the following peer-reviewed publications:

Conference Proceedings

- J. Lefèvre, J. Cousty, B. Perret, and H. Phelippeau. Join, select, and insert: Efficient out-of-core algorithms for hierarchical segmentation trees. In *Discrete Geometry and Mathematical Morphology*, pages 274–286. Springer, 2022. 10.1007/978-3-031-19897-7_22

- J. Lefèvre, J. Cousty, B. Perret, and H. Phelippeau. Out-of-core attribute algorithms for binary partition hierarchies. *Discrete Geometry and Mathematical Morphology*, pages 298–311, 2024. 10.1007/978-3-031-57793-2_23

- Q. Lebon, J. Lefèvre, J. Cousty, and B. Perret. Interactive segmentation with incremental watershed cuts. *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, page 189–200, Nov. 2023. 10.1007/978-3-031-49018-7_14

Journal Publications

- J. Lefèvre, J. Cousty, B. Perret, and H. Phelippeau. Out-of-core algorithms for binary partition hierarchies. *Journal of Mathematical Imaging and Vision*, 67(2):18, 2025. 10.1007/s10851-025-01234-0

- Q. Lebon, J. Lefèvre, J. Cousty, and B. Perret. Incremental watershed cuts: Interactive segmentation algorithm with parallel strategy. *Pattern Recognition Letters*, 189:256–263, Mar. 2025. 10.1016/j.patrec.2024.12.005

Chapter 2

State of the Art on Morphological Hierarchies

2.1 Introduction

Segmentation is a central task in image analysis, and numerous methodologies have been developed to extract meaningful regions from complex data. Among these, the watershed transform has played a historical and influential role. State of the art watershed definition within a framework organized around two main structure: Binary Partition Hierarchy (BPH) and Minimum Spanning Tree (MST).

This chapter presents the conceptual background required to understand the hierarchical segmentation framework adopted in this thesis, with a particular emphasis on the BPH, the central representation used in the remainder of this work. Our goal is to highlight the key ideas, models, and connections that place BPH within the broader landscape of hierarchical and graph-based segmentation methods.

We begin by briefly introducing the watershed transform. We then survey the major families of hierarchical segmentation structures, including component trees, hierarchies of partitions, and hierarchical watersheds, showing how these representations relate to one another. Finally, we turn to existing work that addresses computational challenges associated with processing large-scale data, including streaming, parallel, and distributed algorithms for structures closely related to the BPH. This places our own problem, efficient out-of-core computation of BPH, within a clear methodological and algorithmic context.

2.2 Watershed Segmentation

The watershed transform is one of the most prominent result of mathematical morphology and image segmentation. Its strength lies in its intuitive topographic interpretation, its solid mathematical foundations, and its versatility across discrete formulations. Although numerous variants and algorithms have been proposed over the past decades, they all share a common objective: partitioning an image into meaningful catchment basins separated by watershed lines. In this section, we first recall the historical origins of the watershed and its topographic intuition, before reviewing the main computational frameworks and paradigms in which it has been formalized. We then discuss marker-based approaches, which address the well-known issue of over-segmentation and are central to many practical applications.

2.2.1 Historical Background and Topographic Intuition

The watershed was initially studied in the field of topography to delineate drainage basins and ridges by analyzing the natural flow of water across landscapes. This approach holds significant implications not only for understanding geographical features but also for addressing geopolitical boundaries and resource allocation. We can see example of the watersheds of the North American continent on Figure 2.1. We can clearly see that watershed lines separate the catchment basins of the Atlantic, Pacific and Arctic oceans, *i.e.*, areas where the water flows to the Atlantic, Pacific or Arctic ocean.

Its mathematical foundations date back to the 19th century, with significant contributions from Maxwell [69] and Jordan [51]. The watershed transform was later introduced as an image segmentation tool by Digabel and Lantuejoul [34] and has since become a cornerstone technique in mathematical morphology and computer vision. Popularized by Vincent and Soille [122], this technique gained widespread adoption across diverse domains, including computer vision, medical imaging, and remote sensing, owing to its effectiveness in partitioning data into meaningful regions.

The watershed principle is inspired by the analogy between a gray-level image and a topographic relief. The gray level of each pixel represents an elevation: darker regions correspond to valleys and basins, while lighter regions correspond to peaks and crest lines. This translation from grayscale to elevation is illustrated on Figure 2.2.

On such a landscape, a drop of water naturally flows downhill along the path of steepest descent until it reaches one of the local minima of the surface. All points whose water would end up in the same minimum form what is called a *catchment basin*. The boundaries separating adjacent basins, where water could potentially



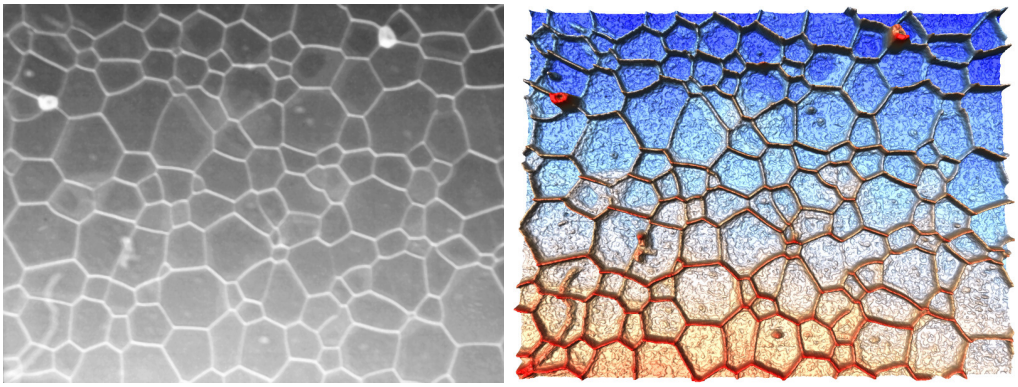
Figure 2.1: Watersheds of the North American continent. The watershed lines separate three catchment basins: the Atlantic, Pacific and Arctic oceans.

flow toward different minima, are the *watershed lines* and define the segmentation contours.

With the time, watershed was defined in many different ways ranging from heuristic region-growing approaches to more rigorous formulations based on graph cuts. In this section, we will focus on the latter definition and in particular on watershed definitions in edge-weighted graphs.

2.2.2 Frameworks and Principles

The watershed transform has been extensively studied in the literature, leading to the development of numerous algorithms and variants. Here is a non exhaustive and unordered selection of watershed variants: watersnakes [84], stochastic watershed [4], exact stochastic watershed [66], robust stochastic watershed [105], stochastic permutation ordering watershed [63], random walker watershed [95], hierarchical watershed, watershed-cut [26], oriented watershed [5], directed probabilistic watershed [101], waterfall [13], viscous watershed [121], topological watershed [12], incremental watershed [55], power watershed [24], mutex watershed [126],



(a) Uranium oxide image (b) Topographic relief

Figure 2.2: **(a)** Original image of uranium oxide **(b)** Corresponding topographic relief where watershed segmentation can be applied.

adaptive watershed [115], compact watershed [82], toboggan-based watershed [60], ultrametric watershed [78] and streamed watershed [48].

Most of these variants adapt the watershed algorithm to address specific problems or impose additional constraints. However, it is more insightful to focus on the different discrete frameworks in which the watershed has been defined. The two most prominent frameworks are based on graph theory: vertex-weighted graphs and edge-weighted graphs.

The most widely used algorithm, introduced by Vincent and Soille [122], follows the *flooding principle*. This paradigm can be visualized as punching holes in the topographic surface at the locations of local minima. As the surface is progressively submerged in water, the basins fill up starting from the minima. The flooding continues and watershed lines are formed when basins meet. This approach is implemented in major image processing libraries such as OpenCV [18], ImageJ/Fiji and commercial software like Amira/Avizo [113]. These algorithms are based on vertex-weighted graphs where each vertex correspond to a pixel and is weighted regarding its grayscale value. Algorithms based on this principle typically utilize hierarchical queues to efficiently manage the flooding process. Watershed are then defined as the set of pixels where flooding from different minima would converge.

An alternative paradigm is the *drop of water principle* [26] we described in the previous section. A drop of water flows downhill along the surface by the steepest descent until it reaches one regional minima, delineating catchment basins, or can reach distinct regional minima, delineating watershed lines. This second principle has been the subject of much theoretical work, and it has been shown to hold in the context of edge-weighted graphs. In this framework, the image is represented as a graph where vertices correspond to pixels and edges linking neighboring pixels

are weighted based on a dissimilarity measure between neighboring pixels. The dissimilarity measure weighting edges can selected based on the application, and can be based on the image intensity, gradient, or any other feature that can be computed from the image data. State-of-the-art methods use gradient computed with learning-based methods, such as convolutional neural networks [112, 44].

Cousty et al. [26] addressed the formalization of watershed as a combinatorial optimization problem on edge-weighted graphs. The *watershed-cut* is then defined as a partition of the graph into minimum spanning forests, where each tree in the forest corresponds to a catchment basin associated with a regional minimum.

Although the watershed framework provides a clear and intuitive way to partition an image, its reliance on all regional minima makes it highly sensitive to noise and small intensity fluctuations. Real images often contain many insignificant minima, which in turn produce an excessive number of regions. This motivates the need for mechanisms that control or guide the segmentation process, leading to the development of marker-based watershed methods.

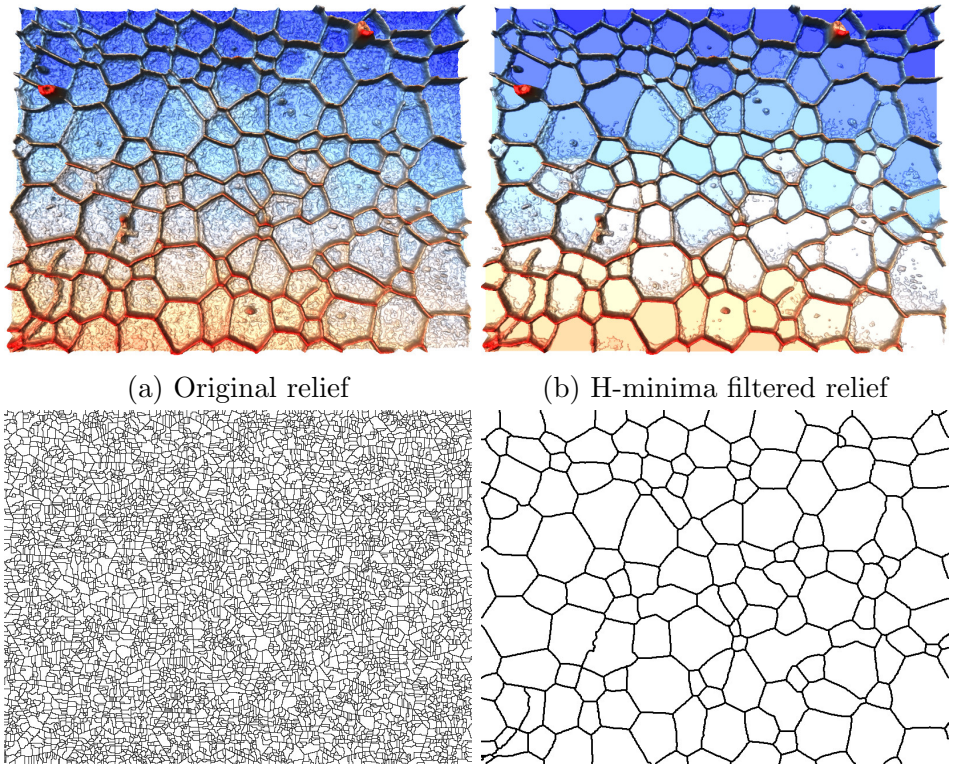
2.2.3 Marker-Based Watershed

Using the flooding principle to draw an analogy, marker-based watershed methods address the sensitivity of the watershed to spurious minima by explicitly controlling the set of regions from which flooding begins. Instead of relying on all regional minima, the user or an automated procedure selects a smaller, meaningful set of minima, referred to as *markers*, that guide the formation of catchment basins.

A first approach to mitigate this issue is to pre-process the input image to reduce the number of minima. Several examples of this methodology are given by Beucher and Meyer [14] in their seminal work. This process of *homotopy modification* consists in modifying the input image in a way that the original set of minima is replaced by a new set of minima. The intuitive idea can easily be formulated with the flooding principle. Instead of boring holes in place of any minima, the user pierces holes at desired location, *e.g.*, one marker in the object of interest and another one in the background. Original minima are filled and the watershed is performed on this new surface.

The *h-minima* transform [109] is a common tool for this purpose: it removes minima whose depth is below a threshold h, implemented by subtracting h to the input image followed by a dual reconstruction. The result of such a filtering is illustrated on Figure 2.3. Note that at this point, there is not an explicit set of minima. Which could however be obtained with the *extended h-minima*, *i.e.*, the minima of the filtered image. However, too aggressive filtering may remove relevant minima, causing the loss of important structures.

A more flexible approach is to allow the user to specify markers interactively using tools such as brushes or magic wands. An example is shown in Figure 2.4,



(a) Original relief (b) H-minima filtered relief

(c) Watersheds of (a) (d) Watersheds of (b)

Figure 2.3: **(a)** Original topographic relief exhibiting many spurious minima **(b)** Homotopy modification with h-minima transform (h=20) **(c)** Watersheds of the original relief **(d)** Watersheds of the h-minima filtered relief.

where placing one marker per cell corrects the over-segmentation.

The marker-based watershed is largely tackled by both Vincent and Soille [122] and Beucher and Meyer [14] algorithms. Note that the homotopy modification is not necessary as these algorithms begin the flooding from the set of markers.

Regarding watershed-cut, Cousty et al. [26] proposed an efficient linear-time algorithm for minimum spanning forest on edge-weighted graphs. Unlike traditional methods, this algorithm bypasses the need for preliminary minima computation or advanced data structures like hierarchical queues. However, this efficiency comes with a limitation: the algorithm does not natively support marker-based operations, which are essential for applications such as interactive segmentation where control over specific regions is needed.

However, the watershed cut induced by any set of markers is exactly the minimum spanning forest relative to those markers. Thus, an classical MST algorithms such as Kruskal's or Prim's, combined with Tarjan's Union-Find data structure,

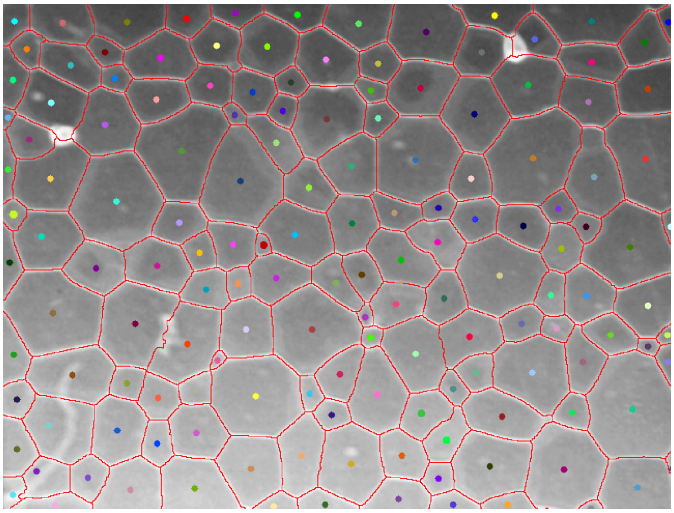


Figure 2.4: Watershed segmentation of the uranium oxide image given the set of disk markers.

yield quasi-linear-time marker-based watershed cuts.

To combine the efficiency of linear-time watershed computation with the flexibility of marker-based segmentation, hybrid approaches have been proposed. These methods first apply a fast watershed to produce superpixels, then construct a region adjacency graph where each node represents a basin and edges correspond to MST edges linking distinct basins. Marker-based MSF algorithms are applied on this reduced graph [73], enabling precise marker-controlled segmentation while preserving computational efficiency.

More recently, learning-based approaches have emerged to reduce the reliance on manually placed markers. Barbosa da Fonseca et al. [8] introduced a differentiable marker-based segmentation layer that enables end-to-end learning of marker proposals within neural networks. By integrating marker generation directly into the training objective, this method automatically produces task-relevant markers.

2.3 Hierarchies of Segmentation

The Binary Partition Hierarchy, the central structure studied in this work, belongs to the broader family of morphological hierarchies.

A single image can typically be segmented at multiple levels of detail, with the appropriate level depending on the underlying task. However, choosing a single scale is often problematic, as objects of interest may appear at different spatial resolutions. This raises a natural question: *why commit to a single scale at all?* Instead of fixing a segmentation scale a priori, hierarchical representations encode

a sequence of nested regions or partitions, ranging from the finest to the coarsest. Once computed, this structure simultaneously captures all relevant scales, enabling multiscale analysis without repeated segmentation.

In this section, we will review the two main families of hierarchical representations: *component trees* (or *hierarchies of partial partitions*) and *hierarchies of partition* (or *hierarchies of complete partitions*). Then, as en extension of the previous chapter, we will then discuss *hierarchical watershed*, exhibiting its connections with the BPH.

2.3.1 Component Trees

One important family within morphological hierarchies is that of *component trees*, which are generally constructed on vertex-weighted graphs. Component trees have been extensively studied in mathematical morphology [79, 90, 16]. Their most prominent variants are the max-tree and its dual, the min-tree [100, 50]. These structures provide an efficient framework for computing connected filters, which modify an image without creating new contours by operating on flat zones [19].

Max-trees represent the inclusion relationship between connected components of upper level sets of an image. For any gray level λ, the upper level set is defined as the set of pixels whose intensity is greater than or equal to λ. Each node in the max-tree corresponds to a connected component of an upper level set, and the tree structure captures how these components are nested within each other as the threshold λ decreases. Thus, leaves of this tree correspond to local maxima of the image, whereas the root represents the entire image domain as illustrated in Figure 2.5. The min-tree is defined analogously, but on lower level sets, capturing inclusion relationships between connected components with intensity values less than or equal to a given threshold.

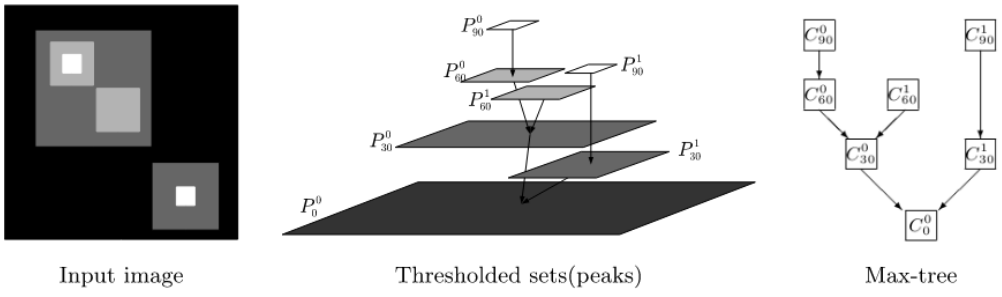
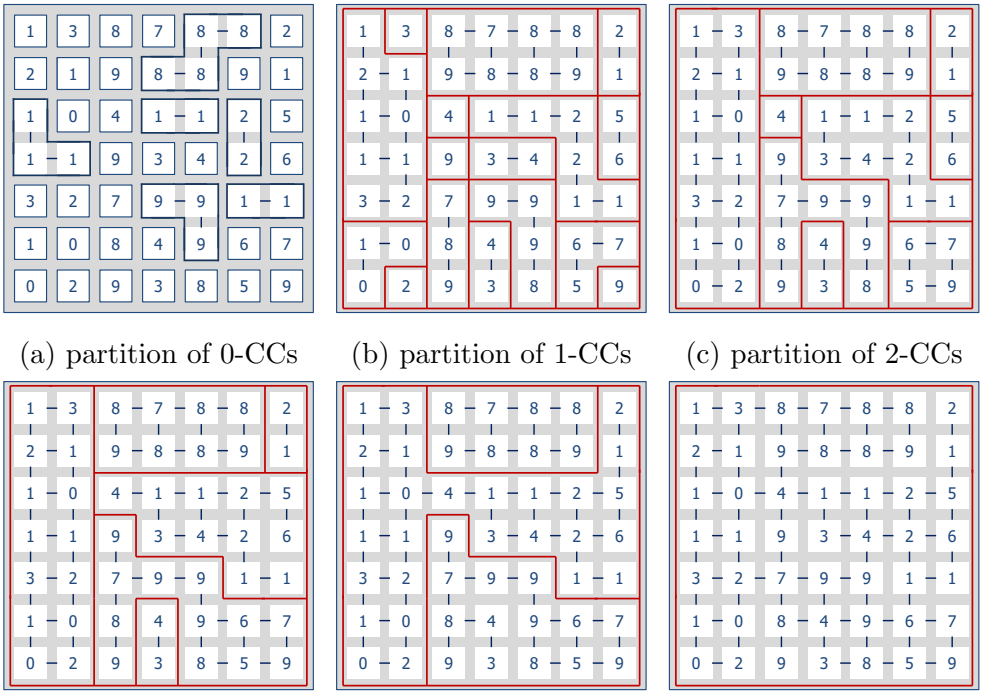


Figure 2.5: A simple gray-scale image, the foreground components of each threshold set, also know as peak components, and the resulting component tree, which is referred to as a max-tree in this case. Courtesy of Gazagnes and Wilkinson [39].

This family also includes the tree of shapes [77], which can be interpreted as a tree describing the inclusion structure of the image's level lines. In this representation, leaves correspond to local extrema, allowing the tree of shapes to capture both bright and dark structures simultaneously. It is closely related to the min-tree and max-tree and can, in fact, be computed by combining them. Recent work has extended this structure to support topological analysis of gray-scale images, giving rise to the topological tree of shapes [88, 70, 71].

2.3.2 Hierarchies of Partition



(a) partition of 0-CCs (b) partition of 1-CCs (c) partition of 2-CCs

(d) partition of 3-CCs (e) partition of 4-CCs (f) partition of 5-CCs

Figure 2.6: Partitions of the quasi-flat zones hierarchy using absolute difference between 4-adjacent pixels as dissimilarity measure. Courtesy of [37].

A second family of hierarchical representations are the *hierarchies of partition*. Often built on edge-weighted graph, each scale correspond to a complete partition of the space. In this family, regions are iteratively merged together following a given order.

Among these, the structure most closely related to what precedes is the *quasi-flat zones hierarchy* (also referred to as α-trees) [99]. Each scale k correspond to the partition of the graph into connected components where the weight of the edges

inside each component is lower than equal to k. An example of such hierarchy is given in Figure 2.6 where each edge $\{x, y\}$ is weighted by the absolute difference of intensity between pixels x and y. As k increases, regions merge together forming coarser partitions. Is has be shown by Cousty et al. [28] that this structure is equivalent to the min-tree of the MST of the edge weighted graph.

Another prominent example is the *binary partition tree* (BPT) [98, 96]. Starting from the finest partition, where each pixel is a singleton region, pairs of neighboring regions are iteratively merged based on a specific criterion, such as color similarity or texture homogeneity. This process continues until all regions are merged into a single segment.

The *binary partition hierarchy by altitude ordering* (BPH) [28, 81] is a particular case of the BPT close to single-linkage clustering. Specifically, in the BPT, region similarity must be computed at each merging step, whereas in the BPH, regions are merged by following a total ordering on edge weights. This makes the BPH both conceptually simpler and computationally advantageous.

The BPH exhibits strong connections with the minimum spanning tree (MST). Actually, the BPH can be seen as the hierarchy built during the execution of Kruskal's algorithm for computing the MST. Each time an edge is added to the MST, the two regions in which its endpoints lie are merged, forming a new region at a higher level in the hierarchy. This process continues until all vertices are connected. As Kruskal's algorithm insert edges in increasing order of weight, the BPH is built by altitude ordering. Leveraging this connection, Najman et al. [81] gave an efficient algorithm to build the BPH in quasi-linear time, relying on Tarjan union-find [116].

The BPH and associated minimum spanning tree, are pivotal for various hierarchical segmentation techniques. Notably, they have been shown [28, 81] to efficiently support the computation of quasi-flat zone hierarchies, constrained connectivity [110] and the watershed along with their hierarchical variations [74, 80, 28]. Recent works has also explored the behavior of combinations of hierarchical watersheds, which happen to be superior to their individual counterparts [102, 103]. In the next section, we will focus on hierarchical watershed.

2.3.3 Hierarchical Watershed

Continuing the analogy between a grayscale image and topographic relief, the watershed we discussed in the previous sections associates a catchment basin with each local minimum, from the largest ocean to the smallest puddle. However, once filled, the smallest puddle will drain into a larger one, then into a stream, a river, and finally into an ocean. In this way, the catchment basin associated with the small puddle is included in several basins associated with larger landforms. This naturally leads to a hierarchical representation of the topographic surface, where

each level corresponds to a different scale of catchment basins.

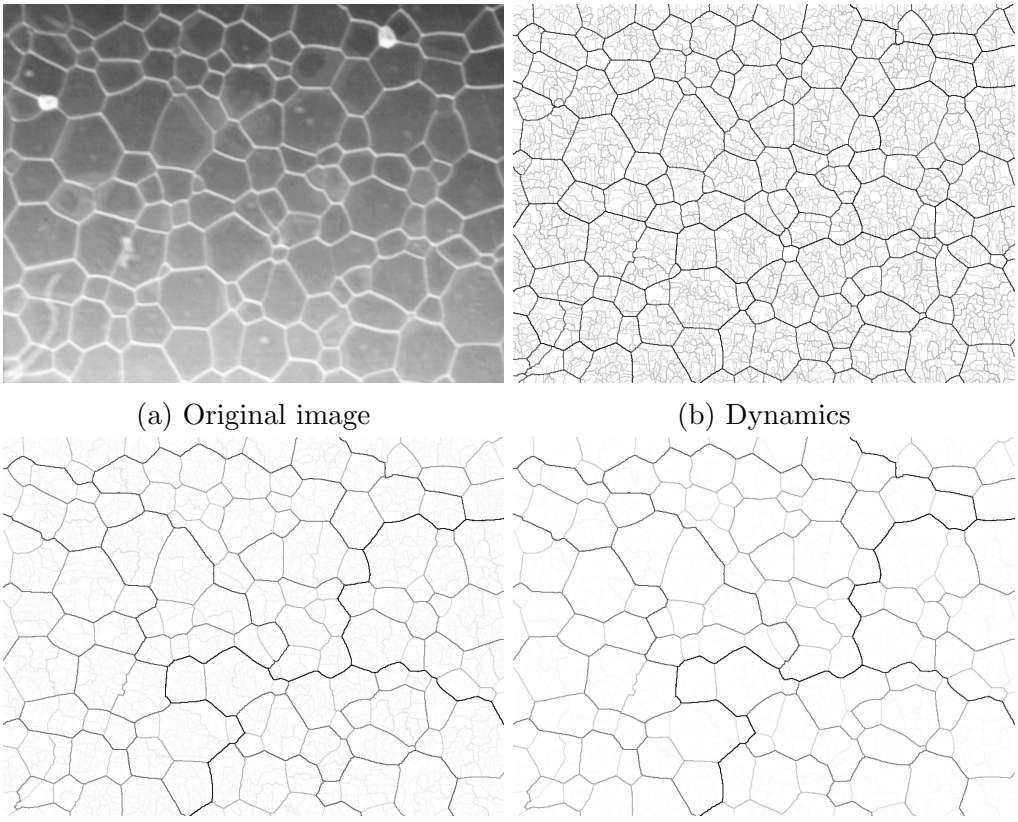
Going back to image analysis, the hierarchical approach of watershed was studied by Najman and Schmitt [80] and Beucher [13]. Later, a formalization of hierarchical watershed was given for edge-weighted graph as a series of nested watershed cuts [25, 28]. The intuition behind this definition is as follows.

The first scale of the hierarchy corresponds to the finest watershed segmentation, where each regional minimum defines a catchment basin. As mentioned earlier, this segmentation is often too detailed for practical applications, as it captures all local variations in the image. To obtain coarser segmentations, we can simplify the topographic relief by progressively filling its minima while keeping upper levels unchanged. At each stage of this simplification, a new watershed segmentation is produced, which is coarser than the previous one. The sequence of these segmentations naturally forms a hierarchy, where boundaries may disappear as the level of simplification increases, but no new boundaries are ever introduced. We will give a formal definition of hierarchical watershed in Section 3.

The order in which minima are filled determines the structure of the resulting hierarchy. Thus, we associate a measure of importance to each minimum. A straightforward approach is to rely on user's input. However, hierarchical watershed is more commonly built upon geometric and topological criteria such as area, height or volume. Such connected filtering operator based on increasing attributes has been the subject of extensive study in mathematical morphology [19], particularly in the context of component trees.

A well-known attribute associated with extrema is dynamics [43], which serves as a contrast estimator. This measure associates a regional minimum to the minimal height required to ascend in order to reach a deeper regional minimum. Dynamics is frequently used in conjunction with watershed segmentation to filter input images, selecting the most prominent minima based on their dynamics. Notably, recent work by Boutry et al. [17] has demonstrated the equivalence between dynamics in mathematical morphology and persistence in persistent homology. Building on the basis of Grimaud [43], Vachier and Meyer [120] introduced the concept of extinction values, which redefines dynamics as the extinction value of a minimum according to its height in the topographical relief. This formalism not only connects dynamics with extinction but also paves the way for extending the concept to other attributes, such as area, volume, topological height, or bounding boxes [106].

In order to visualize the effect of different attributes on hierarchical watershed, Figure 2.7 displays saliency maps derived from hierarchical watershed using various attributes to order the minima. The saliency maps, introduced by Najman and Schmitt [80] under the name of *geodesic saliency*, allows for the visualization of the entire hierarchy in a single image. Each boundary in the saliency map is



(a) Original image (b) Dynamics

(c) Area (d) Volume

Figure 2.7: Saliency maps obtained from hierarchical watershed using different attributes to order the minima.

weighted by the level at which it disappears in the hierarchy, with more prominent boundaries (darker ones) corresponding to more significant features in the image. Cousty et al. [29] provided a bijection between saliency maps and hierarchies based on quasi-flat zones, which includes hierarchical watershed.

The relation between BPH and hierarchical watershed is established by Cousty et al. [28] using the concept of extinction values and persistence. Once the BPH is computed, it can be post-processed to obtain hierarchical watershed with a linear time algorithm [81].

This hierarchical representation have found applications in diverse domains, including biological imaging [9], composite material analysis [108], historical building characterization [87], rock instability detection [54], and remote sensing [52, 65].

2.4 Prior Work on Scalable Graph and Tree Structures

To conclude the previous sections, it is worth recalling that the binary partition hierarchy serves as a versatile intermediate representation with numerous applications, ranging from (hierarchical) watershed construction to quasi-flat zone hierarchies and constrained connectivity. In the standard setting, the BPH can be computed efficiently using Najman et al. [81] algorithm. However, this efficiency relies on the assumption that the entire image graph fits in main memory, a constraint increasingly violated in modern large-scale imaging workflows.

To the best of our knowledge, the specific problem of computing the BPH under memory constraints has not been explicitly addressed in the literature. Nevertheless, several adjacent research areas offer promising methodological insights. Parallel and distributed strategies frequently rely on divide-and-conquer principles to manage large datasets, while the hierarchical analysis community has devoted considerable attention to the scalable construction of related structures, most notably component trees. These works offer valuable insights into scalable hierarchical representations.

In the following sections, we review three key areas relevant to our study: streaming algorithms for minimum spanning tree computation, existing approaches for parallel and distributed component tree construction, and methods for parallel quasi-flat zone hierarchy computation.

2.4.1 Streaming Minimum Spanning Tree

The *Minimum Spanning Tree* is a fundamental structure in hierarchical segmentation, widely used to efficiently compute partitions in methods such as watershed-cut, hierarchical watershed, quasi-flat zone hierarchy, and constrained connectivity. However, traditional MST-based segmentation approaches assume that the entire image is available at once, an assumption that does not hold in large-scale applications. In domains like remote sensing, medical imaging, and video processing, image data is often acquired incrementally due to memory constraints, necessitating an approach that can dynamically update the MST as new data arrives.

To address this challenge, Gigli et al. [40] proposed a method for MST computation in streaming scenarios. Formally, consider an image I divided into k sequentially acquired chunks $\{B_1, B_2, \ldots, B_k\}$, where B_1 arrives first. The objective is to leverage the MST computed on B_1 to efficiently construct the MST of $I = B_1 \cup B_2 \cup \cdots \cup B_k$ as new chunks arrive, without recomputing it from scratch at each step. This ensures an *incremental* update mechanism, where for each new chunk B_i with $i \geq 2$, the MST is updated while preserving previously computed

structures.

To handle such streaming data, dedicated algorithms have been developed to compute the union of two graphs with a non-empty intersection. The proposed method ensures an adaptive MST representation by partitioning it into two distinct components:

- **Stable MST edges**: These represent regions of the MST that remain unchanged as new data is introduced.

- **Unstable MST edges**: These correspond to regions that may be modified when additional data is processed. They are updated incrementally until they reach stability.

Unstable edges are those that belong to simple paths linking two vertices in the *frontier*, *i.e.*, vertices belonging to the 1 pixel overlap between the processed and incoming image chunks. To further optimize memory usage, stable edges can be stored on disk, ensuring that only unstable regions and the current chunk's graph representation are kept in memory. This significantly reduces the computational footprint while allowing efficient MST updates.

Gigli et al. [40] extensively demonstrated applications of this streaming MST framework to various segmentation methods, including quasi-flat zones, watershed-cuts, and constrained connectivity. The incremental nature of the MST update process allows these segmentation techniques to be adapted efficiently.

Taking the example of watershed segmentation, an initial watershed can be computed on the processed chunks. In the same fashion as edges, connected components are classified as *unstable* if they have a non empty intersection with the frontier or *stable* otherwise. As new data arrives, segmentation updates can be restricted to vertices of the new chunks and previously unstable segments, avoiding the need for a complete recomputation. This enables an *out-of-core* approach to segmentation, where only a subset of the data is processed at any given time, ensuring computational feasibility for large-scale datasets.

While the proposed method offers a scalable and memory-efficient solution for MST-based segmentation, it does not inherently support the *streaming construction of hierarchical structures* such as the Binary Partition Hierarchy. Constructing such hierarchical representations would require loading the entire MST into memory before further processing, which contradicts the out-of-core constraint. Addressing this limitation would require new strategies to maintain hierarchical relationships incrementally while ensuring efficient memory management.

2.4.2 Component Trees

As presented in Section 2.3.1, component trees are hierarchical structures that have been the subject of numerous studies in mathematical morphology. Among them, max-trees and their dual representation min-trees, have been extensively studied, particularly in the context of parallel and distributed computing. While early research primarily focused on reducing computational time, recent advances highlight the importance of optimizing inter-process communication, which is crucial for efficiently handling large-scale datasets.

A vast body of research exists on component tree computation, with numerous algorithms developed over the years. A comprehensive review by Carlinet and Géraud [21] classifies these algorithms into three main categories: immersion-, flooding-, and merge-based methods.

Among these, merge-based algorithms are particularly well-suited for parallelism and distributed computing, as they naturally align with the *map-reduce* paradigm. Their general workflow follows these steps:

1. Split the data into chunks;
2. Compute a *partial component tree* for each chunk; and
3. Merge the partial component tree two-by-two in order to form the final global component tree.

Both the *map* (local tree computation) and *reduce* (merging of trees) steps are inherently parallelizable, making them well-suited for implementation on shared-memory systems [68, 86, 125] and more recently with massively parallel GPU algorithm [15]. However while exhibiting good scalability with the example of near linear speed-up on CPU [125], these methods are limited by the memory size of the shared memory system.

In the context of large-scale data processing, distributed-memory approaches are particularly compelling, as they effectively address memory constraints and enable scalability across multiple computing nodes. This approach has been successfully applied to component tree computation, with notable contributions from Kazemier et al. [53] and Götz et al. [41].

Both approaches are based on the same divide-and-conquer strategy where each computing node is responsible for computing the partial component tree of the assigned chunk. The main difference between the two approaches is the output of the computation. While Götz et al. [41] output is equivalent to the one computed with share memory systems, Kazemier et al. [53] further adhere to the divide-and-conquer paradigm by not only splitting the data into chunks but also the output data. Expected result is no longer a single monolithic component tree introducing

the notion of *Distributed Component Forest* (DCF). Given that a component tree can be 20 to 40 times larger than the input image [39], a scalable representation is essential to manage memory efficiently.

The DCF addresses this challenge being a collection of sub-trees of the component tree, each associated with a single chunk. These sub-trees are designed such that filtering operations can be performed independently on each tree, while still yielding the same results as filtering the full component tree. This paradigm shift not only reduces memory requirements but also simplifies the processing of individual tiles.

In order to reduce bandwidth requirements, Kazemier et al. [53] introduced the concept of *boundary-trees*. A boundary-tree is a subtree of a partial component tree that includes regions intersecting the border of a chunk. Hence, the divide-and-conquer strategy described above is extended to include the computation of boundary-trees and the correction of the partial component trees as follow:

1. Split the data into chunks;
2. Compute a partial component tree for each chunk;
3. Extract boundary trees from the partial hierarchies and merge them two-by-two; and
4. Back-propagate the nested relations of the merged boundary trees to each partial hierarchy.

Hence, instead of transmitting entire partial component trees between nodes, only these much smaller boundary trees are exchanged. The authors report that boundary trees often have memory footprints less than 1% of the partial tree in low dynamic range scenarios.

Building upon this foundation, Gazagnes and Wilkinson [39] introduced improvements to handle High Dynamic Range (HDR) data and extended the methods to 3D datasets. These advancements enable processing of 16-bit and 32-bit floating-point data, which is common in scientific imaging applications such as astronomy and medical imaging. Their framework, DISCCOFAN (Distributed Connected COmponent Filtering and ANalysis), integrates multi-scale analysis tools and supports both 2D and 3D data, further expanding the applicability of component tree-based methods.

A core focus of these works is connected filtering based on attributes [19]. In this approach, attributes must be defined *a priori*, meaning they are computed simultaneously with the partial component trees and then adjusted during the merging phase. However, this rigid structure introduced a major limitation: it was not possible to compute an attribute directly on the DCF once it was constructed.

This constraint restricted flexibility in defining and applying new attributes, particularly in cases where post-processing adjustments were necessary.

To address this shortcoming, Gazagnes and Wilkinson [38] introduced an improved methodology that allows parallel attribute computation once the DCF has been built. This enhancement significantly increases flexibility, enabling users to compute and refine attributes dynamically after the component trees have been constructed decoupling component tree construction and post-processing.

This evolution in distributed component tree computation represents a crucial step toward making connected filtering more efficient, scalable, and applicable to a wider range of large-scale datasets.

2.4.3 Quasi-Flat Zone Hierarchy

This study on component trees is echoed in a domain closely related with our research focus: quasi-flat zones hierarchies (QFZ) [75, 85]. While component trees rely on an order relation between pixel values, our interest lies in BPH, which is built upon a total ordering on edge values. The construction of QFZ follows a similar paradigm and has well-established connections with BPH [28]. More specifically, the QFZ can be seen as the min-tree built on the edges of the minimum spanning tree.

Najman et al. [81] introduced an efficient algorithm for QFZ computation that consists of a linear post-processing step applied to the BPH. While this approach offers an elegant decoupling between the hierarchical structure and the root-finding data structures, it comes with a notable drawback: since the QFZ is not built directly, a significant number of unnecessary nodes are generated. Havel et al. [46] estimated that the QFZ node count is typically 40-70% of the total BPH node count, highlighting a major inefficiency in the process.

To address this, Havel et al. [46] proposed an alternative approach to QFZ computation, merging insights from Najman et al. [81] with their previous work [45] on parallel QFZ computation. Their method directly constructs the QFZ while carefully minimizing the number of redundant nodes. This approach follows the general principles of aforementioned component tree algorithms but introduces a unique characteristic: it fuses two hierarchies within the edge-weighted graph framework by inserting edges at their appropriate rank in the hierarchy.

During QFZ construction, when an edge connecting two separate chunks is processed, the algorithm follows a sequential insertion strategy. Specifically, for a newly introduced edge linking two pixels, the algorithm traverses the paths from each pixel to the root of its respective tree and identifies two critical nodes per tree:

- The first node along the path whose rank is less than or equal to the con-

necting edge's weight;

- The second node, which has a rank greater than the edge weight and extends toward the root.

The nodes belonging to the first category, which represent subtrees below the threshold, are detached and merged. This guarantees that lower-level structures within the hierarchy are properly updated. Meanwhile, the second set of nodes, which defines independent paths to their respective tree roots, must be merged using a zipper-like merging strategy. This process preserves the hierarchical relationships while ensuring consistency across the structure. It is important to note that path compression is performed during edge insertion to further minimize the number of unnecessary nodes.

By structuring the insertion process in this manner, the method efficiently maintains tree consistency, minimizes redundant computations, and ensures that hierarchical relationships are either preserved or corrected as new edges are incrementally added.

For BPH, both the lower and upper bounds of tree height are directly dependent on the number of pixels n rather than the number of gray levels of the image, with a worst-case scenario of height n (comb looking tree) and a best-case scenario of $log(n)$ (perfectly balanced tree). This means that applying the QFZ insertion scheme to BPH is likely to be computationally prohibitive.

As the bit depth increases, the number of hierarchy levels expands, leading to a rapid increase in computational cost and memory usage. These limitations suggest that alternative approaches, such as more scalable hierarchical merging techniques, should be considered for BPH construction.

2.5 Conclusion

In this chapter, we recalled the fundamental concepts of watershed and hierarchical segmentation with the aim of situating the binary partition hierarchy and its associated minimum spanning tree within the broader landscape of hierarchical image analysis. We began by revisiting watershed segmentation as an illustrative example through which meaningful regions emerge from topographic or graph-based interpretations of images. We then reviewed the principal families of hierarchical representations used in mathematical morphology, including component trees, hierarchies of partition, and, within this latter category, hierarchical watershed formulations.

Our review of related work highlighted several important contributions toward scalable hierarchical processing, particularly streaming, parallel, and distributed methods for computing minimum spanning trees, component trees, and quasi-flat

zone hierarchies. Although these approaches offer valuable insights, none directly address the specific challenge that motivates this thesis: the computation of the BPH under memory constraints, where the image graph cannot be stored in main memory. This gap confirms the relevance of developing an out-of-core strategy tailored to BPH construction.

These methodologies provide a strong foundation for the out-of-core solution proposed in the following chapters. First, the divide-and-conquer principles underlying parallel and distributed algorithms motivates our tile-based processing framework, in which the image is partitioned into independent units that can be processed sequentially. Second, the hierarchical representation itself must comply with the out-of-core setting: rather than constructing a single monolithic structure, we maintain a collection of per-tile trees. Finally, we rely on a posteriori attribute computation and refinement of the produced hierarchy into watershed-cut and hierarchical watershed representations.

Chapter 3

Preliminary Mathematics

3.1 Introduction

This chapter presents the mathematical foundation necessary for understanding the methods and results presented in subsequent sections.

We begin by outlining the fundamental principles of graph theory, which serve as the primary framework for representing image data. Graphs, composed of vertices and edges, provide a versatile structure to model relationships between image elements, enabling the formalization of segmentation algorithms. Special attention is given to edge-weighted graphs, where weights encode dissimilarity measures, forming the basis for partitioning and hierarchical processing.

Next, we detail hierarchies of partitions, which represent a series of nested partition.

Finally, we recall the concept of binary partition hierarchies, which play a central role in the segmentation methods discussed in this thesis. The connection between BPHs and minimum spanning trees is highlighted, as it forms a pivotal aspect of the algorithms developed later. On this basis, we define watershed-cut and hierarchical watershed.

3.2 Graphs

In this section, we introduce fundamental concepts in graph theory. We begin by defining a graph as a set of vertices connected by edges, followed by key notions like paths, connected components, and edge weights. These concepts lead us to formal definitions of cycles and trees, which are essential in the study of spanning trees. In particular, we will focus on the minimum spanning tree which constitute a key structure for hierarchical analysis

We define a *graph* as a pair $G = (V, E)$ where V is a finite set and E is composed

of unordered pairs of distinct elements in V, *i.e.*, $E \subseteq \{\{x, y\} \subseteq V \mid x \neq y\}$. Each element of V is called a vertex of G, and each element of E is called an edge of G. The set of all vertices and edges of G is also be denoted $V(G)$ and $E(G)$ respectively.

Let G be a graph. We say that x and y are adjacent in V if there exist an edge $u = \{x, y\}$ such that $u \in E$. Let $\pi = \langle x_0, \ldots, x_l \rangle$ be an ordered sequence of vertices of G such that for any $i \in [1, l]$, $x_{i\text{-}1}$ is adjacent to x_i. We say that π is a *path from x_0 to x_l of length l in G*. We say that G is *connected* if there exist a path between any two vertices of V.

Let X and Y be two graphs. If $V(Y) \subseteq V(X)$ and $E(Y) \subseteq E(X)$, we say that Y is a *subgraph* of X, denoted by $Y \subseteq X$. We say that Y is a *connected component* (CC) of X if Y is a connected subgraph of X which is maximal for this property, *i.e.*, for any connected graph Z, $Y \subseteq Z \subseteq X$ implies $Y = Z$.

A *cycle* in a graph G is a path that starts and ends at the same vertex, where all intermediate vertices are distinct. Formally, let $\langle x_0, \ldots, x_l \rangle$ be a path where $x_0 = x_l$ and the vertices $x_0, \ldots, x_{l\text{-}1}$ are pairwise distinct. A graph containing no such path is said to be *acyclic*. A *tree* is a connected, acyclic graph. It is straightforward to note that a tree contains exactly $|V| - 1$ edges.

Let $G = (V, E)$ be a connected graph. A *spanning tree* of G is a subgraph $T = (V, E')$ such that T is a tree and $E' \subseteq E$.

An *edge-weighted graph* is a pair (G, F) where F is a function from E to $\mathbb{R}$ such that for any edge $e \in E$, $F(e)$ is the weight of the edge e.

Given these two notions, we can define the *Minimum Spanning Tree* of a graph Definition 3.2.1.

Definition 3.2.1 (Minimum Spanning Tree)**.** Let (G, F) be a connected edge-weighted graph and let $T = (V, E')$ be a spanning tree of G. The subgraph T is a *Minimum Spanning Tree* (MST) of G if its total weight $\sum_{u \in E'} F(u)$ is less than equal to the one of any other spanning tree of G.

The Minimum Spanning Tree is a fundamental structure in hierarchical analysis, owing to its notably strong relationship with the binary partition tree. This connection will be explored in detail in the subsequent sections.

3.3 Hierarchies of Partition

Let V be a set. A *partition of V* is a set of pairwise disjoint subsets of V. Each element of a partition $\mathbf{P}$ is called a *region* of $\mathbf{P}$. The *ground* of a partition $\mathbf{P}$, denoted $gr(\mathbf{P})$, is the union of its regions. A partition whose ground is V is called a *complete partition of V*. Given two partitions $\mathbf{P}$ and $\mathbf{Q}$ of V, we say that $\mathbf{Q}$ is

a *refinement of* $\mathbf{P}$ if each region of $\mathbf{Q}$ is included in a region of $\mathbf{P}$. A *hierarchy on* V is a sequence $(\mathbf{P}_0, \ldots, \mathbf{P}_\ell)$ of partitions such that for every $\lambda \in \{0, \ldots, \ell - 1\}$, the partition $\mathbf{P}_\lambda$ is a refinement of $\mathbf{P}_{\lambda+1}$. Let $\mathcal{H} = (\mathbf{P}_0, \ldots, \mathbf{P}_\ell)$ be a hierarchy. The integer ℓ is called the *depth of* $\mathcal{H}$ and each partition $\mathbf{P}_\lambda$ is called the λ-*scale of* $\mathcal{H}$. For any $\lambda \in \{0, \ldots, \ell\}$, we denote the λ-scale of $\mathcal{H}$ by $\mathcal{H}[\lambda]$. A *region of* $\mathcal{H}$ is any region that belongs to one of its scales. Formally, a set $R \subseteq V$ is a region of $\mathcal{H}$ if there exists $\lambda \in \{0, \ldots, \ell\}$ such that $R \in \mathbf{P}_\lambda$. The hierarchy $\mathcal{H}$ is said to be *complete* if $\mathcal{H}[0] = \{\{x\} \mid x \in V\}$ and $\mathcal{H}[\ell] = \{V\}$.

The *ground* of a hierarchy, denoted by $gr(\mathcal{H})$, is the ground of its 0-scale.

We denote by $\mathcal{H}_\ell(V)$ the set of all hierarchies on V of depth ℓ, by $\mathcal{P}(V)$ the set of all partitions on V.

In order to represent hierarchies of partitions in a lightweight manner, it is convenient to consider them as trees where leaves corresponds to the ground of the hierarchy, *i.e.*, the set of elementary regions, while the non-leaf nodes represents regions that are unions of other regions. Given a complete hierarchy $\mathcal{H}$ and two of its regions X and Y, we say that X *is a parent* of Y (or that Y *is a child of* X) if $Y \subset X$ and X is minimal for this property, *i.e.*, for any region Z such that $Y \subseteq Z \subset X$ then $Y = Z$. Any region $X \neq V$ has only one parent. The region $X = V$ is called the *root* of $\mathcal{H}$. Any region which is not parent of another one is called a *leaf*. The relation between the series of partitions and the tree representation is illustrated in Figure 3.1. We can see that each non leaf node is associated with two components:

- **The region it represents**, for instance, the region labeled as 6 in Figure 3.1.
- **The scale at which it appears**, that is the finest partition $\mathbf{P}_\lambda$ in which the region first emerges. In the example, region 6 appears at scale $\lambda = 1$.

We will discuss this representation of hierarchies with a specific emphasis on data-structures in Section 4.5.

3.4 Binary Partition Hierarchy

The binary partition hierarchy by altitude ordering, simply called binary partition hierarchy in the following, is associated to a graph and to a total ordering on the edges of this graph. In general, for image or data segmentation, this ordering is obtained by weighting the edges of the graph (*e.g.*, with a dissimilarity measure between vertices) and by sorting them in increasing order of weight.

Let (G, F) be an edge-weighted graph. Let $\prec$ be a total ordering on E. This binary relation is transitive and trichotomous, *i.e.*, for any a and b in E only one of the following relations holds true: $a \prec b$, $b \prec a$ or $a = b$. If $F(a) < F(b)$

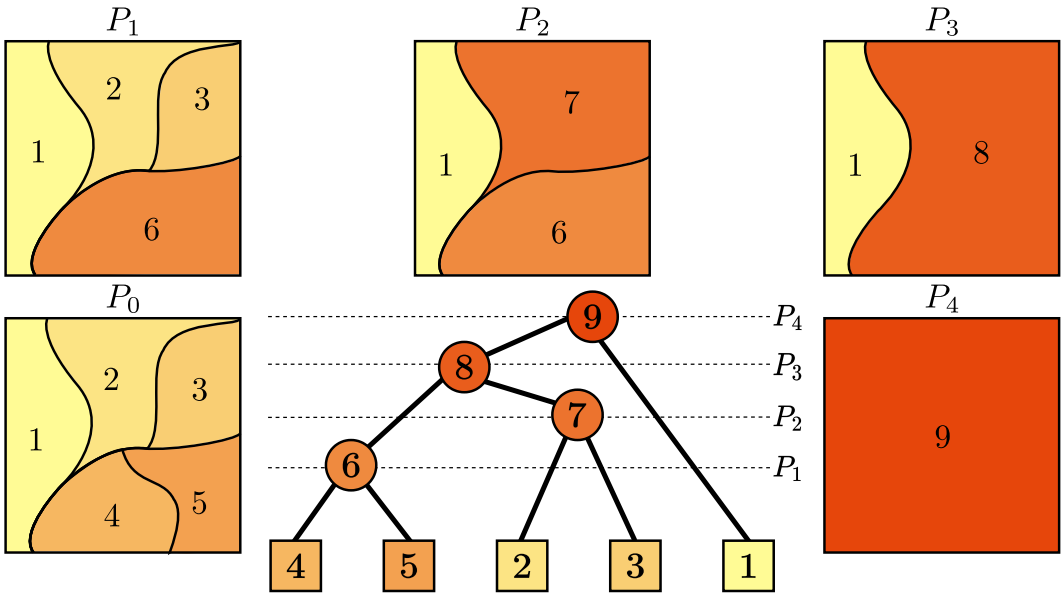


Figure 3.1: The series (P_0, P_1, P_2, P_3) of partitions of the set $\{1, 2, 3, 4, 5\}$ and its tree representation. Each node is associated with the number of the region it represents.

then $a \prec b$. On can notice that relying only on a dissimilarity metrics does not guarantee a total ordering. This issue can be easily addressed by considering edge index as an additional criterion while comparing edges of same weight for F (see Section 4.5).

Let $\prec$ be a total order and let k in $\{1, \ldots, \ell\}$, we denote by $u_k^{\prec}$ the k-th element of E for the order $\prec$. Let u be an edge in E, the *rank of* u *for* $\prec$, denoted by $r^{\prec}(u)$, is the unique integer k such that $u = u_k^{\prec}$.

In order to define the binary partition tree by altitude ordering, we will adopt the formalism of [30]. This structure can be defined thanks to an elementary exterior operation on hierarchies, called an update. Let $\mathcal{H}$ be a hierarchy in $\mathcal{H}_\ell(V)$ and let $\{x, y\}$ be an edge of G. The *update of* $\mathcal{H}$ *with respect to* $\{x, y\}$, denoted by $\mathcal{H} \oplus \{x, y\}$ is the hierarchy such that:

- $(\mathcal{H} \oplus \{x, y\})[\lambda]$ remains unchanged (*i.e.*, $\mathcal{H} \oplus \{x, y\}[\lambda] = \mathcal{H}[\lambda]$) for any λ in $\{0, k\text{ - }1\}$, with k being the rank of $\{x, y\}$ for $\prec$, while
- for any λ in $\{k, \ldots, \ell\}$, we have $(\mathcal{H} \oplus \{x, y\})[\lambda] = \mathcal{H}[\lambda] \setminus \{R_x, R_y\} \cup \{R_x \cup R_y\}$ where R_x (resp. R_y) denotes the region of $\mathcal{H}[\lambda]$ containing x (resp. y).

In other words, the update $\mathcal{H} \oplus \{x, y\}$ consists in inserting the edge $\{x, y\}$ at its correct rank in $\mathcal{H}$ for $\prec$. This is done by merging the regions containing x

and y at scale $k = r^{\prec}(\{x, y\})$ and propagating this merge to every scale $\lambda > k$, while leaving all scales $\lambda < k$ unchanged.

Let $E' \subseteq E$ and let $\mathcal{H}$ be a hierarchy. We set $\mathcal{H} \boxplus E' = \mathcal{H} \oplus u_1 \oplus \cdots \oplus u_{|E'|}$ where $E' = \{u_1, \ldots, u_{|E'|}\}$. The binary operation $\boxplus$ is called the *edge-addition*. Note that given Property 3 of [30] the binary operation $\oplus$ is commutative with respect to edges, *i.e.*, given u and v two edges of E the following statement holds true: $\mathcal{H} \oplus u \oplus v = \mathcal{H} \oplus v \oplus u$. As a result the order in which the edges of E are considered for $\boxplus$ does not matter. We can then define the binary partition tree by altitude ordering only using this last operator.

Definition 3.4.1 (Binary Partition Hierarchy by Altitude Ordering)**.** Let V be a set and let $\perp_V$ be the hierarchy defined by $\perp_V [\lambda] = \{\{v\} \mid v \in V\}$, for any λ in $\{0, \ldots, \ell\}$. The *binary partition hierarchy by altitude ordering* of V for $\prec$, denoted by $\mathcal{B}_V^{\prec}$ is the hierarchy $\perp_V \boxplus E$.

The structure itself being described, we will now focus on the close relationship between the binary partition hierarchy and the minimum spanning tree.

Let $\mathcal{B}^{\prec}$ be a binary partition by altitude ordering of $(G = (V, E), F)$, let R be a region of $\mathcal{B}^{\prec}$ and let $R^{\star}$ be the set of non-leaf regions of $\mathcal{B}^{\prec}$. The *rank* of R, denoted by $r(R)$, is the lowest integer λ such that R is a region of $\mathcal{B}^{\prec}[\lambda]$. We consider the map μ from $R^{\star}$ in E such that, for any non-leaf region R of $\mathcal{B}^{\prec}$, we have $\mu(R) = u_{r(R)}^{\prec}$. We say that $\mu^{\prec}(R)$ is the *building edge* of R. We now introduce an important property of the binary partition hierarchy related to its building edges with Property 3.4.1.

Property 3.4.1. *Let $E' = \{\mu(R) \mid R \in R^{\star}\}$ be the set of building edges of $\mathcal{B}^{\prec}$. The graph $T = (V, E')$ is a minimum spanning tree of G.*

We can establish a closer relationship with a corollary: the leaf nodes of the BPH correspond bijectively to the vertices of the MST, while the non-leaf nodes correspond bijectively to the edges of the MST. This relation is depicted in Figure 3.2. The BPH is "rooted" in the base graph with each leaf node is mapped to a vertex of the MST, and each non-leaf node is mapped to an edge of the MST via μ, as illustrated in red.

As stated previously, the BPH is a very versatile structure and in the following sections we will focus on a particular segmentation that can be obtained from this hierarchy: the watershed.

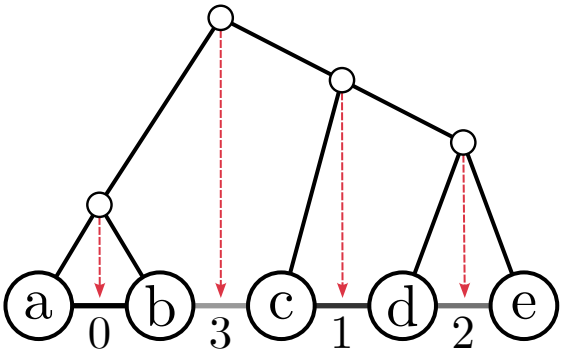


Figure 3.2: A BPH built on a 1D edge-weighted graph $\{a, b, c, d, e\}$. The red arrows represent the map μ from non-leaf regions to edges of the MST. The leaves of the BPH correspond to the vertices of the MST.

3.5 Watershed

As stated in the previous chapter, the watershed is a fundamental morphological tool for image segmentation. Specifically, the watershed-cut defined on edge-weighted graphs [26] exhibits strong connections with minimum spanning trees and binary partition hierarchies. In this section, we first give a formal definition of the watershed-cut and its properties. Then, we explore its relationship with minimum spanning trees and binary partition hierarchies.

3.5.1 Watershed in Edge-Weighted Graphs: Watershed-Cut

The watershed cut [26] is defined following the drop of water principle within the framework of edge-weighted graphs. Unlike other previous approaches in the discrete framework, authors introduced a formalization of the aforementioned principle leading to important properties.

1. **Consistency**: Watershed cuts can be consistently defined using either catchment basins or dividing lines. This means that they can be characterized by the regions or by the separations between these regions.

2. **Optimality**: Watershed cuts are optimal in the terms of Minimum Spanning Forest (MSF). They are indeed equivalent to the separations induced by minimum spanning forests relative to the minima of the graph.

In this section, we will focus on the notion of relative MSF. In order to introduce this concept, it is essential to define the notions of minima and extension which

play a key role in the definition of the watershed. These notions intuitively link to the watershed, as we previously observed a bijection between local minima and catchment basins.

Let $X \subseteq G$. An edge $\{x, y\}$ is said to be *adjacent to* X if $\{x, y\} \cap V(X) \neq \emptyset$ and $\{x, y\} \notin E(X)$. We can define regional minima as follows:

Definition 3.5.1 (Regional minimum (of a graph))**.** Let $G = (V, E, F)$ be an edge-weighted graph. Let $X \subseteq G$. The subgraph X is a *minimum of* F if:

1. X is connected;
2. the weight of any edge adjacent to X is strictly greater than the weight of any edge of X; and
3. all edge in $E(X)$ have the same weight.

Then, we adopt the formalism of extension from Bertrand [12].

Definition 3.5.2 (Extension, from Definition 12 in [12])**.** Let X and Y be two non empty subgraph of G. We say that Y is an *extension* of X if each connected component of Y contains exactly one connected component of X.

Hence an extension of a the set M of regional minima is a subgraph such that each connected component of the extension contains exactly one connected component of the set of regional minima. Catchment basins are then the connected components of the graph induced by a specific extension of the regional minima.

In the classical definition of a forest, each tree is inherently acyclic. In the following we will relax this constraint through the concept of a *relative forest*. Given two subgraph X and Y, we say that Y is a forest relative to X if it is an extension of X such that any cycle present in Y already exists in X. Here is a formal definition of a relative forest.

Definition 3.5.3 (Relative Forest, from Definition 7 in [28])**.** Let X and Y be two non-empty subgraphs of a graph G. We say that Y is a forest relative to X if:

1. Y is an extension of X; and
2. For any extension $Z \subseteq Y$ of X, we have $Z = Y$ whenever $V(Z) = V(Y)$.

In Definition 3.5.3, the first condition states that each connected component of Y contains exactly one connected component of X, and the second ensures that Y is minimal for this property.

Let X be a subgraph of G, we says that Y is a *spanning forest relative to* X if Y is a forest relative to X and $V(Y) = V(G)$. We can now define the notion of relative minimum spanning forest. In this way, we can constraint the extensions of the regional minima to follow a criterion given by a the combinatorial optimization problem of the minimum spanning tree.

Definition 3.5.4 (Relative Minimum Spanning Forest, from Definition 8 in [28])**.** Let X and Y be two subgraphs of a graph G. We say that Y is *a minimum spanning forest* (MSF) relative to X (for F, in G) if:

1. Y is a spanning forest relative to X; and
2. The weight of Y, defined as $F(Y) = \sum_{e \in E(Y)} F(e)$, is less than or equal to the weight of any other spanning forest relative to X.

In this case, Y is referred to as *a relative MSF*.

Interestingly we can define the minimum spanning tree such that it corresponds to the usual definition (Definition 3.2.1) with the notion of relative MSF. The minimum spanning tree of a graph G is indeed simply the MSF relative to the subgraph $(\{x\}, \emptyset)$ with $x \in V$.

We say that $X \subseteq G$ is a *connected component* of G if X is a connected subgraph of G which is maximal for this property, *i.e.*, for any connected graph Z, $X \subseteq Z \subseteq G$ implies $X = Z$. We note by $CC(G)$ the set of connected components of G. The *connected component partition of* G is the partition defined as $\{V(X) \mid X \in CC(G)\}$.

Through the optimality property of [26], we can now define the watershed-cut as the connected component partition of a minimum spanning forest relative to the regional minima of the graph.

Definition 3.5.5 (Watershed-Cut)**.** Let $G = (V, E, F)$ be an edge-weighted graph. A *watershed-cut* of G is the connected component partition of a minimum spanning forest of G relative to the regional minima of G.

This MSF viewpoint is particularly convenient because it allows us to extend the definition of the watershed from minima to arbitrary marker sets.

3.5.2 Watershed-cut from Markers

We defined watershed-cut as the connected component partition of a minimum spanning forest rooted in the regional minima. By substituting the regional minima with markers, the watershed-cut aligns with marker-based watershed segmentation such as formalized in Theorem 3.5.1 and proved by Allène et al. [2]. From

an algorithmic approach, this formulation allows leveraging Kruskal's MST algorithm, enhanced by Tarjan's Union-Find structure, to achieve a quasilinear time complexity.

Theorem 3.5.1 (Marker-based Watershed-Cut)**.** Let $G = (V, E, F)$ be an edge-weighted graph, let $M \subset V$ be a marker set and $G_M \subseteq G$ be the graph induced by M. If $\mathbf{P}$ is a connected component partition of a minimum spanning forest relative to G_M, then $\mathbf{P}$ if a watershed-cut for the marker set M.

3.5.3 Link with Binary Partition Hierarchy

The link between watershed-cut and BPH was stated in Cousty et al. [28]. Indeed, the cuts are the "highest separations" of minima in the minimum spanning tree. Let $T \subseteq G$ be a minimum spanning tree and let M_0 and M_1 be two distinct minima of G. Let u be the edge of T such that the removal of u from T results in the minimum spanning forest relative to M_0 and M_1. Let $\pi = \langle a, \ldots, b \rangle$ be the unique path in T with $a \in M_0$ and $b \in M_1$. The edge u is the edge of highest altitude on the path π, *i.e.*, :

$$F(u) = \max_{v \in \pi} F(v)$$

Consequently, u is the building edge of the smallest region R of the BPH $\mathcal{B}^{\prec}$ built on G containing M_0 and M_1. Intuitively, considering the tree representation of $\mathcal{B}^{\prec}$, the minima M_0 and M_1 are represented by two distinct nodes, as defined in Definition 3.5.6, and R corresponds to their lowest common ancestor. This relationship forms the foundation our work on incremental interactive segmentation which will be extensively explored and discussed in the Chapter 7.

Definition 3.5.6 (Minimum of a graph (given its associated Binary Partition Hierarchy))**.** Let $\mathcal{B}$ be a Binary Partition Hierarchy built on a graph G. The region n of $\mathcal{B}$ is a *minimum of* G if:

1. Let p be the parent of n in $\mathcal{B}$, we have altitude$(n) <$ altitude(p);
2. Each child c of n in $\mathcal{B}$, is either a leaf or a non-minimum in $\mathcal{B}$.

Edges of the MST corresponding the the lowest common ancestor of distrinct markers, called "*watershed edge*" can be identified through a linear-time post-processing of the BPH, as described by Najman et al. [81]. The goal of this post-processing step is to flag each edge of the MST in order to indicate whether or not it is a watershed edge that should be removed from the MST. This can be achieved by browsing nodes of the BPH in increasing order of altitude. During

the traversal, each node is checked to determine if it is a minimum. For each non-leaf node, a counter is incremented indicating the number of distinct minima in the subtree rooted at this node. Each node whose minima count is higher than their children is a watershed node. Consequently, its building edge merges distinct catchment basins, being a watershed edge. Corresponding algorithms are given in Section 5.5.1.

In this way, edges belonging to the watershed cut can be determined with the BPH. In the next section, we will further exploit the hierarchical representation for the sake of the watershed.

3.6 Hierarchical Watershed

A formalization of hierarchical watershed was given for edge-weighted graph as a series of nested watershed cut [25, 28].

Let G be an edge-weighted graph and let C to be an attribute mapping any subgraph of G to a positive integer. We say that C is increasing if, for any two subgraphs X and Y such as $X \subseteq Y$, we have $C(X) \leq C(Y)$. Typical examples include geometric attributes such as area, height, or volume.

Regarding the formulation of watershed, we are only interested in extrema oriented connected operators. Thus, as established by Cousty et al. [28], we will define the hierarchical watersheds with the notions of *extinction values* and *persistence*.

As introduced by Vachier and Meyer [120], we can define the extinction value of the regional minima according to a criterion C. Let $\Phi = (\Phi_0(G), \ldots, \Phi_k(G))$ be series of progressive filtration of G for C indexed from the finest level $\Phi_0(G) = G$ to the coarsest level $\Phi_k(G)$. For each regional minimum M of G, we define its extinction value $\epsilon(M) \in \{0, \ldots, k\}$ as the largest index i such that M is still a regional minimum of $\Phi_i(G)$. We note $\mathbf{P}_{CC}(G)$ the *connected component partition of* G. By using any increasing criterion, we can give a first expression of hierarchical watershed with Definition 3.6.1.

Definition 3.6.1 (MSF hierarchy)**.** Let G be an edge-weighted graph and let $\mathcal{M}$ be the set of its regional minima. For any $i \in \{0, \ldots, k\}$, we define

$$\mathcal{M}_i \;=\; \{\, M \in \mathcal{M} \mid \epsilon(M) \geq i \,\}.$$

Let $(G_0, \ldots, G_k)$ be a sequence of subgraphs of G such that:

1. for each $i \in \{0, \ldots, k\}$, G_i is a minimum spanning forest of G relative to $\mathcal{M}_i$; and

2. for any $i \in \{1, \ldots, k\}$, we have $G_{i-1} \subseteq G_i$.

The series $\big(\mathbf{P}_{CC}(G_0), \ldots, \mathbf{P}_{CC}(G_k)\big)$ is called a *minimum spanning forest hierarchy of* G.

While the notion of extinction values is sufficient to define hierarchical watershed, persistence provides a more practical tool from an algorithmic perspective.

The persistence of an edge u is defined as the largest index j such that the two vertices incident to u still belong to different regions in the partition $\mathbf{P}_{CC}(\Phi_j(G))$. Let $\mathcal{B}$ be the BPH of G, and let ρ be a map that associates to each building edge u of $\mathcal{B}$ a positive integer $\rho(u) \in \{0, \ldots, n\}$ equal to its persistence. The hierarchical watershed can then be defined with respect to this persistence map as in Definition 3.6.1.

Property 3.6.1 (MSF hierarchy by persistence)**.** *Let $G = (V, E, F)$ be an edge-weighted graph and let $T = (V, B)$ be a minimum spanning tree of G. Let $\rho : B \to \{0, \ldots, n\}$ be the persistence map associated with an MSF hierarchy in the sense of Definition 3.6.1. For each $i \in \{0, \ldots, n\}$, define*

$$B_i \;=\; \{\, u \in B \mid \rho(u) \leq i \,\},$$

and let $\mathbf{P}_i = \mathbf{P}_{CC}(V, B_i)$ be the partition of V induced by the connected components of the graph (V, B_i). Then the series $(\mathbf{P}_0, \ldots, \mathbf{P}_n)$ is a minimum spanning forest hierarchy of G in the sense of Definition 3.6.1.

In other words, thresholding the ultrametric (persistence) associated with a given MSF hierarchy recovers that very hierarchy as a nested sequence of connected-component partitions.

We will use this definition of hierarchical watershed in the rest of this thesis. The practical computation of hierarchical watershed will be discussed in Section 5.6.

3.7 Conclusion

In this chapter, we have established the theoretical foundation of this thesis, highlighting key concepts and their interconnections. We have explored the relationships between the binary partition hierarchy, the minimum spanning tree, the watershed cut, and the hierarchical watershed, providing a cohesive framework for the work that follows. In the subsequent chapters, we will focus, one the one hand, on the theoretical aspects of our out-of-core hierarchical framework, and on the other hand, on the practical aspects of the implementation of this framework thus returning to the notions introduced previously.

Chapter 4

Binary partition Hierarchy Out-of-Core Calculation

Related Publications

- J. Lefèvre, J. Cousty, B. Perret, and H. Phelippeau. Join, select, and insert: Efficient out-of-core algorithms for hierarchical segmentation trees. In *Discrete Geometry and Mathematical Morphology*, pages 274–286. Springer, 2022. 10.1007/978-3-031-19897-7_22

- J. Lefèvre, J. Cousty, B. Perret, and H. Phelippeau. Out-of-core algorithms for binary partition hierarchies. *Journal of Mathematical Imaging and Vision*, 67(2):18, 2025. 10.1007/s10851-025-01234-0

4.1 Introduction

In previous chapters, we introduced the binary partition hierarchy as a versatile representation for hierarchical segmentation, supporting a wide range of applications in image analysis, from "flat" and hierarchical watershed transforms to connected filters and quasi-flat zone hierarchies. Despite its algorithmic efficiency, the BPH's construction and manipulation are memory-intensive operations, making them particularly challenging when dealing with the massive images generated by modern imaging techniques such as electron microscopy.

In this chapter we address the construction of the BPH under memory constraint. The established algorithm introduced by Najman et al. [81] constructs it by progressively merging regions in increasing order of weights. However, the global nature of this process which requires a full edge sorting and a union-find traversal over the entire graph, imposes strict memory constraints. As a result,

standard BPH construction quickly becomes impractical for large 2D or 3D images on memory-limited systems.

We propose to address this limitation by presenting a framework for out-of-core computation of a light-weight but exact representation of the BPH called the *distribution of binary partition hierarchy*. Following the idea introduced by Kazemier et al. [53], the approach we propose decomposes both the graph and the hierarchical structure into manageable subparts.

In particular, we present two types of out-of-core BPH computations. The first one is able to consider the distribution of the BPH over any generic partition of the space while the second one, originally presented in Cousty et al. [30], deals only with causal partitions. We present the advantages and drawbacks of these two types of computations in the form of

1. Properties 4.4.2 and 4.4.3 asserting to what extent the out-of-core constraint is respected by the second computation,
2. Counterexample 4.4.1 showing the theoretical limits of the first one, and
3. An experimental study validating both approaches and experimentally comparing the proposed computation schemes.

This chapter is directly adapted from the journal publication Lefèvre et al. [59], with minimal modifications to fit the structure of the thesis.

4.2 Distribution of Binary Partition Hierarchy

In this section, we define the structure we are seeking to build, namely the *distribution of BPH*. Intuitively, distributing a hierarchy over a graph partitioned into chunks consists in splitting it into a set of smaller hierarchies such that:

1. each smaller hierarchy corresponds to a selected subpart of the whole hierarchy that hits a chunk of the graph; and
2. the initial hierarchy can be reconstructed by "gluing" those smaller hierarchies.

We first present the operations that allow for selecting a subset of a partition and of a hierarchy that hit a given subset of the space, before providing the definition of a distribution of a hierarchy.

Let $2^{|V|}$ be the set of all subsets of V, the operation *sel* is the map from $2^{|V|} \times \mathcal{P}(V)$ to $\mathcal{P}(V)$ which associates with any subset X of V and any partition $\mathbf{P}$ of V the subset $sel(X, \mathbf{P})$ of $\mathbf{P}$ which contains every region of $\mathbf{P}$ that contains an element of X. On this basis, we can now define the first core operation of our framework, the *select* operation.

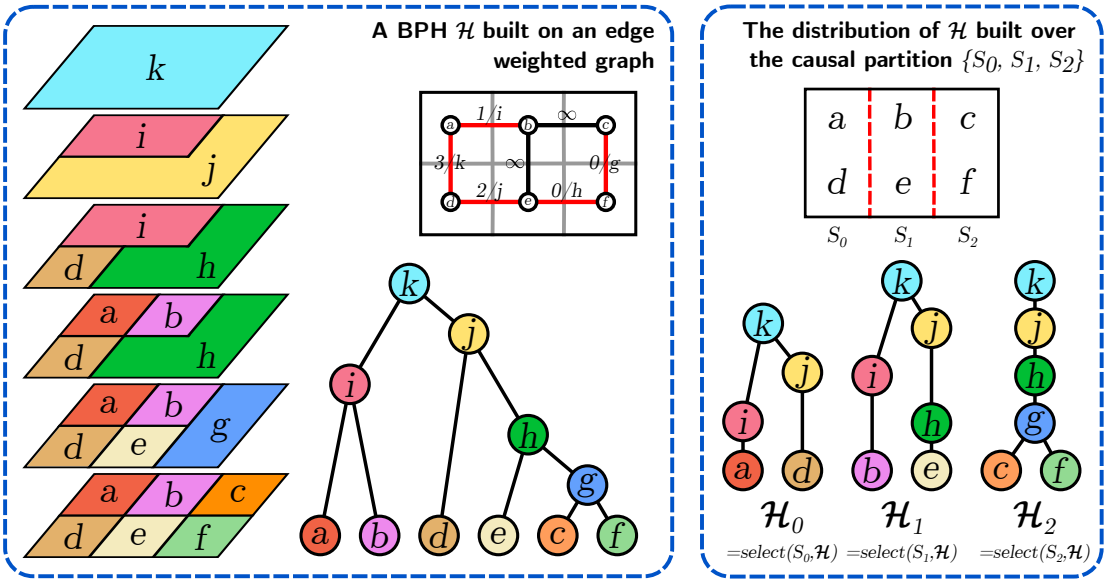


Figure 4.1: Left, a BPH $\mathcal{H}$ built on an edge-weighted graph (*weight*/*index*) and the visualization of the nested series of partition. Red edges of the graph belong to the minimum spanning tree. Right, the distribution $\{\mathcal{H}_0, \mathcal{H}_1, \mathcal{H}_2\}$ of $\mathcal{H}$ over the causal partition $\{S_0, S_1, S_2\}$.

Definition 4.2.1 (Select)**.** The operation *select* is the map from $2^{|V|} \times \mathcal{H}_\ell(V)$ in $\mathcal{H}_\ell(V)$ which associates with any subset X of V and to any hierarchy $\mathcal{H}$ on V the hierarchy select $(X, \mathcal{H}) = (\text{sel}(X, \mathcal{H}[0]), \ldots, \text{sel}(X, \mathcal{H}[\ell]))$. We say that select $(X, \mathcal{H})$ is the *selection of* X *in* $\mathcal{H}$.

Intuitively, this operation consists in the extraction of a sub-hierarchy from a given hierarchy, comprising every region that intersects the input subset. Analogously, within the context of a tree representation of a hierarchy, it corresponds to selecting the subtree encompassing all selected leaves corresponding to the vertices of the input subset and their parents.

In Figure 4.1, the hierarchy $\mathcal{H}_0$ corresponds to the result from applying the operation select $(\{a, d\}, \mathcal{H})$.

We can use this latter definition to define a representation consisting in the decomposition of a hierarchy in a set of smaller hierarchies.

Definition 4.2.2 (Distribution)**.** Let $\mathbf{P}$ be *a complete partition on* V and let $\mathcal{H}$ be a hierarchy on V. *The distribution of* $\mathcal{H}$ *over* $\mathbf{P}$ is the set

$$\{\text{select}(R, \mathcal{H}) \mid R \in \mathbf{P}\}$$

The distribution of a hierarchy is depicted on Figure 4.1 with $\{\mathcal{H}_0, \mathcal{H}_1, \mathcal{H}_2\}$ being the distribution of $\mathcal{H}$ over the partition $\{\{a, d\}, \{b, e\}, \{c, f\}\}$. Given the distribution of a hierarchy, we can reconstruct the original hierarchy by joining the elements of the distribution together as stated in Property 2 of [30].

4.3 An Algebraic Framework for Binary Partition Hierarchy Out-of-Core Computation

In this section, we recall the definition of two fundamental operations, *join* and *insert* introduced in [30]. These operations, together with *select* operation, enable the computation of BPH distributions in an out-of-core fashion.

We denote by γ the operator that maps to any two distinct subsets X and Y of V the subset of E made of every edge of E that contains exactly one vertex of X and exactly one vertex of Y, *i.e.*, $\gamma(X, Y) = \{\{x, y\} \in E \mid x \in X \wedge y \in Y\}$. The set $\gamma(X, Y)$ is called the *common neighborhood of* X *and* Y. For instance, in Figure 4.1, the common neighborhood of the two regions S_0 and S_1 contains the edges $\{a, b\}$ and $\{d, e\}$: $\gamma(S_0, S_1) = \{\{a, b\}, \{d, e\}\}$.

Let $\mathcal{X}$ and $\mathcal{Y}$ be two hierarchies in $\mathcal{H}_\ell(V)$. We say that $\mathcal{Y}$ is *smaller than* $\mathcal{X}$ if, for any λ in $\{0, \ldots, \ell\}$, the partition $\mathcal{Y}[\lambda]$ is a refinement of $\mathcal{X}[\lambda]$. If $\mathcal{Y}$ is smaller than $\mathcal{X}$, we write $\mathcal{Y} \sqsubseteq \mathcal{X}$. The set $\mathcal{H}_\ell(V)$ equipped with the relation $\sqsubseteq$ is a lattice whose supremum is denoted by $\sqcup$ (see [97] for properties of this lattice and supremum and see Figure 4.2 for an illustration of supremum). This notion allows us for introducing the operation *join*.

Definition 4.3.1 (Join)**.** Let $\mathcal{X}$ and $\mathcal{Y}$ be two hierarchies in $\mathcal{H}_\ell(V)$. The *join of* $\mathcal{X}$ *and* $\mathcal{Y}$, denoted by $join(\mathcal{X}, \mathcal{Y})$, is the hierarchy defined by $join(\mathcal{X}, \mathcal{Y}) = (\mathcal{X} \sqcup \mathcal{Y}) \boxplus F$, where F is the common neighborhood of the grounds of $\mathcal{X}$ and of $\mathcal{Y}$, *i.e.*, $F = \gamma(gr(\mathcal{X}), gr(\mathcal{Y}))$.

The operation join is illustrated in Figure 4.3. We can see that edges 4 and 5 are inserted at their proper rank in $\mathcal{J}$ discarding the region 8 which does not belong to the MST.

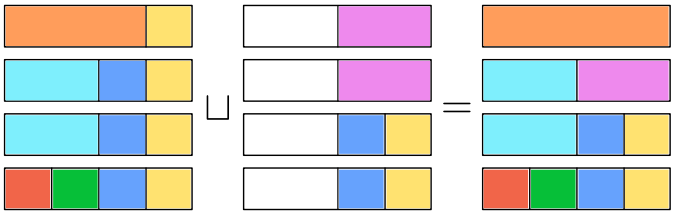
With *join*, we have the ability to merge two hierarchies into a single one. Now we give the definition of *insert* that is the last operation of this framework allowing for correcting the nested relation of a hierarchy given another one.

Definition 4.3.2 (Insert)**.** Let $\mathcal{X}$ and $\mathcal{Y}$ be two hierarchies.

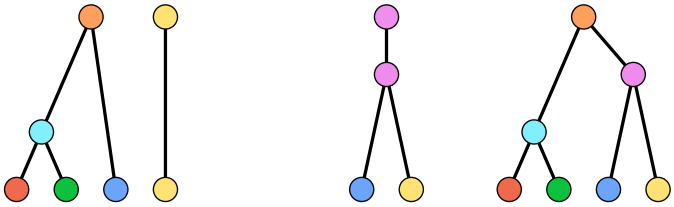
We say that $\mathcal{X}$ *is insertable in* $\mathcal{Y}$ if, for any λ in $\{0, \ldots, \ell\}$, for any region Y of $\mathcal{Y}[\lambda]$, Y is either included in a region of $\mathcal{X}[\lambda]$ or is included in $V \setminus gr(\mathcal{X}[\lambda])$.



(a) Edge-weighted graph



(b) Nested partitions



(c) Tree representation

Figure 4.2: Illustration of the supremum of hierarchies. The leftmost hierarchy correspond to the BPH built over the graph excluding the rightmost edge. The middle hierarchy correspond to the BPH built over the graph composed of the excluded edge and its extremities. Finally, the rightmost hierarchy is the supremum of the two previous ones.

> Let $\mathcal{X}$ be insertable in $\mathcal{Y}$. The *insertion of* $\mathcal{X}$ *into* $\mathcal{Y}$ is the hierarchy $\mathcal{Z}$, such that, for any λ in $\{0, \ldots, \ell\}$, $\mathcal{Z}[\lambda] = \mathcal{X}[\lambda] \cup \{R \in \mathcal{Y}[\lambda] \mid R \cap gr(\mathcal{X}[\lambda]) = \emptyset\}$. The insertion of $\mathcal{X}$ into $\mathcal{Y}$ is denoted by $\text{insert}(\mathcal{X}, \mathcal{Y})$.

The insert operation is illustrated on Figure 4.4. We can see that $\mathcal{X}$ is *insertable* into $\mathcal{Y}$ because i) the two rightmost leaves of $\mathcal{Y}$ are included in $V \backslash gr(\mathcal{X}[\lambda])$ and ii) every other region of $\mathcal{Y}$ is included into a region of $\mathcal{X}$ at the same scale. As a result, the region U of $\mathcal{Y}$ is discarded in $\mathcal{Z}$ because $U \cap gr(\mathcal{X}[5]) \neq \emptyset$ ($gr(\mathcal{X}[6])$ highlighted in red).

4.4 High Level Calculi of Binary Partition Hierarchy Distribution

In this section, we present the high-level calculi that allow us to build the distribution of the BPH over a complete partition of the space relying on the previously defined algebraic operators. We first present the generic partition calculus, which

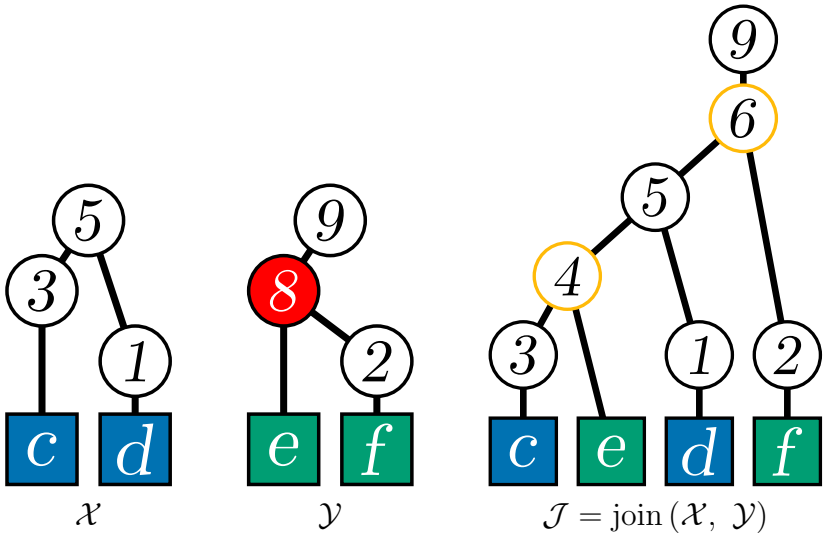


Figure 4.3: *Join* of $\mathcal{X}$ and $\mathcal{Y}$ given their common neighborhood $\{\{d, f\}, \{c, e\}\}$ of ranks 6 and 4.

is able to build the distribution of the BPH over any complete partition of the space. Then, we present the causal partition calculus, which is a more specific case of the generic one, but which is more efficient in practice.

4.4.1 Generic Partition

In this section, we present a calculus to build the distribution of the BPH over any complete partition of the space, using the three select, insert, and join operations. This calculus relies on the construction of an intermediary structure, that we call a *Global Border Tree* (GBT). After presenting, this calculus, we finish the section with a counter example showing the limitation of this approach for certain out-of-core computation contexts.

We denote by $\delta^\bullet$ the operator that maps any subset F of E to the subset of V made of every vertex of V that belongs to an edge in F, *i.e.*, $\delta^\bullet(F) = \cup F$.

Let A and B be any two subsets of V. We set $\gamma^\bullet_B(A) = \delta^\bullet(\gamma(A, B)) \cap A$. In other words, $\gamma^\bullet_B(A)$ contains every vertex in A which is adjacent to a vertex in $B \setminus A$. If A is a subset of B, the set $\gamma^\bullet_B(A)$ is called the *(internal) border of* A. These operators are illustrated in Figure 4.5.

Given these operators and the operator *select*, we can define the *global border tree* of a hierarchy over a partition of the space.

Definition 4.4.1 (Global Border Tree)**.** Let $\mathbf{P}$ be a complete partition on V

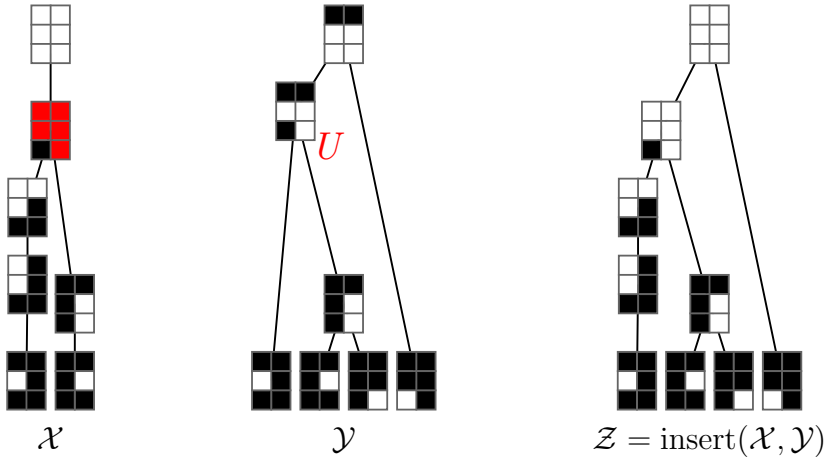


Figure 4.4: Insertion of $\mathcal{X}$ into $\mathcal{Y}$ to obtain $\mathcal{Z}$. Each region is represented by the vertices of the input space included in this region in white. The region U of $\mathcal{Y}$ at altitude 6 is discarded from $\mathcal{Z}$ because of its non-null intersection with $gr(\mathcal{X}[6])$ (highlighted in red).

> and let $\mathcal{H}$ be a hierarchy on V. Let F be the set made of every vertex which belongs to the border of a region of $\mathbf{P}$: $F = \cup\{\gamma_V^\bullet(R) \mid R \in \mathbf{P}\}$. The *Global Border Tree of* $\mathcal{H}$ *over* $\mathbf{P}$ is the hierarchy select $(F, \mathcal{H})$.

A detailed algorithm to build the global border tree associated with a partition $\mathbf{P}$ of V into tiles and to an ordering $\prec$ on the edges of G is presented in Algorithm 1. Intuitively, this algorithm aims at building the (local) BPH associated with each tile in $\mathbf{P}$ (line 4), selecting the border of each tile in these local BPH (line 5), and incrementally joining them together to form the global border tree (line 6). In order to build the local BPH, we call the algorithm proposed by Najman et al. [81], hereafter denoted by PWK. The detailed algorithms to

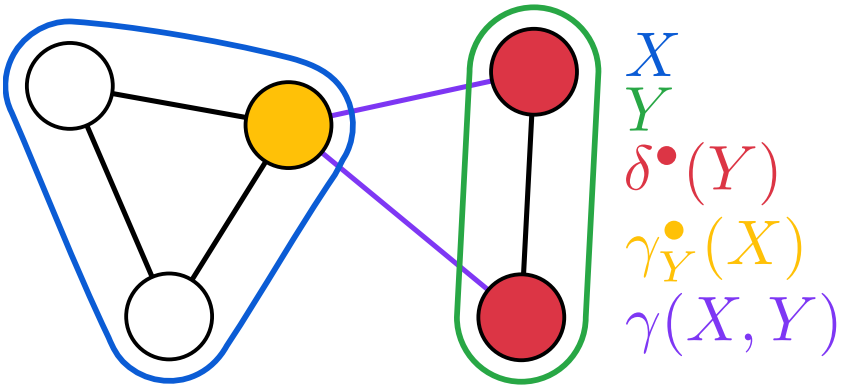


Figure 4.5: Operators to access specific subsets of the graph.

compute the results of join and selection operations are described in Section 4.6.

Algorithm 1: GLOBAL BORDER TREE

Data: A graph (V, E), a total order $\prec$ on E, and a partition $\mathbf{P} = \{S_0, \ldots, S_k\}$ of V

Result: The global border tree $\mathcal{M}_k$ of $\mathcal{B}^{\prec}$ over $\mathbf{P}$, and the set $\{\mathcal{B}^{\prec}_{S_0}, \ldots, \mathcal{B}^{\prec}_{S_k}\}$ of partial BPH associated to the tiles of $\mathbf{P}$.

1 Call PWK algorithm to compute $\mathcal{B}^{\prec}_{S_0}$
2 $\mathcal{M}_0 :=$ select $\left(\gamma^{\bullet}_V(S_0), \mathcal{B}^{\prec}_{S_0}\right)$
3 **foreach** i **from** 1 **to** k **do**
4 Call PWK algorithm to compute $\mathcal{B}^{\prec}_{S_i}$
5 $\mathcal{S} :=$ select $\left(\gamma^{\bullet}_V(S_i), \mathcal{B}^{\prec}_{S_i}\right)$
6 $\mathcal{M}_i :=$ join $(\mathcal{S},\ \mathcal{M}_{i-1})$

At each step i, we compute $\mathcal{S}$ being the border tree of the partial hierarchy of index i (line 5). Consequently, $\mathcal{M}_i$ is the *partial global border tree* computed on $\cup_{j=0}^{i} \{S_j\}$ so

$$\mathcal{M}_i = \text{select}\left(F, \mathcal{B}^{\prec}_{\cup\{S_j | j \in \{0,\ldots i\}\}}\right)$$
$$\text{with} \quad F = \bigcup_{j=0}^{i} \{\gamma^{\bullet}_V(S_j)\}$$

This latter is obtained by calling *join* on $\mathcal{M}_{i\text{-}1}$ and $\mathcal{S}$ on line 6.

Property 4.4.1. *Let $\mathbf{P} = (S_0, \ldots, S_k)$ be a complete partition of V, let $\mathcal{H}$ be a hierarchy on V, and let $\mathcal{M}$ be the GBT of $\mathcal{H}$ over $\mathbf{P}$. In Algorithm 2, $\mathcal{M}_k = \mathcal{M}$.*

Let us now present Algorithm 2 that allows one to obtain the distribution of the BPH, using the global border tree as an intermediary data structure. Algorithm 2 starts by computing the global border tree and the partial BPHs associated with the tiles of the space by calling Algorithm 1. Then, the tiles are browsed (line 2) and, for each of them, we select its border in the global border tree (line 3) before inserting this selection in the partial BPH associated with the given tile (line 4).

It is noteworthy that the number of calls to the three operations remains linear with respect to the number of tiles in the partition of the space.

Another notable property is that this calculation scheme remains valid regardless of the partition of the space and the order in which tiles are considered as long

Algorithm 2: GBT-based distributed BPH

Data: A graph (V, E), a total order $\prec$ on E, and a partition $\mathbf{P} = \{S_0, \ldots, S_k\}$ of V.
Result: The distribution $\{\mathcal{B}_0, \ldots, \mathcal{B}_k\}$ of the BPH $\mathcal{B}_V^{\prec}$ over $\mathbf{P}$.

```
Call Algorithm 1 to compute the GBT M and the set {B^≺_{S_0}, ..., B^≺_{S_k}} of
  partial BPH associated to the tiles of P
foreach i from 0 to k do
    S := select(γ•_V(S_i), M)
    B_i := insert(S, B^≺_i)
```

as the borders and common neighborhoods of two tiles are known. This computation is also valid as long as, for each tile, we can store in the principal memory of the computer the triplet formed by the global border tree $\mathcal{M}$, the partial BPH $\mathcal{B}_i^{\prec}$, and the element $\mathcal{B}_i$ of the distribution associated with the given tile.

Despite these advantageous characteristics, this computation has the drawback of requiring to load in memory the global border tree which can grow arbitrarily large. In the context of out-of-core computation, a desirable characteristic would be that the amount of memory necessary to a given step of the algorithm is bounded. In other words, one may wish that the number of regions of $\mathcal{S}$, $\mathcal{B}_i^{\prec}$, and $\mathcal{M}$ does not exceed the size of $\mathcal{B}_i$, the element of the distribution associated with the i-th tile of the partition. By the very construction presented in Algorithm 2, it can be seen that $\mathcal{S}$ and $\mathcal{B}_{S_i}^{\prec}$ are always smaller than $\mathcal{B}_i$. However, unfortunately, this is not the case of $\mathcal{M}$, which can be larger than $\mathcal{B}_i$ and larger than the union of multiple elements of the distribution as shown by the following counterexample. We will see in the next section, that if we consider a particular partition of the space, called a causal partition, then a second algorithm to compute the distribution of a BPH can be proposed for which this situation cannot occur, *i.e.*, in this case, the size of the data structure needed at each computation step never exceeds the size of two consecutive elements of the resulting distribution.

Example 4.4.1. Let us consider the set $V = \{x_0, \ldots, x_7\}$ of vertices and the edges $E = \{\{x_i, x_{i+1}\} \mid i \in \{0, \ldots 6\}\}$ ordered such that $\{x_0, x_1\} \prec \{x_1, x_2\} \prec \ldots \prec \{x_6, x_7\}$. The BPH for this ordering $\prec$ is the hierarchy $\mathcal{H} = (\mathcal{H}[0], \ldots, \mathcal{H}[7])$ such that $\mathcal{H}[0] = \{\{x_i\} \mid i \in \{0, \ldots, 7\}\}$ and $\mathcal{H}[7] = V$, and such that, for any $\lambda \in \{1, \ldots, 6\}$, we have:

$$\mathcal{H}[\lambda] = \{e_{\lambda\text{-}1}\} \cup \{\{x_i\} \mid i \in \{\lambda + 1, \ldots, 7\}\} \tag{4.1}$$

where

$$e_\lambda = \{x_i \mid i \in \{0, \ldots, \lambda\}\} \tag{4.2}$$

A representation of $\mathcal{H}$ is shown in Figure 4.6. Let $\mathbf{P}$ be the partition of V such that

$$\begin{aligned} \mathbf{P} = \{P_0 = \{x_0, x_1\}, P_1 = \{x_2, x_3\}, \\ P_2 = \{x_4, x_5\}, P_3 = \{x_6, x_7\}\} \end{aligned} \tag{4.3}$$

The distribution of $\mathcal{H}$ over $\mathbf{P}$ is the set of hierarchies

$$\{\mathcal{L}_i = \text{select}\,(P_i, \mathcal{H}) \mid i \in \{0, \ldots, 3\}\} \tag{4.4}$$

whose region sets are

$$R(\mathcal{L}_0) = \{\{x_0\}, \{x_1\}, e_0, \ldots, e_6\} \tag{4.5}$$
$$R(\mathcal{L}_1) = \{\{x_2\}, \{x_3\}, e_1, \ldots, e_6\} \tag{4.6}$$
$$R(\mathcal{L}_2) = \{\{x_4\}, \{x_5\}, e_3, \ldots, e_6\} \tag{4.7}$$
$$R(\mathcal{L}_3) = \{\{x_6\}, \{x_7\}, e_5, e_6\} \tag{4.8}$$

The GBT $\mathcal{M}$ (bold blue in Figure 4.6) contains the following regions.

$$\begin{aligned} R(\mathcal{M}) = \{\{x_1\}, \{x_2\}, \{x_3\}, \{x_4\}, \{x_5\}, \\ \{x_6\}, e_0, e_1, e_2, e_3, e_4, e_5, e_6\} \end{aligned} \tag{4.9}$$

We can indeed see that $R(\mathcal{M})$ contains 13 regions which is greater than the number of regions in $\mathcal{L}_3$ (since $|R(\mathcal{L}_3)| = 4$). We can further observe that the number of regions in $\mathcal{M}$ (in blue on the figure below) is greater than the sum of the number of regions in the two consecutive elements $\mathcal{L}_2$ and $\mathcal{L}_3$ of the distribution. Indeed, $|R(\mathcal{L}_2)| + |R(\mathcal{L}_3)| = 6 + 4 = 10$.

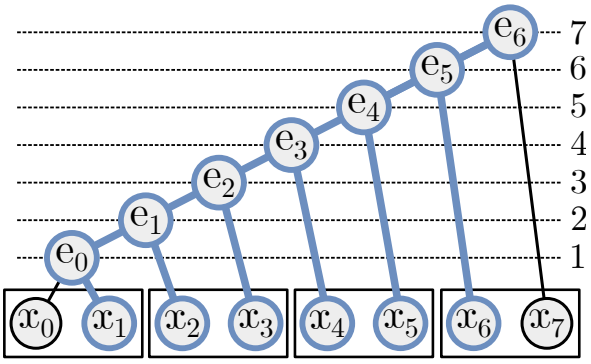


Figure 4.6: Illustration of Counterexample 4.4.1. The partition of the space $\{x_0, \ldots, x_7\}$ into 4 tiles is represented by the black rectangles, the BPH is represented by the black and blue tree while the GBT is highlighted in bold blue.

4.4.2 Causal Partition

In this section, we recall the principle of the out-of-core calculus of the distribution of the BPH over a causal partition introduced in [30]. The presented workflow is designed to limit the size of the data structures simultaneously present in memory. We ensure that at each stage the size of the computed structure is bounded by the size of neighboring elements of the distribution.

In the following, we focus on the specific case of a 4-adjacency graph representing a 2D image that can be divided into slices. This setting is not restrictive, as the method naturally generalizes to higher dimensions. In particular, it extends seamlessly to regular grid graphs, such as the 6-adjacency used for 3D images, as discussed in Section 4.7

Let h and w be two integers representing the height and the width of an image. In the following, the set V is the Cartesian product of $\{0, \ldots, h\text{-}1\} \times \{0, \ldots, w\text{-}1\}$. Thus, any element x of V is a pair $x = (x_i, x_j)$ such that x_i and x_j are the coordinates of x. With the 4-adjacency relation, the set E of edges is equal to $\{\{x, y\} \in V \mid |x_i - y_i| + |x_j - y_j| \leq 1\}$.

A *causal partition* of V is a complete partition $\mathbf{P} = (S_0, \ldots, S_k)$ of V such that each element S_i has at most two neighbors in $\mathbf{P}$, *i.e.*, for any $i \in \{0, \ldots, k\}$, the set $\{j \neq i \mid \gamma(S_i, S_j) \neq \emptyset\}$ has cardinality at most 2. Typical examples of causal partitions are obtained by dividing an image into consecutive vertical slices (see Figure 4.6).

Given any causal partition $\mathbf{P}$ of V, Algorithm 3 allows computing the distribution of the BPH $\mathcal{B}^{\prec}$ over $\mathbf{P}$.

This algorithm can be divided in two parts: a causal and an anti-causal traversal of the slices. Each of these parts relies on the same idea. First, start with

Algorithm 3: Out-of-core BPH for causal partitioning [30].

Data: A graph (V, E), a total order $\prec$ on E, and a causal partition $\mathbf{P} = (S_0, \ldots, S_k)$ of V

Result: The distribution $(\mathcal{B}_0^{\downarrow}, \ldots, \mathcal{B}_k^{\downarrow})$ of the BPH $\mathcal{B}_V^{\prec}$ over $\mathbf{P}$.

1 $\mathcal{B}_0^{\uparrow} := \mathcal{B}_{S_0}^{\prec}$ `// call PWK algorithm`

2 **foreach** i **from** 1 **to** k **do** `// Causal traversal of the slices`

3 Call PWK algorithm to compute $\mathcal{B}_{S_i}^{\prec}$

4 $\mathcal{M}_i^{\uparrow} := \text{join}\left(\text{select}\left(\gamma_{S_i}^{\bullet}(S_{i\text{-}1}), \mathcal{B}_{i\text{-}1}^{\uparrow}\right), \text{select}\left(\gamma_{S_{i\text{-}1}}^{\bullet}(S_i), \mathcal{B}_{S_i}^{\prec}\right)\right)$

5 $\mathcal{B}_i^{\uparrow} := \text{insert}(\text{select}\left(\gamma_{S_{i\text{-}1}}^{\bullet}(S_i), \mathcal{M}_i^{\uparrow}\right), \mathcal{B}_{S_i}^{\prec})$

6 $\mathcal{B}_k^{\downarrow} := \mathcal{B}_k^{\uparrow}$; $\mathcal{M}_k^{\downarrow} := \mathcal{M}_k^{\uparrow}$

7 **foreach** i **from** $k-1$ **to** 0 **do** `// Anti-causal traversal of the slices`

8 $\mathcal{B}_i^{\downarrow} := \text{insert}(\text{select}\left(\gamma_{S_{i+1}}^{\bullet}(S_i), \mathcal{M}_{i+1}^{\downarrow}\right), \mathcal{B}_i^{\uparrow})$

9 **if** $i > 0$ **then** $\mathcal{M}_i^{\downarrow} := \text{insert}(\text{select}\left(\gamma_{S_{i\text{-}1}}^{\bullet}(S_i), \mathcal{B}_i^{\downarrow}\right), \mathcal{M}_i^{\uparrow})$

the causal traversal. Given a causal partition of V into $k+1$ slices, for any i in $\{0, \ldots, k\}$ compute the BPH on S_i with a call to the algorithm PWK (line 3). Then, for any i in $\{1, \ldots, k\}$ *select* the part of this hierarchy containing the vertices adjacent to the previous slice and *join* it with the part of the hierarchy associated with the previous slice containing the vertices adjacent to the current slice, leading to the "*merged tree*" denoted by $\mathcal{M}_i^{\uparrow}$ (line 4). The merged tree is then *inserted* in the current BPH which gives $\mathcal{B}_i^{\uparrow}$ (line 5). The hierarchies $\mathcal{B}_i^{\uparrow}$ associated with slice i misses the information located in slices of higher indices, and consequently only the last local hierarchy $\mathcal{B}_k^{\uparrow}$ belongs to the distribution, *i.e.*, , $\mathcal{B}_k^{\uparrow} = \text{select}\,(S_k, \mathcal{H})$.

In order to compute the whole distribution, and after having spread information in the causal direction, information must be back propagated in the reverse anti-causal direction so that each local hierarchy is enriched with the global context (lines 7 to 9). At each step i, we construct a new merged tree $\mathcal{M}_i^{\downarrow}$ between each pair of neighboring hierarchies, which this time takes into account all the information present in the input data. This merged tree is then inserted into $\mathcal{B}_i^{\uparrow}$ to form $\mathcal{B}_i^{\downarrow}$, which as shown in [30] is the element of the distribution associated with the slice S_i: $\mathcal{B}_i^{\downarrow} = \text{select}\,(S_i, \mathcal{B}^{\prec})$.

The next lemma, which can be deduced from Theorem 17 of [30] and Definition 4.3.2, shows that the intermediary data structures $\mathcal{M}_i^{\uparrow}$ and $\mathcal{M}_i^{\downarrow}$ of Algorithm 3 can be expressed in a way similar to the intermediary data structures $\mathcal{M}_i$ and $\mathcal{M}$ of Algorithms 1 and 2. Then, this allows us to compare these intermediary data

structures and to deduce notably that the intermediary data structures of Algorithm 3 are always smaller than those of Algorithms 1 and 2 (see Property 4.4.1).

Lemma 4.4.1. *Let $(S_0, \ldots, S_k)$ be a causal partition of V. Let i be any element in $\{1, \ldots, k\}$. The following statements hold true:*

1. $\mathcal{M}_i^{\uparrow} = select\left(\delta^{\bullet}(\gamma(S_{i\text{-}1}, S_i)), \mathcal{B}^{\prec}_{\cup\{S_j | j \in \{0,\ldots i\}\}}\right)$;
2. $\mathcal{M}_i^{\downarrow} = select\left(\delta^{\bullet}(\gamma(S_{i\text{-}1}, S_i)), \mathcal{B}^{\prec}_{V}\right)$.

From Lemma 4.4.1, we deduce the following property which states that the intermediary data structures of *Algorithm* 3 are subparts of those of Algorithms 1 and 2.

Property 4.4.2. *Let $\mathbf{P} = (S_0, \ldots, S_k)$ be a causal partition of V. Let i be any element in $\{0, \ldots, k\}$. Let $\mathcal{M}$ be the Global Border Tree built on V over $\mathbf{P}$ and $\mathcal{M}_i$ be the Global Border Tree on $\cup_{j=0}^{i}\{S_j\}$. Then, we have $\mathcal{M}_i^{\uparrow} \sqsubseteq \mathcal{M}_i$ and $\mathcal{M}_i^{\downarrow} \sqsubseteq \mathcal{M}$.*

By considering Property 4.4.2, we can derive some important implications regarding the sizes of certain components within the GBT structures.

This structural inclusion has direct implications for the memory requirements and computational complexity of the algorithm. The next corollary formalizes the bounds on the sizes of the intermediate trees used in Algorithm 3 in comparison to those from the reference algorithms.

Corollary 4.4.1. *The sizes of the merged trees computed in Algorithm 3 are bounded as follows:*

$$\max_{1\leq i\leq k} |\mathcal{M}_i^{\uparrow}| \leq \max_{1\leq i\leq k} |\mathcal{M}_i|$$

$$\max_{1\leq i\leq k} |\mathcal{M}_i^{\downarrow}| \leq |\mathcal{M}|$$

The proofs of Property 4.4.2 and Corollary 4.4.1 are given in Appendix 8.2, and Appendix 8.3 respectively.

The next property, which is the main result of this section, establishes that the size of the intermediary data structures of Algorithm 3 is, at each step of the algorithm, smaller than the size of the data structures that we seek to build at this step: in the step involving the construction of the element $\mathcal{B}_i^{\prec}$ of the distribution of the BPH, the size of the auxiliary data structures never exceeds the size of the two consecutive elements $\mathcal{B}_{i\text{-}1}^{\prec}$ and $\mathcal{B}_i^{\prec}$. This property does not hold for the generic partition approach, as shown by Example 4.4.1.

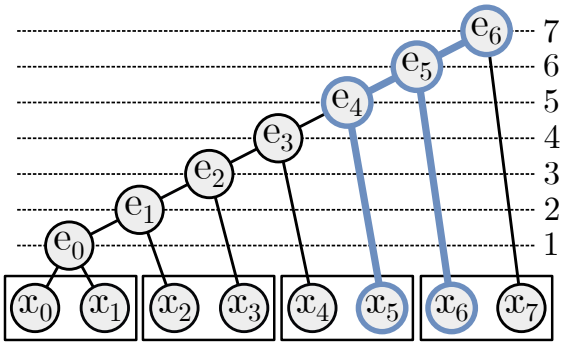


Figure 4.7: Given a partition of the space in 4 slices, the BPH in black and the merged tree of the last two slices in bold blue.

Property 4.4.3. *Let $(S_0, \ldots, S_k)$ be a causal partition of V and let $\mathcal{H}$ be the BPH built on V. Let i be any element in $\{1, \ldots, k\}$.*
Given $n_i = |select(S_{i\text{-}1}, \mathcal{H})| + |select(S_i, \mathcal{H})|$, we have $|\mathcal{M}_i^{\uparrow}| \leq n_i$ and $|\mathcal{M}_i^{\downarrow}| \leq n_i$.

Property 4.4.3 does not hold true in general when using Algorithms 1 and 2. A proof is given in Appendix 8.4.

An insight into the proof of this property is observed in Figure 4.7. We can see the merged tree for the last two slices in bold blue which contains 5 nodes: 2 leaves and 3 non-leaf nodes. As a recall, the GBT associated with the very same BPH and partition presented in Figure 4.6 contains 13 nodes: 6 leaves and 7 non-leaf nodes. Then, the number of regions in the GBT is comparable to that of the entire BPH (15 nodes) while the merged tree is closer, in size, to a subtree of the BPH associated with one of its neighboring tiles.

Hence, unlike the calculation described in the previous section, only the information associated with two neighboring tiles is required at each stage. The merged tree computed at each iteration of both causal an anti-causal traversal is systematically smaller than equal to the GBT (strictly smaller when $|\mathbf{P}| > 2$). This makes this algorithm a more scalable method than the previous approach, at the cost of a higher number of steps. These practical considerations are addressed in Section 7.4.

4.5 Data Structures

In this section, we discuss the data structures used in the algorithms defined in the following sections. We first present the data structures used to represent the BPH embedding all the necessary information to manage such structure in an out-

of-core context. Then we give the representation of the graph with a specific focus on 3D applications.

4.5.1 Representation of the BPH

The data structure representing a local hierarchy $\mathcal{H}$ assumes that the nodes of the hierarchy are indexed in a particular order and relies on three "attributes":

- $\mathcal{H}$`.map`: Maps the indices from the local context (a given slice) to the global context (the entire graph).
- $\mathcal{H}$`.par`: Represents the parent array, encoding the parent-child relationships between tree nodes.
- $\mathcal{H}$`.weights`: Assigns a weight to each non-leaf node, corresponding to the one of its associated building edge.

More precisely, given a binary partition hierarchy $\mathcal{H}$ with n regions, each region is associated to an unique integer between 0 and n - 1. Moreover, this indexing of the regions of $\mathcal{H}$ follows a topological order such that:

1. Any leaf region is indexed before any non-leaf region;
2. Two leaf regions $\{x\}$ and $\{y\}$ are sorted with respect to an arbitrary order on the element V. In practice, we use the *raster scan order* of the pixels. Thus, $\{x\}$ has an index lower than $\{y\}$ if x is before y with respect to the raster scan order;
3. Two non-leaf regions are sorted according to their rank, *i.e.*, the order of their building edges for $\prec$.

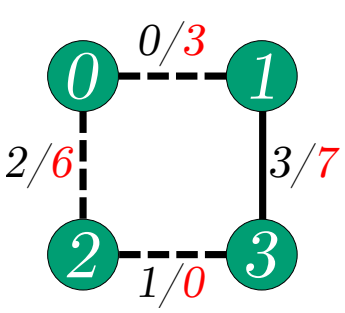
This order can be seen as an extension of the order $\prec$ on E to the set $V \cup E$ that enables to efficiently browse the nodes of a hierarchy according to their scale of appearance in the hierarchy; and to efficiently match regions of V with the leaves of the hierarchy.

By abuse of notation, this extended order is also denoted by $\prec$ in the following. Any hierarchy resulting from the algorithms presented in the next sections satisfies this property and ensures the efficiency of the implementation through the major advantages of facilitating the searching of a given region or of a list of regions in a hierarchy.

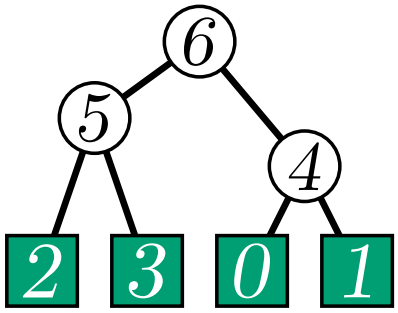
To keep track of the global context, a link between the indices in the local tree and the global indices in the whole graph is stored in the form of an array `map`, which associates any region $R \in \mathcal{H}$ with:

- **if R is a leaf region:** the vertex x of the graph G such that $R = \{x\}$, *i.e.*, `map[i]` $= x$;
- **otherwise (if R is a non-leaf region):** its building edge, *i.e.*, `map[i]` $= \mu^{\prec}(R)$.

The parent relation of the hierarchy is stored thanks to an array `par` such that `par[i]=j` if the region of index j is the parent of the region of index i.



(a) Edge-weighted graph G



(b) Binary Partition Hierarchy $\mathcal{H}$

Node Index	0	1	2	3	4	5	6
`par`	4	4	5	5	6	6	6
`map`	2	3	0	1	1	0	2
`weights`	-	-	-	-	0	3	6

(c) Representation in memory

Figure 4.8: (a) An edge-weighted graph with edge/vertex index in black/white and edge weights in red, (b) its BPH where each node is associated with a local index, and (c) the data structures used to represent the BPH.

The binary partition hierarchy is built for a particular ordering $\prec$ of the edges of G. In practice, this ordering is induced by weights computed over the edges of G. To this end, we store an array `weights` of size $|\mathcal{R}^{\star}(\mathcal{H})|$, *i.e.*, the number of non-leaf regions, elements such that, for every region R in $\mathcal{R}^{\star}(\mathcal{H})$ of index i, `weights[i]` is the weight of the building edge $\mu^{\prec}(R)$ of region R. The edges can then be compared according to the following total order induced by the weights: we set $u \prec v$ if the weight of u is less than the one of v or if u and v have equal weights but u comes before v with respect to the raster scan order.

We can see such representation in Figure 4.8. Each vertex of the base graph or edge of the MST is reindexed in the associated BPH. The node 2 of G correspond to the leaf region 0 in the BPH with $\mathcal{H}$.`map[0]` $= 2$. We can see as previously

mentioned that the array `weight` does not have an entry for the leaf regions of the BPH.

4.5.2 Graph Management

In the upcoming algorithm, efficient access to specific parts of the graph is essential. For instance, when processing two chunks of the base graph, we often need to access their common neighborhood. This section introduces mechanisms for accessing various relevant sub-parts of the graph, using the example of a 6-adjacency graph. Importantly, the concepts presented here generalize to any regular graph, *i.e.*, any graph in which the adjacency of each vertex is defined by a fixed structuring element.

The 6-adjacency is the three-dimensional extension of the 4-adjacency used in 2D. It defines each vertex as connected to its four nearest neighbors within the same slice, similar to the 4-adjacency illustrated in Figure 4.8a, as well as its two closest neighbors in the adjacent slices (one front and one behind). A commonly used alternative is the 26-adjacency, which extends the 8-adjacency to 3D by including diagonal neighbors in all directions.

While 8- and 26-adjacency can yield slightly improved results in many algorithms, they come at a higher cost in memory and computational resources. Additionally, they introduce topological ambiguities, particularly when constructing a Khalimsky grid which is a crucial structure for saliency visualization.

We begin by considering the vertices of the graph. These are stored in *raster scan order* (also known as *row-major order*), meaning the elements are traversed row by row, then column by column, and finally slice by slice, as illustrated in Figure 4.9. The 3D grid is represented as an array of dimensions $D \times H \times W$, corresponding to the width, height, and depth of the volume.

Let n denote the index of a given vertex in this 1D array representation. Assuming n is not on a border, the indices of its 6-adjacent neighbors can be computed directly from the grid dimensions and its position within the array.

$$\text{top} = n - W \qquad \text{bottom} = n + W \tag{4.10}$$

$$\text{left} = n - 1 \qquad \text{right} = n + 1 \tag{4.11}$$

$$\text{back} = n - W \cdot H \qquad \text{front} = n + W \cdot H \tag{4.12}$$

In regular graphs, there exist multiple strategies for indexing edges. A generic and elegant method involves using a structuring element, modeled as an oriented graph. In this representation, the origin corresponds to a reference node, and its successors denote neighboring nodes, identified by their relative positions. Edges are then defined by connecting the reference node to each of its "valid" neighbors,

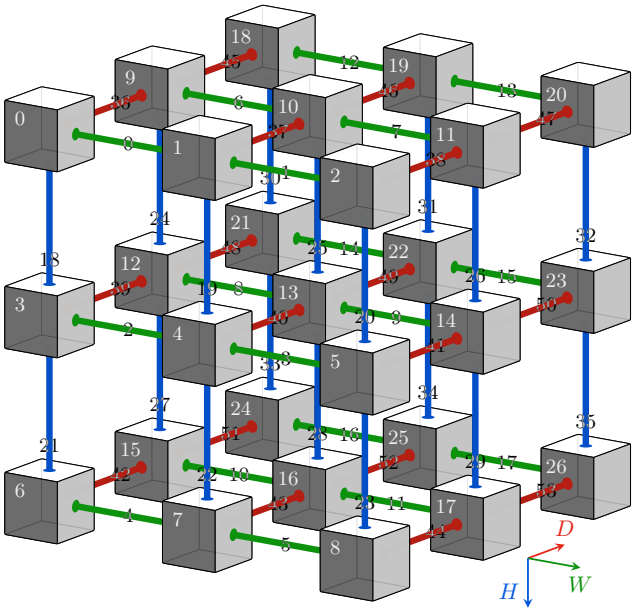


Figure 4.9: A 6-adjacency grid of size $3 \times 3 \times 3$ with vertices indexed in raster scan order. Green horizontal edges are indexed first, followed by blue vertical edges, and finally red depth edges.

those that lie within the domain of the grid. The index of a given edge is computed from two components: *i)* an arbitrary but fixed order on the edges of the structuring element, and *ii)* the index of the reference node.

While this indexing strategy is elegant and cache-compliant, it is not optimal for our specific needs. In the current setting, we are particularly interested in quickly computing the indices of a node's adjacent edges. To this end, we adopt a more direct method: edges are indexed in *raster scan order*, first assigning indices to all horizontal edges, then to vertical ones, and finally to edges in the depth direction. This indexing scheme is illustrated in Figure 4.9 with green edges first, then blue and finally the red ones.

We begin by recalling the mapping from a 1D index n to its corresponding 3D coordinates (z, y, x) in a grid of size $D \times H \times W$:

$$x = n \bmod W \tag{4.13}$$

$$y = \left\lfloor \frac{n \bmod (W \cdot H)}{W} \right\rfloor \tag{4.14}$$

$$z = \left\lfloor \frac{n}{W \cdot H} \right\rfloor \tag{4.15}$$

Given a node n located at position (z, y, x), the indices of its incident edges in

the 6-adjacency configuration are calculated as follows:

$$\text{top} = N^H + z \cdot (W \cdot (H-1)) + (y-1) \cdot W + x \tag{4.16}$$

$$\text{bottom} = N^H + z \cdot (W \cdot (H-1)) + y \cdot W + x \tag{4.17}$$

$$\text{left} = n - \left\lfloor \frac{n}{W} \right\rfloor - 1 \tag{4.18}$$

$$\text{right} = n - \left\lfloor \frac{n}{W} \right\rfloor \tag{4.19}$$

$$\text{back} = N^{HV} + n - (W \cdot H) \tag{4.20}$$

$$\text{front} = N^{HV} + n \tag{4.21}$$

Here, N^H denotes the total number of horizontal edges in the graph, and N^{HV} is the combined number of horizontal and vertical edges.

It is also possible to give source and target vertex of an edge given its index. Thus for an edge $u = \{s, t\}$ of index e we proceed as follow:

$$s = \begin{cases} e \bmod (W-1) + \lfloor \frac{e}{W-1} \rfloor \bmod H + \lfloor \frac{e}{(W-1)\cdot H} \rfloor \cdot H \cdot W, & \text{if } e < N^H \\ e \bmod ((W) \cdot \lfloor \frac{e}{W \bmod (H-1)}) \cdot W + \lfloor \frac{e}{(H-1)\cdot W} \rfloor \cdot H \cdot W, & \text{else if } e < N^{HV} \\ e - N^{HV}, & \text{else} \end{cases} \tag{4.22}$$

$$t = \begin{cases} s+1, & \text{if } e < N^H \\ s+W, & \text{else if } e < N^{HV} \\ s + H \cdot W, & \text{else} \end{cases} \tag{4.23}$$

As a consequence, given the `map` associated with a region of a BPH, it is possible retrieve all the information related to its building edge.

This explicit indexing scheme allows for fast and efficient access to edge data, which is essential for the out-of-core computation routines that follow. It provides a practical compromise between memory layout efficiency and direct neighborhood access.

4.6 Algorithms

In this section, we present the algorithms underlying the three core algebraic operations of our framework: *select*, *join*, and *insert*. Each operation is described with detailed pseudo-code and accompanied by a complexity analysis. These algorithms are designed to operate efficiently on the data structures introduced in the previous section. Together, they form the computational backbone for constructing the distribution of the binary partition hierarchy.

4.6.1 Select

In this part, we give Algorithm 4 to compute the result of the *select* operation. This operation consists in "selecting" all regions of a given hierarchy $\mathcal{H}$ intersecting a subset X of the space.

Select algorithm proceeds in 3 steps:

1. *Lines 5-9.* Mark any leaf-node of $\mathcal{H}$ that corresponds to an element of X, *i.e.*, any leaf-region $\{x\}$ with $x \in X$;

2. *Lines 11-15.* Traverse the hierarchy from leaves to root and mark any node that is a parent of a marked node; computing its new index in the output hierarchy with `sNodeIdx`;

3. *Lines 18-24.* Build the hierarchy $\mathcal{S}$ whose nodes are the marked nodes of $\mathcal{H}$.

In Algorithm 4, we assume that X is sorted and that $X \subseteq gr(\mathcal{H})$, which is always the case in Algorithm 3. For each element $X\texttt{[}i\texttt{]}$ of X, we search for the index j of a leaf of $\mathcal{H}$ mapped to $X\texttt{[}i\texttt{]}$, *i.e.*, such that $\mathcal{H}\texttt{.map[j]} = X\texttt{[}i\texttt{]}$. To this end, it is necessary to make a traversal of the leaves of $\mathcal{H}$. As mentioned before, the leaves correspond to the first indices by construction. The first step can then be performed in linear time with respect to the number of leaf regions of $\mathcal{H}$. The second step consists in traversing the whole hierarchy from leaves to root in order to mark every region of $\mathcal{H}$ which belongs to select $(X, \mathcal{H})$ *i.e.*, regions parent of a marked one. The complexity of this step is therefore linear with respect to the number of regions of $\mathcal{H}$. Finally, the last step boils down to extracting the hierarchy select $(X, \mathcal{H})$ from the marked nodes. For this, a new hierarchy is created by traversing $\mathcal{H}$ again. As the traversal is done by increasing order of index, the properties relating to the weights of the building edges and order of appearance of regions are preserved. The complexity of this last step is linear with respect to the number of regions of $\mathcal{H}$. Thus, Algorithm 4 has a linear $O(n)$ complexity, where n is the number of regions of $\mathcal{H}$.

4.6.2 Join

Intuitively, this operation merges two hierarchies according to their common neighborhood, that is the set of edges linking their grounds. In [46], the authors proposed an algorithm that can be used to successively add edges of the common neighborhood at their correct rank. Essentially, this involves updating the hierarchy by traversing the branches associated with the endpoints of each added edge. The worst-case complexity is then linear with respect to the size of the hierarchy for adding a single edge. Thus the overall complexity of such join procedure would

Algorithm 4: SELECT

```
   Data: A hierarchy H, a subset X of V
 1 . Result: The hierarchy S = select(X, H)
 2 . Initialize an array sNodeIdx to -1 for every region of H
 3 i := 0                                  // i iterates over X
 4 j := 0                                  // j over the leaves of H
 5 while i < |X| and j < |H.leaves| do
 6     if X[i] = H.map[j] then
 7         sNodeIdx[j] := 0
 8         i := i + 1
 9     j := j + 1
10 count := 0
11 foreach n from 0 to |H|-2 do
12     if sNodeIdx[n] ≠ -1 then
13         sNodeIdx[n] := count
14         sNodeIdx[H.par[n]] := 0
15         count := count + 1
16 sNodeIdx[H.root] := count
17 n_S := 0
18 foreach n from 0 to |H|-1 do
19     if sNodeIdx[n] ≠ - then
20         S.par[n_S] := sNodeIdx[H.par[n]]
21         S.map[n_S] := H.map[n]
22         if n ∈ R*(H) then
23             S.weight[n_S - |X|] := H.weight[n_S - |H.leaves|]
24     n_S := n_S + 1
25 return S
```

be $O(kn)$ where n is the number of regions of the considered hierarchies and k is the number of edges in the common neighborhood. In this section, we drop the multiplicative dependency in the size of the neighborhood F at the cost of introducing a sorting of F and we present an algorithm whose complexity is quasi-linear with respect to the size n of the hierarchies and linearithmic with respect to the number k of edges in F.

A detailed presentation of the proposed algorithm is given in Algorithm 5 which calls auxiliary functions presented in Algorithm 6. Intuitively, in order to compute the join of two hierarchies $\mathcal{X}$ and $\mathcal{Y}$, Algorithm 5 consists in "emulating" PWK

Algorithm 5: JOIN

Data: $\mathcal{X} and\ \mathcal{Y}$: two hierarchies, F common neighborhood of $gr(\mathcal{X})$ and $gr(\mathcal{Y})$.
Result: A collection $Q_D = \text{join}\,(\mathcal{X},\ \mathcal{Y})$

```
1  foreach node n_i of X do Q_D.MakeSet(i)
2  foreach node n_i of Y do Q_D.MakeSet(i + |X.leaves|)
3  aDescendent(X, 0)
4  aDescendent(Y, |X.leaves|)
5  F :=sort(F)
6  i_1 := |X.leaves|; i_2 := |Y.leaves|; i_3 := 0
7  while i_1 < |X| or i_2 < |Y| or i_3 < |F| do
8      if F[i_3] ≺ X.map[i_1] and F[i_3] ≺ Y.map[i_2] then
9          (x, y) := F[i_3]; m := F[i_3]; w := weight(F[i_3]); i_3 += 1
10     else if X.map[i_1] ≺ Y.map[i_2] then
11         (x, y) := X.desc[i_1]; m := X.map[i_1]; w := X.weight[i_1]; i_1 += 1
12     else
13         (x, y) := Y.desc[i_2]; m := Y.map[i_2]; w := Y.weight[i_2]; i_2 += 1
14     c_x := Q_D.FindCanonical(x); c_y := Q_D.FindCanonical(y)
15     if c_x! = c_y then
16         n := Q_D.Union(c_x, c_y); Q_D.map[n] := m
17         Q_D.weight[n − (|X.leaves| + |Y.leaves|)] := w
```

algorithm on the graph obtained from:

1. edges associated with the non-leaf nodes of $\mathcal{X}$ and of $\mathcal{Y}$ and;
2. edges in the common neighborhood F of $\mathcal{X}$ and $\mathcal{Y}$.

Therefore, all these edges are considered in increasing order with respect to $\prec$ and, for each edge, it is decided if this edge must be considered or not in the creation process of the join hierarchy. The decision is made based on the potential creation of a cycle if this edge was added during the minimum spanning tree creation process.

A main observation can be made to highlight the difference between the situation encountered in the contexts of join algorithm and PWK algorithm: in the context of join, some edges, which are associated to the nodes of the hierarchies $\mathcal{X}$ and $\mathcal{Y}$, are made of vertices that do not belong to the underlying space (*i.e.*, the common neighborhood of the slices supporting the grounds of $\mathcal{X}$ and $\mathcal{Y}$).

To process them, we compute a pair attribute `desc` for each non-leaf nodes of $\mathcal{X}$ and $\mathcal{Y}$. Roughly speaking, this attribute assigns a source and target vertex

to each of these edges. For a given node, the source is determined as a leaf of the sub-tree rooted in its first child (arbitrarily chosen), whose `map` necessarily refers to a node of the underlying space. On the other hand, the target is either another a leaf of the sub-tree rooted in its second child or, alternatively, assigned a value of -1 in cases where the node has only one child (*e.g.*, region of $\mathcal{X}$ of rank 4 in Figure 4.3).

When such edge is found, the standard algorithm can be shortcut leading to a modified version of the PWK auxiliary functions presented in Algorithm 6 at line 12. This test detects the edges for which a shortcut must occur based on the value of `desc`. Indeed, if the target of an edge is equal to -1, no cycle can be created. As a result, the node is added as the parent of the canonical element associated with the source of this edge. This attribute is pre-computed for every node of $\mathcal{X}$ and of $\mathcal{Y}$ by the auxiliary function `aDescendent`. Overall, the following steps are performed in Algorithm 5:

- *Lines 1-2.* Initialize the Union-find data structures;
- *Lines 3-4.* Compute the attribute `desc` for both $\mathcal{X}$ and $\mathcal{Y}$;
- *Lines 5 to 13.* Browse the edges in increasing order. Observe that it implies sorting the edges in the common neighborhood F of $\mathcal{X}$ and $\mathcal{Y}$ in increasing order for $\prec$ (non-leaf nodes of $\mathcal{X}$ and $\mathcal{Y}$ are already sorted by construction);
- *Lines 15-17* Apply PWK steps, calling the modified version of the auxiliary functions.

The first step complexity is linear with respect to the number of elements of $gr(\mathcal{X}) \cup gr(\mathcal{Y})$. The second step uses the auxiliary function `aDescendent` to compute attributes `desc` for both $\mathcal{X}$ and $\mathcal{Y}$ consisting in a leaves to root traversal which gives a linear complexity with respect to the number of regions of each hierarchy. It should be noted that this last function takes a parameter *shift* which allows to index the leaves of the second hierarchy after those of the first. Third step requires to sort the edges of F with respect to $\prec$ before browsing the edges which implies a complexity of $O(k \times log(k) + |\mathcal{X}| + |\mathcal{Y}|)$ with k the number of edges in F. The fourth step is equivalent to PWK algorithm in terms of complexity. Then, its complexity is $O(m \times \alpha(n))$ where m is sum of the number of edges in F and the number of non-leaf nodes of $\mathcal{X}$ and $\mathcal{Y}$, where n is the number of leaf nodes in $\mathcal{X}$ and $\mathcal{Y}$ and where $\alpha()$ is the inverse Ackermann function which grows sub-logarithmically.

Algorithm 6: AUXILIARY FUNCTIONS FOR JOIN ALGORITHM

```
   // The functions called hereafter on Q_T and Q_BT are those
      described in [81]
 1 Procedure Q_D.MakeSet(q)
 2 |   Q_D.Root[q] := q
 3 |   Q_BT.MakeSet(q)
 4 |   Q_T.MakeSet(q)
 5
 6 Function Q_D.FindCanonical(q)
 7 |   return Q_T.FindCanonical(q)
 8
 9 Function Q_D.Union(c_x, c_y)
10 |   t_u := Q_D.Root[c_x]
11 |   Q_BT.par[t_u] := Q_BT.size
12 |   if c_y = -1 then  Q_D.Root[c_x] := Q_BT.size
13 |   else
14 |   |   t_v := Q_D.Root[c_y]
15 |   |   Q_BT.par[t_v] := Q_D.size
16 |   |   c := Q_T.Union(c_x, c_y)
17 |   |   Q_D.Root[c] := Q_BT.size
18 |   end
19 |   Q_BT.MakeSet(Q_BT.size)
20 |   return Q_D.size - 1
21
22 Function aDescendent(H, s)
23 |   foreach node n of H do  H.desc[n]:= (-1, -1)
24 |   foreach leaf node n of H do  H.desc[n].first:= n + s
25 |   foreach non-root non-leaf node n of H in increasing order for ≺ do
26 |   |   p := H.par[n]
27 |   |   if H.desc[p].first= -1 then H.desc[p].first:= H.desc[n].first
28 |   |   else H.desc[p].second:= H.desc[n].first
29 |   end
```

4.6.3 Insert

In this part, we present Algorithm 7. The operation *insert* aims to enrich a hierarchy $\mathcal{Y}$ by inserting the regions of another hierarchy $\mathcal{X}$ by computing a new hierarchy $\mathcal{Z} = \text{insert}(\mathcal{X}, \mathcal{Y})$ (called the insertion of $\mathcal{X}$ into $\mathcal{Y}$). Regions within $\mathcal{X}$ are inserted at the correct rank in the output hierarchy and, some regions of $\mathcal{Y}$ are discriminated and discarded from the output hierarchy.

As stated in Definition 4.3.2, $\mathcal{X}$ must be *insertable* into $\mathcal{Y}$, this assumption holds true at each call to *insert* in Algorithm 2 and Algorithm 3.

From a high-level point of view, insert algorithm proceeds in two main steps:

1. *Lines 3-15.* Identify and renumber the regions of $\mathcal{X}$ and $\mathcal{Y}$ that belong to $\mathcal{Z}$ and store the correspondences between their current indices and their new in $\mathcal{Z}$. It can be observed that this step is necessary since a region of $\mathcal{Z}$ can be duplicated in both $\mathcal{X}$ and $\mathcal{Y}$ and that some regions of $\mathcal{Y}$ are discarded from $\mathcal{Z}$. In order to perform this step, the regions of $\mathcal{X}$ and $\mathcal{Y}$ are simultaneously browsed in increasing order for $\prec$. The correspondences between the regions of the hierarchies are stored in three arrays: $\text{C}_{\mathcal{X}\to\mathcal{Z}}$, $\text{C}_{\mathcal{Y}\to\mathcal{Z}}$, and $\text{C}_{\mathcal{Z}\to\mathcal{X},\mathcal{Y}}$;

2. *Lines 17-28.* Build the parenthood relation (`par`) of the hierarchy $\mathcal{Z}$ using the parenthood relation of the hierarchies $\mathcal{X}$ and $\mathcal{Y}$ and the correspondences between the regions of the hierarchies. At the same time, we also build the attributes `map` and `weight` associated with $\mathcal{Z}$.

During the first step, each region of the two hierarchies $\mathcal{X}$ and $\mathcal{Y}$ is considered once and processed with a limited number of constant-time instructions. Thus, the overall time complexity of Lines 3-15 is linear with respect to the number of nodes of $\mathcal{X}$ and $\mathcal{Y}$. The worst-case complexity of the second step is also linear with respect to the number of nodes of $\mathcal{X}$ and $\mathcal{Y}$ since $\mathcal{Z}$ contains at most all regions of each of hierarchy. Thus, the overall complexity of Algorithm 7 is $O(|\mathcal{X}| + |\mathcal{Y}|)$.

Algorithm 7: INSERT

Data: Two hierarchies $\mathcal{X}$ and $\mathcal{Y}$ such that $\mathcal{X}$ insertable into $\mathcal{Y}$.
Result: $\mathcal{Z}$: the hierarchy insert($\mathcal{X}, \mathcal{Y}$)

$x := 0;\ y := 0;\ z := 0;$ `// indices for the nodes/regions of` $\mathcal{X}$, $\mathcal{Y}$, `and` $\mathcal{Z}$
Initialize an array InZ of $|\mathcal{Y}|$ Booleans to true (resp. to false) for every leaf (resp. non-leaf) region of $\mathcal{Y}$
while $x < |\mathcal{X}|$ **or** $y < |\mathcal{Y}|$ **do**
    if $x < |\mathcal{X}|$ **and** $y < |\mathcal{Y}|$ **and** $\mathcal{X}.\text{map}[x] = \mathcal{Y}.\text{map}[y]$ **then**
        `// Duplicate region` (x, y) `found in` $\mathcal{X}$ `and` $\mathcal{Y}$, `keep (and renumber) it in` $\mathcal{Z}$
        $\text{C}_{\mathcal{X}\to\mathcal{Z}}[x] := z;\ \text{C}_{\mathcal{Y}\to\mathcal{Z}}[y] := z;\ \text{C}_{\mathcal{Z}\to\mathcal{X},\mathcal{Y}}[z] := (x, y)$
        $x \mathrel{+}= 1;\ y \mathrel{+}= 1;\ z \mathrel{+}= 1$
    else if $\mathcal{Y}.\text{map}[y] \prec \mathcal{X}.\text{map}[x]$ **then**
        if $InZ[y] = true$ **then** `// Keep (and renumber) region` y `in` $\mathcal{Z}$
            $InZ[\mathcal{Y}.\text{par}[y]] := true$
            $\text{C}_{\mathcal{Y}\to\mathcal{Z}}[y] := z;\ \text{C}_{\mathcal{Z}\to\mathcal{X},\mathcal{Y}}[z] := (-1, y)$
            $y \mathrel{+}= 1;\ z \mathrel{+}= 1$
        else $y \mathrel{+}= 1$ `// Discard region` y `from` $\mathcal{Z}$
    else `// Keep (and renumber) region` x `in` $\mathcal{Z}$
        $\text{C}_{\mathcal{X}\to\mathcal{Z}}[x] := z;\ \text{C}_{\mathcal{Z}\to\mathcal{X},\mathcal{Y}}[z] := (x, -1)$
        $x \mathrel{+}= 1;\ z \mathrel{+}= 1$
$\mathcal{Z} :=$ initialize a tree structure with $n_{\mathcal{Z}} = z$ nodes
foreach z **from** 0 **to** $n_{\mathcal{Z}}$ **do**
    $(x, y) := \text{C}_{\mathcal{Z}\to\mathcal{X},\mathcal{Y}}[z]$
    if $x \neq -1$ **then**
        $\mathcal{Z}.\text{map}[z] := \mathcal{X}.\text{map}[x]$
        if $[x] = \mathcal{X}.\text{root}$ **then** $\mathcal{Z}.\text{par}[z] := z$
        else $\mathcal{Z}.\text{par}[z] := \text{C}_{\mathcal{X}\to\mathcal{Z}}[\mathcal{X}.\text{par}[x]]$
        if $z \geq |\mathcal{X}.leaves|$ **then**
            $\mathcal{Z}.\text{weight}[z - |\mathcal{X}.leaves|] := \mathcal{X}.\text{weight}[x - |\mathcal{X}.leaves|]$
    else
        $\mathcal{Z}.\text{map}[z] := \mathcal{Y}.\text{map}[y]$
        if $y = \mathcal{Y}.\text{root}$ **then** $\mathcal{Z}.\text{par}[z] := z$
        else $\mathcal{Z}.\text{par}[z] := \text{C}_{\mathcal{Y}\to\mathcal{Z}}[\mathcal{Y}.\text{par}[y]]$
        if $z \geq |\mathcal{Y}.leaves|$ **then**
            $\mathcal{Z}.\text{weight}[z - |\mathcal{Y}.leaves|] := \mathcal{Y}.\text{weight}[x - |\mathcal{Y}.leaves|]$

4.7 Experiments

In this section, we assess the performances of our framework over 3D 8-bit images. We conduct a comparative analysis of various computation patterns and partitioning strategies. The experiments are executed on a workstation equipped with an Intel Xeon processor running at 3.90GHz and 256GB RAM. Processed images are stored in HDF5 format [117] to facilitate tiled reading while intermediate and output data structures are managed through Zarr [76] using the Blosc compressor.

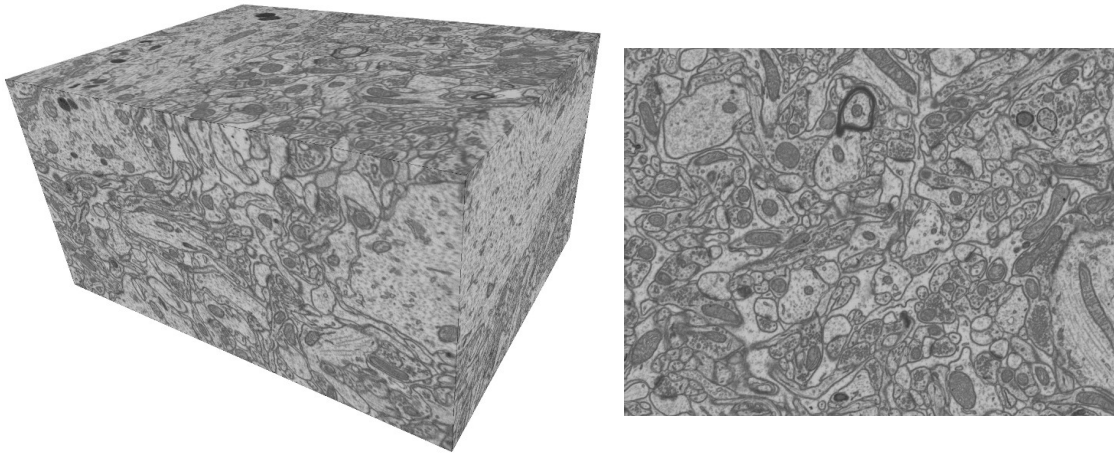
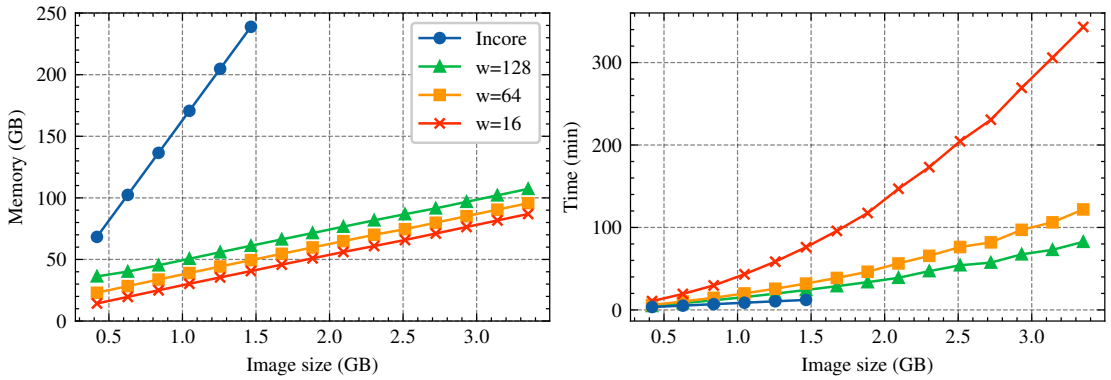


Figure 4.10: FIB-SEM volume of the CA1 hippocampus region of the brain [62]. The left image shows a 3D rendering of the volume while the right image is a slice of the volume. The image is composed of $2048 \times 1536 \times 1065$ voxels with an isotropic size of 5 nm.

The experiments are conducted on real images, specifically a FIB-SEM volume of the CA1 hippocampus region of the brain with dimensions presented in Figure 4.10 of approximately 3.3 gigavoxels. It should be noted that it is not possible to run the classic algorithm for this size of volume on the workstation on which we carried out the experiments. At least 8 slices are needed to be able to compute the distribution.

The first experiment consists of selecting several subsets of the test image of increasing size in order to observe memory consumption trends relative to the size of the input image. In this way, we can compare the in-core algorithm and the calculation of the distribution with the causal algorithm presented in Algorithm 3 varying size and number of tiles. Indeed, in Figure 4.11 the first data point on the green curve corresponds to a causal partition of 2 tiles of $128 \times 1536 \times 1065$ while for the orange curve this corresponds to 4 tiles with a width of 64 voxels, and for the red curve 16 tiles with a width of 16 voxels. Then, for each subsequent data



(a) Peak Memory Consumption (b) Execution Time

Figure 4.11: Comparative analysis of peak memory consumption and execution time between the In-Core algorithm and the out-of-core causal given as a function of the image size. The variable w correspond to the slice width.

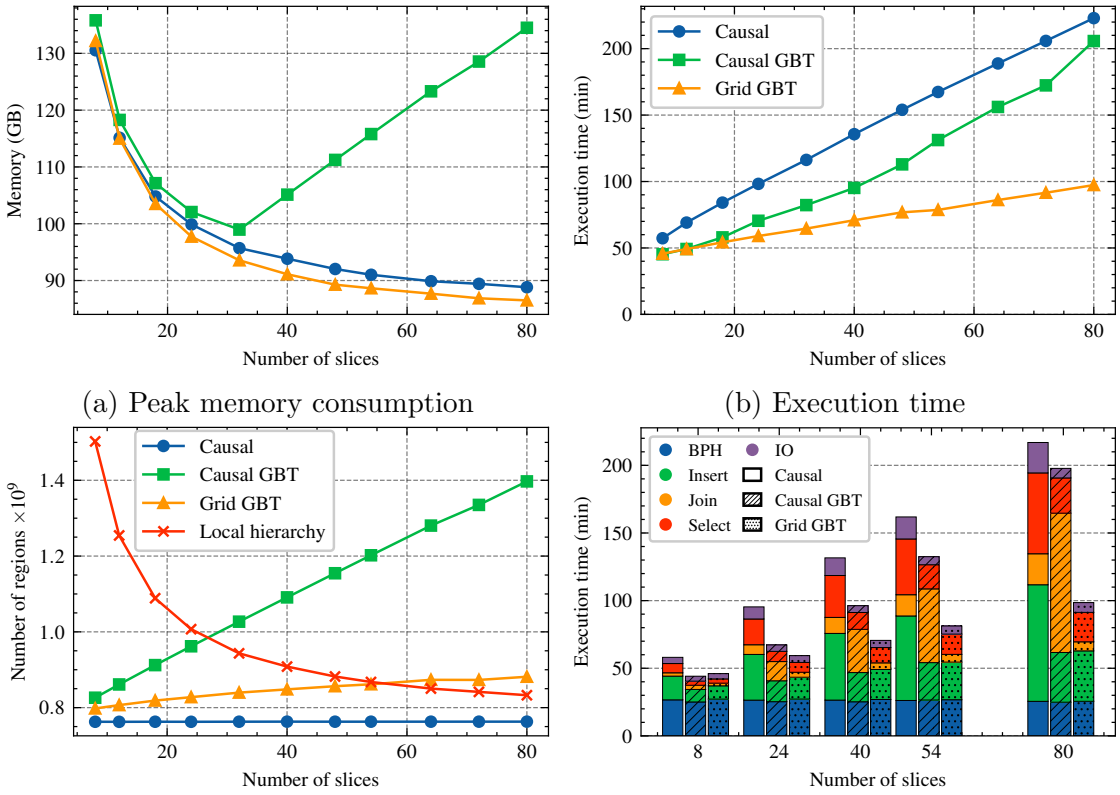
point, the number of slices is incremented by 1, 2, and 8, respectively. Tendencies depicts the evident disparity in peak memory utilization between the out-of-core and in-core algorithms. Note that beyond 1.5GB, the in-core algorithm cannot produce results.

While both algorithms demonstrate a linear increase in memory consumption proportionate to image size, the out-of-core algorithm's slope is 6.7 times lower as that of the conventional in-core approach. Consequently, considering the current workstation, the upper limit of image size amenable to processing ascends from 1.5 GB to 10 GB with w=16. Moreover, we observe that the peak memory consumption decreases as we increase the number of slices.

In terms of execution times, it appears that, as expected, the OOC calculation time is longer than the in-core approach. However, the order of magnitude remains similar. We can also see that processing time increases with the number of tiles, highlighting a trade-off between execution time and memory consumption.

In the preceding experiment, we exclusively utilized only one of the presented calculation patterns. To compare them, we computed the distribution associated with the complete FIB-SEM volume while incrementally increasing the number of tiles. In the Figure 4.12, `Causal` refers to the high-level computation detailed in Section 4.4.2, `Causal GBT` represents the global border tree approach detailed in Section 4.4.1 applied to a causal image partitioning, and `Grid GBT` denotes the same approach but within the context of grid partitioning.

At first glance, the peak memory consumption in Figure 4.12a decreases with the increasing number of slices for all three methodologies. However, `Causal GBT` approach suffers from a sudden increase in memory consumption becoming no



(a) Peak memory consumption

(b) Execution time

(c) Region count

(d) Core operations timings

Figure 4.12: Comparative analysis across different computation patterns on FIB-SEM volume as a function of the number of slices.

longer competitive. This abrupt escalation in memory consumption is due to the fast increase in the GBT size, correlated with the number of voxel at the interface between two tiles. In Figure 4.12c, we can see that the number of regions of `Causal GBT` quickly exceeds the average number of local hierarchy regions. While the number of regions in a local hierarchy decreases as a function of the number of slices, the number of regions in the global border tree increases linearly.

In contrast, the grid partitioning method does not exhibit such phenomenon in the first graph because the slope of the `Grid GBT` size is much less pronounced compared to `Causal`. As expected, the size of the largest merged tree in for `Causal` remains nearly constant regardless the number of tiles. While an explosion in memory consumption may occur for an excessive number of tiles, in the `Grid GBT`, it is not the case for the Causal one, which boasts superior scalability.

However, in Figure 4.12b, we can see that the execution time of GBT-based approaches remains systematically lower than the causal with an average speedup of 2 for `Grid GBT`. Therefore, this algorithm is a good method for reducing com-

putation time as long as the number of tiles remains reasonable, which seems to be beneficial from an application point of view in order to strike a happy medium between memory consumption and computation time.

This disparity in computation time between the two GBT-based and `Causal` approaches can be easily explained by considering the size of the data structures and the number of calls to each operation. In Figure 4.12d we present a comparison of the the execution time of each core operation involved in the distribution computation. The term "BPH" refers to Playing with Kruskal algorithm [81], and IO encompasses both reading and writing operations on disk.

Firstly, when contrasting `Causal GBT` with `Grid GBT`, the only notable difference lies in *join*. The steep slope observed in the `Causal GBT` approach is a consequence of the rapid growth in the size of the GBT, making the merging process with a local border tree increasingly resource intensive.

Secondly, `Grid GBT` outperforms `Causal` in terms of execution time for each operations, except for BPH. It is easy to see that, given the methodology, the number of calls to *insert*, *select* and IO is halved. Regarding *join* and *select*, the performance improvement is also attributed to the fact that in causal partitioning, the number of voxels at the interface grows linearly with the number of tiles, while in grid partitioning, it is proportional to the cubic root of the number of tiles as illustrated in Figure 4.13.

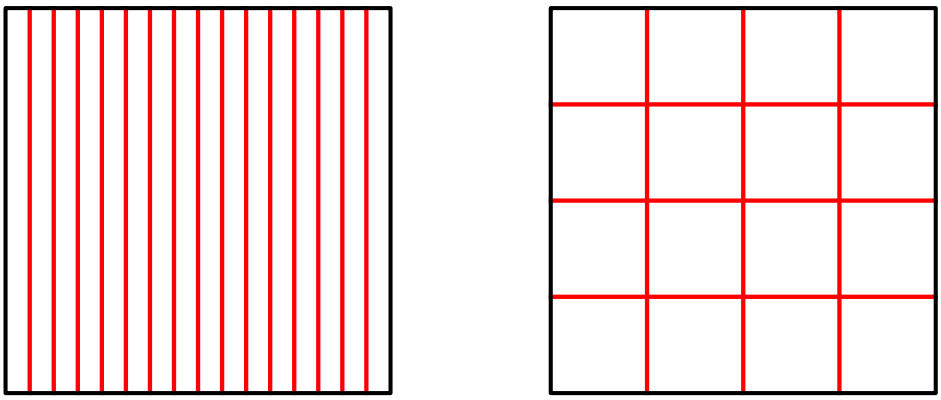


Figure 4.13: Causal and grid partition into 16 tiles. It is necessary to draw 15 frontiers (in red) for the causal partition and only 9 for the grid partition to reach the same number of slices.

4.8 Conclusion

In this chapter, we introduced the concept of *distribution of the binary partition hierarchy*, which consist in a collection of subtrees of the binary partition hierarchy. Each subtree is associated with a partition of the space, and is of size manageable in memory for regular workstation.

In order to build this representation under memory constraint, we gave two high-level calculi with different properties. The first calculus is applicable to any partition of the space, while the second is tailored specifically for causal partitions. Both calculi rely on three fundamental algebraic operations on hierarchies, namely *select*, *join*, and *insert*, which exhibit linear or linearithmic complexities.

We also outlined the properties necessary to ensure that the second calculus satisfies the out-of-core constraints. Our experiments compared these workflows, revealing that although the first calculus lacks certain guarantees, it may still be suitable for applications with a moderate number of slices. In the next chapter we propose efficient algorithms for processing the distribution of a BPH, with the goal of enhancing the out-of-core toolkit by integrating hierarchical analysis tools for (marked) watershed and hierarchical watershed segmentation.

Chapter 5

Out-of-Core Attribute and Watershed Algorithms

Related Publication

- J. Lefèvre, J. Cousty, B. Perret, and H. Phelippeau. Out-of-core attribute algorithms for binary partition hierarchies. *Discrete Geometry and Mathematical Morphology*, pages 298–311, 2024. 10.1007/978-3-031-57793-2_23

5.1 Introduction

In hierarchical image analysis, attributes play a central role in refining segmentations and enabling powerful filtering techniques. Connected filtering methods [99, 19] often rely on these attributes to remove or preserve structures based on meaningful criteria like area, volume, or contrast. Extinction values [43, 120, 106] are commonly used to design extrema oriented filter which are widely used in hierarchical segmentation frameworks.

Notably, the structure that motivated this work highly relies on attributes. Indeed, the watershed computation from the BPH has been given by Najman et al. [81] based on extrema determination and a *watershed node* attribute. We propose another construction from a different attribute dedication to interactive segmentation [55, 56] which will be discussed in Chapter 7. Hierarchical watershed rely even more on attribute computation with persistence and extinctions values [28].

Attribute computation has been extensively studied in various contexts including incremental [107], parallel [125] and distributed [38, 39, 53] approaches. Considering the parallel and distributed settings, the attribute is computed during the construction of the structure. This requires the attribute to be calculated to be chosen a priori, which introduces rigidity into the workflow.

In contrast, our method is designed for out-of-core processing, where it is undesirable to compute attributes during the distribution computation phase. Doing so would significantly increase memory usage, as both the hierarchical structure and the associated attribute values would need to be retained simultaneously. To address this limitation, we propose a two-step approach: first, compute the distribution of the Binary Partition Hierarchy; then, compute the *distribution of the attribute*. The objective is to ensure that filtering each hierarchy in the distribution individually yields the same result as filtering the complete BPH of the entire image.

Note that during the course of this work, a related approach was introduced by Gazagnes and Wilkinson [38], who extended their distributed component tree framework to support *a posteriori* parallel attribute computation. Their method, however, specifically targets component trees in a distributed cluster environment, relying on message passing between compute nodes and the explicit construction of intermediate hierarchies to propagate attributes.

In contrast, our contribution focuses on Binary Partition Hierarchies in an out-of-core processing framework, designed to handle very large datasets that cannot fit entirely in memory. Importantly, our method does not require the explicit construction or storage of intermediate hierarchies, which considerably reduces both memory overhead and I/O cost. Additionally, we go beyond connected filtering by providing algorithms to construct other representations such as the watershed and its hierarchical declination.

In this chapter, we extend our out-of-core BPH framework to support attribute computation in a manner that mirrors traditional in-core processing. Rather than solely distributing the BPH, we propose dedicated algorithms that allow attributes to be computed and propagated across the distribution, ensuring that the same operations used in in-core settings can be applied seamlessly in the out-of-core case.

A central element of this contribution is the Propagate algorithm, which serves as the fundamental mechanism for attribute computation within the out-of-core distribution. This algorithm efficiently propagates information across the hierarchy without requiring global synchronization, enabling the computation of a wide range of attributes such as area, volume, height, and minima. These attributes then form the foundation for more advanced segmentation techniques, including the out-of-core watershed and hierarchical watershed methods.

The remainder of this chapter is organized as follows. We first formalize the notion of an attribute distribution and introduce the Propagate algorithm tailored for causal partition, detailing its role in out-of-core computation. We then illustrate the genericity of this methodology by describing specific attribute computation strategies illustrating how they can be implemented in an out-of-core

setting. Finally, we demonstrate how this propagation model enables efficient out-of-core watershed segmentation, in both its standard and hierarchical forms.

5.2 Distribution of Attributes

In this section, we introduce the concept of the *distribution of an attribute*. To begin, let us recall the formal definition of an attribute in the context of a hierarchy:

Definition 5.2.1 (Attribute)**.** An *attribute on* $\mathcal{H}$ is a mapping A associating a scalar value $A(R)$ to every region R of $\mathcal{H}$.

We now adapt this definition to the distributed case. Due to regions spanning multiple slices, the same region may appear in multiple local hierarchies. This is illustrated in Figure 5.1c, where a distribution $(\mathcal{H}_0, \mathcal{H}_1, \mathcal{H}_2)$ is derived from a hierarchy H on a partition $(\{d, a\}, \{e, b\}, \{c, f\})$. In this example, region k, the root of the original hierarchy $\mathcal{H}$, appears in all three local hierarchies. This behavior is typical for most non-leaf regions, with the exception of regions such as g in $\mathcal{H}_2$, being parent of two leaf-regions in $gr(\mathcal{H}_2)$.

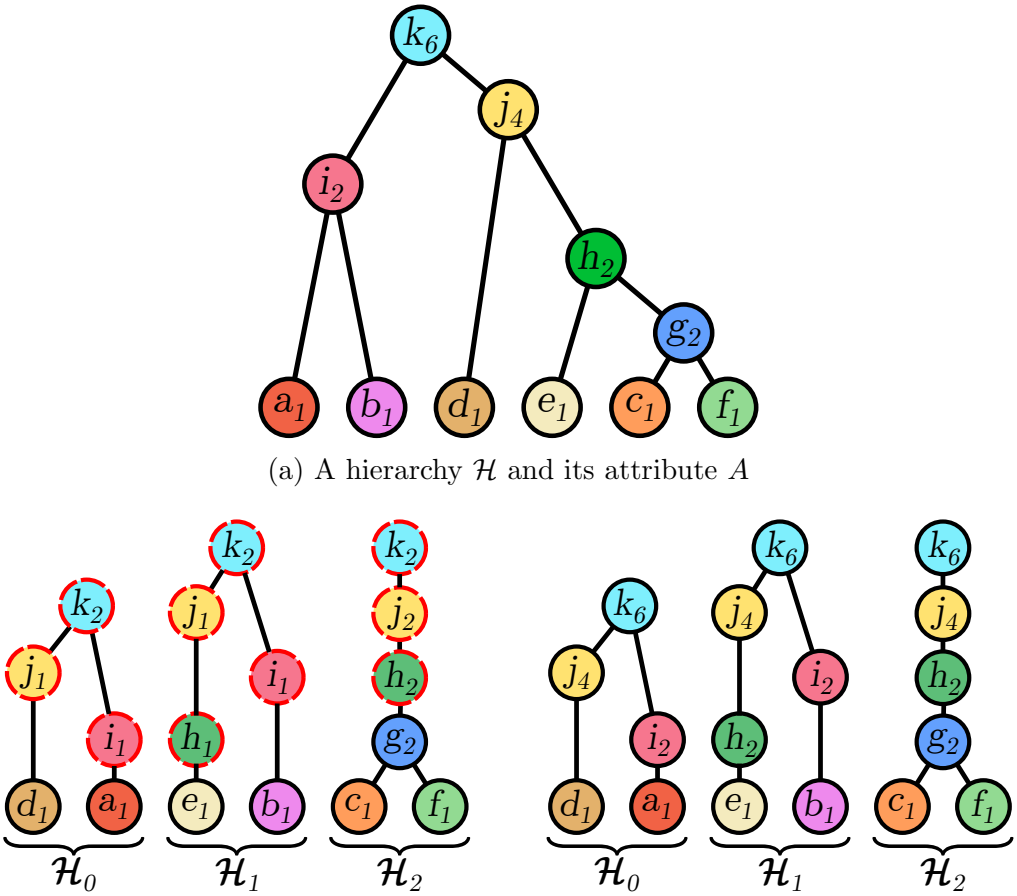
To formalize attribute behavior in such a setting, we extend the notion of an attribute to its distribution over a BPH distribution as follows:

Definition 5.2.2 (Distribution of an attribute)**.** Let A be and attribute, let $\mathcal{H}$ be a hierarchy and let $\delta_{\mathcal{H}} = (\mathcal{B}_0, \cdots, \mathcal{B}_k)$ be an associated distribution. The *distribution of* A *over* $\delta_{\mathcal{H}}$ is the series $\delta_A = (A_{\mathcal{B}_0}, \cdots, A_{\mathcal{B}_k})$ of attributes on $\delta_{\mathcal{H}}$ such that for any i in $\{0, \ldots, k\}$ and any R in $\mathcal{B}_i$, we have $A(R) = A_{\mathcal{B}_i}(R)$.

This definition ensures that the attribute values remain consistent across all occurrences of a region within the distributed hierarchy. For instance, in Figure 5.1c, the attribute values (noted as subscripts) are correctly distributed: each instance of a region shares the same attribute value, matching that of the original hierarchy. In contrast, Figure 5.1b violates this consistency, as regions highlighted with red dashed borders display attribute values that differ from their counterparts in the original hierarchy, thereby failing to form a valid attribute distribution.

The central problem addressed in this chapter can now be stated as follows:

Problem. *Given the distribution* $\delta_{\mathcal{H}}$ *of the hierarchy* $\mathcal{H}$*, compute the distribution* δ_A *of an attribute* A *without explicitly computing* $\mathcal{H}$ *nor* A *while maintaining the out-of-core constraint, that is having a limited amount of memory at each computation steps.*



(a) A hierarchy $\mathcal{H}$ and its attribute A

(b) Distribution of $\mathcal{H}$ but not of A

(c) Distribution of $\mathcal{H}$ and A

Figure 5.1: Illustration of the distribution of the area attribute A. Each node is identified by a letter and its area is given as subscript. Nodes with red dashed borders in **(b)** are regions where the attribute differs from the original hierarchy.

5.3 Propagate Algorithm: a Fundamental Brick for Out-of-Core Attribute Computation

In this section, we introduce the PROPAGATE algorithm, a fundamental step in all subsequent attribute computations. Since a region may span multiple slices, computing its attribute using only the information available in a single local hierarchy of the distribution is generally not feasible. Let's consider the example of the area attribute. We computed the area on the hierarchy $\mathcal{H}$ on Figure 5.1a using the established algorithm. Then on Figure 5.1b, by applying the same algorithm independently on each local hierarchy we did not obtain the distribution of this attribute. These inconsistencies arise because the area of a region is based on the

area of its children, which may reside in different local hierarchies. For instance, in Figure 5.1b, the area of region i is computed as the sum of the areas of regions a and b, which are located in different local hierarchies. Consequently, the area of region i computed in $\mathcal{H}_0$ is only based on the area of $a \in gr(\mathcal{H}_0)$, leading to an incorrect distribution of the area attribute.

To address this issue, the out-of-core attribute computation follows a two-step process: an initial local computation phase similar to Figure 5.1b, followed by a merging phase using the PROPAGATE algorithm.

More precisely, the computation proceeds as follows:

1. **Local computation**: Compute a *partial attribute* value independently on each tree in the distribution.

2. **Causal propagation**: Propagate partial attribute values from neighboring trees in a causal direction, *i.e.*, from slices with lower indices to those with higher indices. By the end of this step, the attribute values for the last tree in the distribution are fully determined.

3. **Anti-causal backpropagation**: Propagate the corrected attribute values back in the opposite direction, *i.e.*, from slices with higher indices to those with lower indices. At the end of this step, all attribute values across the entire distribution are correctly computed.

In this approach, each local hierarchy is visited twice: once during the causal pass and once during the anti-causal pass. While ensuring that, at any given moment, no more than two neighboring hierarchies need to be held in memory simultaneously. This design maintains efficiency and minimizes memory usage, making it well-suited for out-of-core processing.

We now provide a detailed presentation of the PROPAGATE algorithm, introduced in Algorithm 8, which serves as a core component of our out-of-core attribute computation strategy. This algorithm transforms a local version of an attribute, initially computed independently on each slice, into a consistent global version across the entire hierarchy distribution.

To achieve this, the algorithm processes regions that are duplicated across neighboring hierarchies corresponding to successive slices. Attribute values associated with these shared regions are propagated in two stages. First, a *causal pass* is performed from lower to higher slice indices (lines 1 to 3 in Algorithm 8). Then, an *anti-causal pass* propagates the updated attribute values in the reverse direction, from higher to lower slice indices (lines 5 to 6).

During the causal phase, attribute values from duplicated regions in consecutive slices S_i and S_{i+1} are combined using a binary merging operator, denoted by $\oplus$. This operator is passed as a parameter to the algorithm and can be chosen

Algorithm 8: PROPAGATE

Params: The distribution $(\mathcal{B}_0, \ldots, \mathcal{B}_k)$ of a BPH; a series $(\texttt{attr}_{\mathcal{B}_0}, \ldots, \texttt{attr}_{\mathcal{B}_k})$ of attributes; and a binary operator $\oplus$

Result: A new series $(\texttt{attr}^{\downarrow}_{\mathcal{B}_0}, \ldots, \texttt{attr}^{\downarrow}_{\mathcal{B}_k})$ of attributes where values of regions replicated on consecutive slices have been propagated.

1 $\texttt{attr}^{\uparrow}_{\mathcal{B}_0} := \texttt{attr}_{\mathcal{B}_0}$
2 **foreach** i from 1 to k **do**
3 $\quad \texttt{attr}^{\uparrow}_{\mathcal{B}_i} := \text{MERGE}(\mathcal{B}_{i-1}, \mathcal{B}_i, \texttt{attr}^{\uparrow}_{\mathcal{B}_{i-1}}, \texttt{attr}_{\mathcal{B}_i}, \oplus)$
4 $\texttt{attr}^{\downarrow}_{\mathcal{B}_k} := \texttt{attr}^{\uparrow}_{\mathcal{B}_k}$
5 **foreach** i from $k-1$ to 0 **do**
6 $\quad \texttt{attr}^{\downarrow}_{\mathcal{B}_i} := \text{MERGE}(\mathcal{B}_{i+1}, \mathcal{B}_i, \texttt{attr}^{\downarrow}_{\mathcal{B}_{i+1}}, \texttt{attr}^{\uparrow}_{\mathcal{B}_i}, \triangleleft)$
7 **return** $\left(\texttt{attr}^{\downarrow}_{\mathcal{B}_0}, \ldots, \texttt{attr}^{\downarrow}_{\mathcal{B}_k}\right)$

according to the nature of the attribute. In the following section we will, for instance, consider the sum, maximum, or minimum. The result of this merging phase ensures that the final tree in the distribution holds fully resolved attribute values.

In the anti-causal pass, the attribute value associated with the region in the slice of higher index is copied as the attribute value of the same replicated regions in the slices of lower index. Thereby ensuring consistency throughout the entire hierarchy distribution.

To identify regions that are duplicated across two consecutive hierarchies, $\mathcal{X}$ and $\mathcal{Y}$, the PROPAGATE algorithm invokes an auxiliary function called MERGE, which is detailed in Algorithm 9. This function performs a simultaneous traversal of both hierarchies in increasing order. A pair of nodes $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ are recognized as occurrences of the same region when the condition $\mathcal{X}.\texttt{map}[x] = \mathcal{Y}.\texttt{map}[y]$ holds, as shown on line 3 of Algorithm 9. In this case, their corresponding attribute values, $\texttt{attr}_{\mathcal{X}}[x]$ and $\texttt{attr}_{\mathcal{Y}}[y]$, are merged.

The result of this merging operation is assigned to node y in $\mathcal{Y}$, following the rule:

$$\texttt{attr}'_{\mathcal{Y}}[y] := \texttt{attr}_{\mathcal{X}}[x] \oplus \texttt{attr}_{\mathcal{Y}}[y]$$

where $\oplus$ denotes a generic binary merging operator.

To enable the back-propagation of attribute values from $\mathcal{Y}$ to $\mathcal{X}$, we use the binary operator $\triangleleft$, defined as $a \triangleleft b = a$. When this operator, referred to as the *assign-first* operator, is used as the merging function (that is, when $\oplus = \triangleleft$), the merge operation simplifies to:

$$\texttt{attr}_{\mathcal{X}}(x) \triangleleft \texttt{attr}_{\mathcal{Y}}(y) = \texttt{attr}_{\mathcal{X}}(x)$$

Algorithm 9: MERGE

Params: Two hierarchies $\mathcal{X}$ and $\mathcal{Y}$; two attributes $\texttt{attr}_\mathcal{X}$ and $\texttt{attr}_\mathcal{Y}$ associated with these hierarchies ; and a binary operator $\oplus$

Result: A new attribute $\texttt{attr}'_\mathcal{Y}$, update of $\texttt{attr}_\mathcal{Y}$

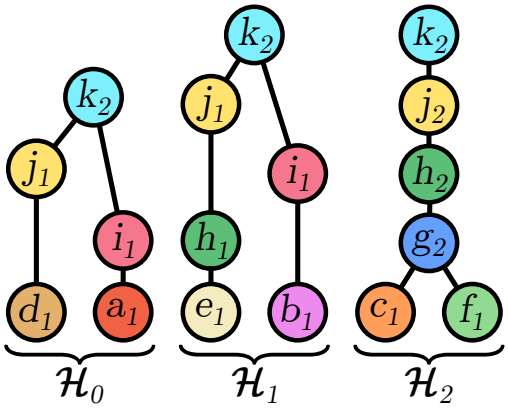
```
1  x := 0; y := 0                    // x iterates over X and y over Y
2  while x < |X| or y < |Y| do
3      if x < |X| and y < |Y| and X.map[x] = Y.map[y] then
4          attr'_Y[y] := attr_X[x] ⊕ attr_Y[y]
5          x := x + 1;  y := y + 1
6      else if X.map[x] ≺ Y.map[y] then  x := x + 1
7      else
8          attr'_Y[y] := attr_Y[y]
9          y := y + 1
10 return attr'_Y
```

This behavior ensures that the attribute value from $\mathcal{Y}$ is unconditionally propagated to its corresponding node in $\mathcal{X}$.

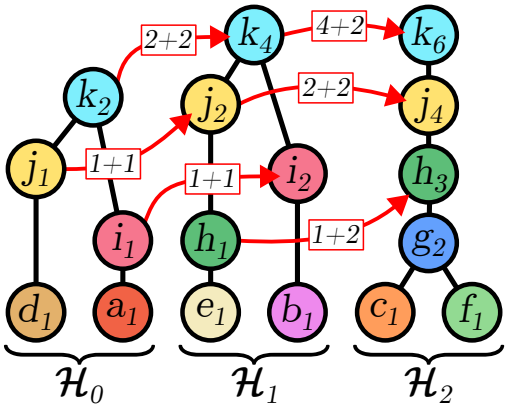
Figure 5.2 illustrates the use of PROPAGATE Algorithm. More precisely, in Figure 5.2a, an initial attribute value is mapped to each node (subscript) of the local hierarchies (*i.e.*, , $(attr_{\mathcal{H}_0}, attr_{\mathcal{H}_1}, attr_{\mathcal{H}_2})$). The values of $(attr^{\uparrow}_{\mathcal{H}_0}, attr^{\uparrow}_{\mathcal{H}_1}, attr^{\uparrow}_{\mathcal{H}_2})$ obtained after the causal pass of PROPAGATE are then shown in Figure 5.2b. In particular, red arrows indicate the mapping between replicated regions detected by MERGE as well as the computation which is made to obtain the value of the updated attribute (here $\oplus = +$ is given to MERGE). Finally, Figure 5.2c shows the result of $(attr^{\downarrow}_{\mathcal{H}_0}, attr^{\downarrow}_{\mathcal{H}_1}, attr^{\downarrow}_{\mathcal{H}_2})$ obtained after the anti-causal pass.

It can be observed that every node of each tree is browsed once during the execution of MERGE Algorithm. Thus, the overall time-complexity of MERGE Algorithm is linear with respect to the number of regions of $\mathcal{X}$ and $\mathcal{Y}$. Furthermore, given a causal partition with $k+1$ slices, it can be seen that PROPAGATE performs exactly $2 \times k$ calls to MERGE and that the overall time complexity of PROPAGATE Algorithm is $O(k + N)$ where N is the sum of the numbers of regions of the local hierarchies $\mathcal{B}_0, \ldots, \mathcal{B}_k$.

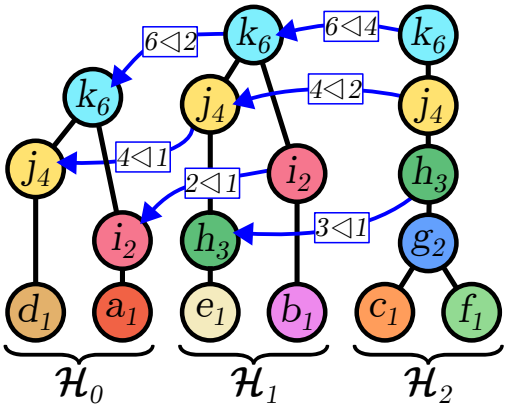
In the next section, we will demonstrate how the PROPAGATE algorithm can be utilized to compute classical attributes in a out-of-core setting by appropriately selecting the merging operator $\oplus$ during the propagation process.



(a) Attribute initialization.



(b) Causal pass.



(c) Anti-causal pass.

Figure 5.2: Illustration of the PROPAGATE algorithm. Each node is identified by a letter, and its attribute value (area) is indicated as a subscript. Red arrows show the replicated regions detected by MERGE and point to the region whose attribute is updated. Operand order is given by Algorithm 8.

5.4 Out-of-Core Attributes Algorithms

In this section, following the general scheme proposed in Section 5.3, we present out-of-core algorithms to compute common attributes in hierarchical analysis. To ensure compliance with memory constraints, each attribute is formulated to minimize the use of auxiliary data structures. For instance, when computing the area attribute, a common approach involves maintaining a structure that stores the child nodes of each non-leaf node. However, this data structure requires $O(n)$ space complexity, where n represents the number of elements in the underlying data, making it impractical for large-scale processing.

For each attribute, we will first describe the in-core algorithm and then detail how to adapt it to the out-of-core setting.

5.4.1 Area

The area attribute is one of the most fundamental attributes in hierarchical analysis, as it simply assigns to each region in a hierarchy the number of pixels it contains. Beyond its direct use in analyzing region sizes, area also serves as a foundation for computing other important attributes, such as volume, which integrates intensity values over a region, mean grayscale, which normalizes intensity by area, and circularity, which relates area to perimeter to characterize shape compactness.

In the tree-based representation of a hierarchy, the area of a node n can be obtained recursively by setting $\texttt{area}(n) = 1$ for every leaf n. For a non-leaf node, the area is obtained by summing the areas of its children:

$$\texttt{area}(n) = \sum_{c \in \texttt{children}(n)} \texttt{area}(c), \tag{5.1}$$

where the set of children of node n is defined as:

$$\texttt{children}(n) = \{c \mid n = \texttt{parent}(c)\}\,. \tag{5.2}$$

The recursive computation of the area attribute consist in the propagation of values associated with the leaves to the root of the hierarchy. Such bottom-up propagation can be achieved by the ACCUMULATE algorithm, which is detailed in Algorithm 10. We expressed this recursion in a generic way dropping the operator sum used in Equation (5.1) and replacing it by a generic operator $\oplus$. We can for example compute bounding boxes by associating coordinates to each leaves and accumulating them with *min* and *max* operators.

Algorithm 10: ACCUMULATE

Params: A hierarchy $\mathcal{B}$, a map L from the leaves of $\mathcal{B}$ to a scalar, and a binary operator $\oplus$.
Result: An attribute A.

```
foreach leaf R of B do
    A[R] = L(R)
foreach non-root region R of B in topological order do
    A[H.par[R]] = A[H.par[R]] ⊕ A[R]
return A
```

Going back to our case, this algorithm can be used to compute the distribution of the area attribute with a proper initialization of leaf values. In order to count

the number of pixels in each region of a distribution, we initialize the area of the leaves with 1 and then call PROPAGATE with the + operator.

Algorithm 11: AREA DISTRIBUTION

Params: A distribution $\delta_{\mathcal{H}} = \{\mathcal{B}_0, \ldots, \mathcal{B}_k\}$ of hierarchies.
Result: Distribution of area attribute.

```
foreach i from 0 to k do
    Initialize an array L to 1 for every leaf region of B_i
    area_{B_i} = ACCUMULATE(B_i, area_{B_i}, +)
return PROPAGATE((area_{B_0}, ..., area_{B_k}), δ_H, +)
```

5.4.2 Volume

In the tree-based representation of a hierarchy, the volume $V(n)$ of a node n is computed recursively. For a leaf node n, we set $V(n) = 0$. For a non-leaf node, the volume is defined as the sum of the volume contributions from its own region and its descendants:

$$V(n) = A(n) \times |\texttt{altitude}(\texttt{parent}(n)) - \texttt{altitude}(n)| + \sum_{c \in \texttt{children}(n)} V(c) \quad (5.3)$$

where $A(n)$ denotes the area of node n.

In the distributed setting, the volume is first initialized locally using this recursive formulation. However, as with the area attribute, at this stage, each region lacks the volume contributions from included regions that do not belong to the same local hierarchy. To ensure a globally consistent volume computation, the partial volume values are corrected by applying PROPAGATE with the + operator.

Algorithm 12: VOLUME DISTRIBUTION

Params: A distribution $\delta_{\mathcal{H}} = \{\mathcal{B}_0, \ldots, \mathcal{B}_k\}$ of hierarchies.
Result: Distribution of volume attribute.

```
foreach i from 0 to k do
    Locally initialize area_{B_i} i.e., , lines 1-3 of Area
    foreach non-root region R of B_i in topological order do
        vol_{B_i}[R] += area_{B_i}[R] × (alt[B_i.par[R]] − alt[R])
        vol_{B_i}[B_i.par[R]] += vol_{B_i}[R]
return PROPAGATE((vol_{B_0}, ..., vol_{B_k}), δ_H, +)
```

5.4.3 Height

The height of a region R in a BPH $\mathcal{H}$ is defined as the difference between the altitude of R and the altitude of the lowest region included within R. In the tree-based representation of a hierarchy, the height of a node n, denoted as $H(n)$, is computed recursively as follow. For a leaf node, the height is initialized as $H(n) = 0$. For a non-leaf node, the height is given by

$$H(n) = \texttt{altitude}(\texttt{parent}(n)) - \texttt{altitude}(n) + \max_{c \in \texttt{children(n)}} H(c) \tag{5.4}$$

In a distributed setting, after initializing the height of each local hierarchy using the above recursion, the child node contributing to the maximum height or a region in the global hierarchy may reside in a different local hierarchy. To address this, a global search is required to identify and integrate the missing height contributions. This is achieved by applying PROPAGATE with the supremum operator, ensuring that the maximum height is correctly propagated throughout the hierarchy.

Algorithm 13: HEIGHT DISTRIBUTION

Params: A distribution $\delta_{\mathcal{H}} = \{\mathcal{B}_0, \ldots, \mathcal{B}_k\}$ of hierarchies.
Result: Distribution of height attribute.

```
foreach i from 0 to k do
    Initialize heig_{B_i} to 0 for every node
    foreach non-root region R of B_i in topological order do
        heig_{B_i}[B_i.par[R]] :=
          max(heig_{B_i}[B_i.par[R]], alt[B_i.par[B_i.par[R]]] −
          alt_{B_i}[B_i.par[R]] + heig_{B_i}[R])
return PROPAGATE((heig_{B_0}, ..., heig_{B_k}), δ_H, ∨)
```

Note that this method can also be used to compute the topological height, *i.e.*, the maximum length of a paths from a node to a descendant leaf node.

5.4.4 Minima

A regional minimum of an edge-weighted graph is a set of vertices connected by edges whose adjacent edges are all of strictly higher weight. In order to identify such regional minima with the tree-based representation of a BPH, we recall the criteria of Definition 3.5.6:

1. the altitude of the parent of n is greater than the one of n; and
2. every node included in the sub-tree rooted in n is either a leaf or a non-minimum.

We can formalize and test these two criteria in our framework. Firstly, Equation (5.5) defines the first criterion. In order to compute this criterion for a region R, we only need to know the altitude of the parent of R. This information is available locally by construction. Indeed, every parent region of a region R in a hierarchy $\mathcal{H}$ is included in each local hierarchy of $\delta_{\mathcal{H}}$ containing R. Thus, we can compute the criterion locally.

$$L_{\mathcal{B}_i}(R) = \begin{cases} \text{true}, & \text{if } \texttt{altitude}(R) < \texttt{altitude}(\texttt{parent}(R)) \\ \text{false}, & \text{otherwise or if } R \text{ is a leaf} \end{cases} \tag{5.5}$$

$$E^L_{\mathcal{B}_i}(R) = \begin{cases} \text{false}, & \text{if } R \text{ is a leaf} \\ \bigwedge_{c \in \texttt{children}(R)} \neg L_{\mathcal{B}_i}(c), & \text{otherwise} \end{cases} \tag{5.6}$$

Algorithm 14: MINIMA DISTRIBUTION

Params: A distribution $\delta_{\mathcal{H}} = \{\mathcal{B}_0, \ldots, \mathcal{B}_k\}$ of hierarchies.
Result: Distribution of minima attribute.

foreach i from 0 to k **do**
  foreach region R of $\mathcal{B}_i$ in topological order **do**
    $\texttt{includesMin}[R] := false$; $\texttt{mini}_{\mathcal{B}_i}[R] := false$
  foreach non-leaf region R of $\mathcal{B}_i$ in topological order **do**
    if $\texttt{alt}[R] < \texttt{alt}[\mathcal{B}_i.\texttt{par}[R]]$ and $\texttt{includesMin}[R] = false$ **then**
      $\texttt{mini}_{\mathcal{B}_i}[R] := true$; $\texttt{includesMin}[R] := true$
    $\texttt{includesMin}[\mathcal{B}_i.\texttt{par}[R]] := \texttt{includesMin}[R]$
return PROPAGATE($(\texttt{mini}_{\mathcal{B}_0}, \ldots, \texttt{mini}_{\mathcal{B}_k})$, $\delta_{\mathcal{H}}$, $\wedge$)

Secondly, Equation (5.6) defines the second criterion. This last criterion depends on the children of R, so we need to make sure it is satisfied for all children in the distribution. Then its computation cannot be performed locally because as a regional minima children of a region R can belong another local hierarchy, introducing a false positive. We can verify this criterion by calling PROPAGATE with the infimum operator to compute a *global* criterion. Then, an occurrence of a region R in the distribution is a minimum if $M_{\mathcal{B}_i}(R) = L_{\mathcal{B}_i}(R) \wedge E^G_{\mathcal{B}_i}(R)$.

In this algorithm, criterion E^L is emulated with an array `includesMin` initialized at *false* for every region and which is set to *true* as soon as a region of the sub-tree rooted in the current node contains a node which verifies L (in other words contains a minimum). The attribute `includesMin` is spread from child to parent at each iteration and conditions minimum discrimination.

5.5 Out-of-Core Watershed

In this section, we introduce an out-of-core watershed algorithm that builds upon the previously described out-of-core attribute computation techniques. Our approach leverages the relationship between the BPH and the watershed-cut, as discussed in Section 3.5, to efficiently compute watershed segmentation in a memory-constrained setting.

We first present the marker-based watershed cut, which serves as the foundation of our method. We then extend this approach to the minima-based watershed, which requires an additional preprocessing step.

5.5.1 Marker-Based

In-Core Computation

In a first time, we will describe in detail the computation of the watershed-cut rooted in markers from the BPH with a horizontal cut. Such methodology is illustrated on Figure 5.3.

Let $G = (V, E)$ be an edge-weighted graph with two markers placed on vertices a and b. In the BPH $\mathcal{B}$ built on G, we can identify the node of altitude 2 as a watershed node, *i.e.*, the node corresponding to the edge of the MST to remove in order to obtain the watershed cut rooted in the vertices a and b. Indeed, this is the smallest region containing two distinct markers. We are able to discriminate this node with a boolean attribute W defined as follows:

$$W(R) = \begin{cases} 1, & \text{if } R \text{ is a watershed node} \\ 0, & \text{otherwise} \end{cases} \tag{5.7}$$

This attribute is computed following a procedure similar to the one described by Najman et al. [81]. We give a version of this algorithm that does not necessitates children computation in Algorithm 15. Any region that contain more markers than its children is a watershed node. This criterion is computed at Line 5 as C_1 on the basis of an array `cnt`. This array correspond to the attribute C representing the number of distinct markers contained within each region of the hierarchy.

To compute the attribute C we initialize it for each leaf region. If the corresponding vertex is marked then C is set to 1, otherwise it is set to 0. The marker counts are then accumulated from the leaves to the root using the ACCUMULATE procedure, described in Algorithm 10, with the + operator.

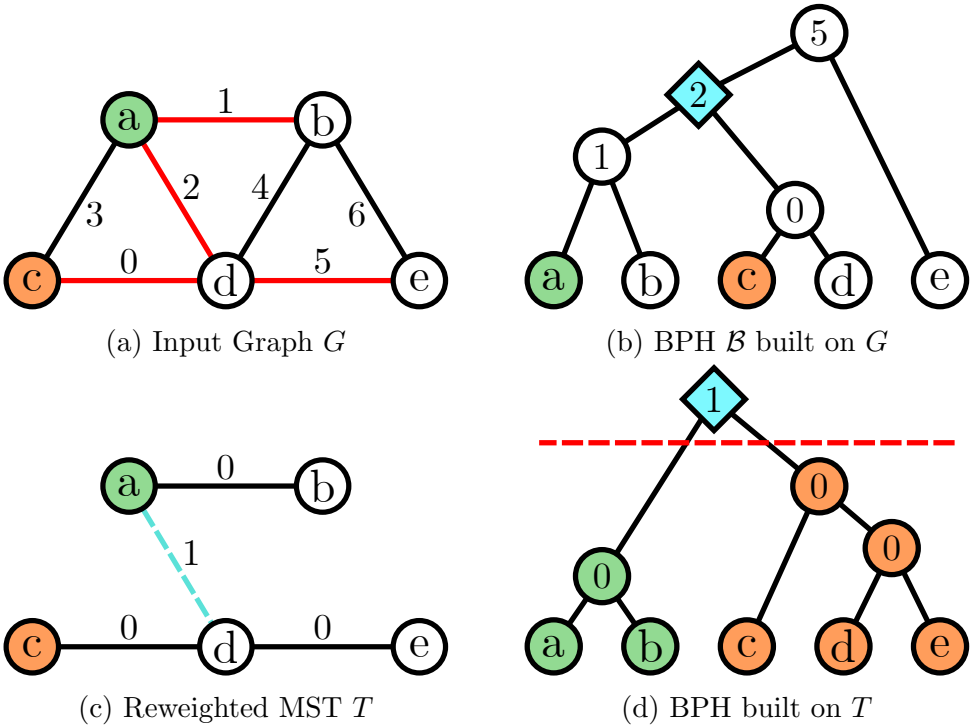
The second criterion is computed at Line 6 as C_2 allows us to compute W by browsing regions in topological order. The strategy consist in putting aside non watershed nodes. Thus, in addition to C_1 we have to get rid of regions that does not contain any marker.

Another aspect that differs from the analogous algorithm proposed by Najman et al. [81] is that C is not determined at the same time as W. This choice has been made in order to comply with the out-of-core workflow that will be presented in the next section.

This can be achieved given the map μ from non-leaf regions of the BPH $\mathcal{B}$ to their building edges in the underlying graph $G = (V, E)$. From this map, we can derive its minimum spanning tree $T = (V, E_{\text{mst}})$ using Property 3.4.1. The watershed-cut partition of G is then obtained by considering the graph $F = (V, E_{\text{mst}} \setminus S)$, where:

$$S = \{\mu(R) \mid R \in \mathcal{B} \wedge W(R)\} \tag{5.8}$$

As discussed in Section 3.5 for the classical in-core case, once the watershed nodes are identified, we can remove their building edges from the MST to obtain the watershed-cut.



(a) Input Graph G

(b) BPH $\mathcal{B}$ built on G

(c) Reweighted MST T

(d) BPH built on T

Figure 5.3: Illustration of the watershed cut using BPH and horizontal cut. The graph G is shown in **(a)** with two markers on a and c, the BPH $\mathcal{B}$ built on G is shown in **(b)** and the watershed node is outlined in cyan, the reweighted MST T is shown in **(c)** with the corresponding watershed edge represented in dashed cyan, and the BPH built on T is shown in **(d)**.

Algorithm 15: WATERSHED NODES

Params: A hierarchy $\mathcal{B}$, an array `cnt` of the minimum/marker count of each region of $\mathcal{B}$.

Result: A binary array `ws` indicating whether a node is a watershed node or not.

```
Initialize an array ws to True for every region of B
foreach leaf R of B do
    ws[R] := False;
foreach non-root region R of B in topological order do
    C1 := cnt[n] < cnt[B.par[R]]
    C2 := cnt[n] > 0
    ws[B.par[R]] := ws[B.par[R]] ∧ C1 ∧ C2
return ws
```

Such procedure is illustrated in Figure 5.3c, where the watershed edge $\{a, d\} \in S$ is represented in dashed cyan. This edge is indeed the building edge of the watershed node previously identified. Then, the labeling can be obtained by running any connected component labeling algorithm.

While being efficient, this approach is not satisfying from an out-of-core perspective. Indeed, to obtain the watershed-cut labeling for each local hierarchy, we need to run a connected component labeling algorithm on the entire graph F. In the following we will adopt a method that eliminates the need for such an algorithm ensuring that all necessary information is available locally.

To compute the watershed-cut partition, we reweight the minimum spanning tree T of G by assigning a weight of 1 to the watershed edge (a, d) and setting all other edge weights to 0. To do so, we use the MST computed alongside with the BPH during PWK algorithm [81]. The reweighing procedure is described in Algorithm 16. We leverage the fact that both edges of the MST and regions of the BPH are sorted with respect to a total order. We then construct the BPH from this reweighted MST and apply a horizontal cut at level 1 to obtain the final partition of G.

Algorithm 16: Reweight MST

Params: A hierarchy $\mathcal{B}$, the watershed attribute `ws` of $\mathcal{B}$.
Result: A weighted minimum spanning tree T.

$T' = (V, E, w) = \mathcal{B}$.`MST`
$b := 0$; $t := 0$
Initialize an array w_{ws} of size $|E|$ to 1
while $b < |\mathcal{B}|$ **or** $t < |E|$ **do**
  if $b < |\mathcal{B}|$ **and** $t < |E|$ **and** $\mathcal{B}$.`map`$[b] = T'$.`map`$[t]$ **then**
    w_{ws}`[`t`]` := `ws`[b]
    $b := b + 1$; $t := t + 1$
  else if $\mathcal{B}$.`map`$[b] \prec T'$.`map`$[t]$ **then** $b := b + 1$
  else $t := t + 1$
return $T = (V, E, w_{\text{ws}})$

Let $\mathcal{H}$ be a hierarchy. The horizontal cut of $\mathcal{H}$ at altitude k corresponds to the partition $\mathbf{P}_\lambda$ for the highest λ such that each regions $R \in \mathbf{P}_\lambda$ has an altitude lower than k. Algorithm 17 describes the horizontal cut algorithm which associates a unique label to each region of the partition $\mathbf{P}_\lambda$.

Algorithm 17: Horizontal Cut

Params: A hierarchy $\mathcal{B}$ with n nodes and l leaves, and a threshold θ.
Result: A label array L for the leaves after horizontal cut at altitude θ.

$L := [0, \ldots, n$ - $1]$
foreach non-leaf node i of $\mathcal{B}$ in inverse topological order **do**
  if $\mathcal{B}$.`alt`$[i] \leq \theta$ **then**
    L`[`i`]` := L`[`$\mathcal{B}$.`par`$[i]$`]`
foreach leaf i of $\mathcal{B}$ **do**
  if $\mathcal{B}$.`alt`$[\mathcal{B}$.`par`$[i]] \leq \theta$ **then**
    L`[`i`]` := L`[`$\mathcal{B}$.`par`$[i]$`]`
return L`[0 :`l`]`

The BPH of the reweighted MST and the labeling induced by the horizontal cut at altitude 1 are illustrated on Figure 5.3d. In order to obtain a labeling from the cut, we associate with all nodes within the same subtree below the horizontal cut the same label. Resulting in the partition $\{\{a, b\}, \{c, d, e\}\}$.

This final step is crucial, as illustrated by the BPH in Figure 5.3b. If we were to perform a horizontal cut at altitude 2 to separate below the watershed node, we would obtain the partition $\{\{a, b\}, \{c, d\}, \{e\}\}$. The vertex e will belong to a third singleton region that cannot be merged to any basin solely based on this hierarchy. In this specific case, it would work if and only if the watershed node is the root.

In the following section we will adapt this methodology to the out-of-core setting only reusing tools we previously introduced.

Out-of-Core Computation

We can outline three main steps to compute the watershed-cut in an out-of-core setting illustrated on Algorithm 18:

1. Compute the distribution of the watershed attribute W across the hierarchy;
2. Build the distribution of the reweighted MST; and
3. Assign labels to the vertices by performing a horizontal cut at level 1 for each local hierarchy.

Algorithm 18: Out-of-core marker-based watershed

Params: A distribution $\delta_{\mathcal{H}} = \{\mathcal{B}_0, \dots, \mathcal{B}_k\}$ of hierarchies, and a series of markers $(M_0, \dots, M_k)$ for each hierarchy.
Result: A series of watershed partition $WS_0, \dots, WS_k$ rooted in the markers $M_0, \dots, M_k$.

```
// Compute the count of distinct markers
```
1 **foreach** i from 0 to k **do**
2 $\quad \texttt{cnt}^{\uparrow}_{\mathcal{B}_i} := \textsc{Accumulate}(\mathcal{B}_i, M_i, +)$ ▷ Algorithm 10
3 $(\texttt{cnt}_{\mathcal{B}_0}, \dots, \texttt{cnt}_{\mathcal{B}_k}) := \textsc{Propagate}(\left(\texttt{cnt}^{\uparrow}_{\mathcal{B}_0}, \dots, \texttt{cnt}^{\uparrow}_{\mathcal{B}_k}\right), \delta_{\mathcal{H}}, +)$
```
// Compute the watershed nodes
```
4 **foreach** i from 0 to k **do**
5 $\quad \texttt{ws}_{\mathcal{B}_i} := \textsc{WatershedNodes}(\mathcal{B}_i, \texttt{cnt}_{\mathcal{B}_i})$ ▷ Algorithm 15
```
// Reweight the MST of each hierarchy
```
6 **foreach** i from 0 to k **do**
7 $\quad G_i := \textsc{ReweightMST}(\mathcal{B}_i, \texttt{ws}_{\mathcal{B}_i})$ ▷ Algorithm 16
```
// Compute the distribution on the reweighted MSTs
```
8 $(\mathcal{B}^W_0, \dots, \mathcal{B}^W_k) := \textsc{Distribution}((G_0, \dots, G_k))$ ▷ Algorithm 3
```
// Perform horizontal cuts at altitude 1
```
9 **foreach** i from 0 to k **do**
10 $\quad \mathbf{W}_i := \textsc{HorizontalCut}(\mathcal{B}^W_i, 1)$ ▷ Algorithm 17
11 **return** $(\mathbf{W}_0, \dots, \mathbf{W}_k)$

1. Algorithm 18 Lines 1-5. In the previous section, we show ho to compute the attribute W based on the attribute C that assigns to each region the number of distinct marker it contains. This attribute is computed using the Accumulate

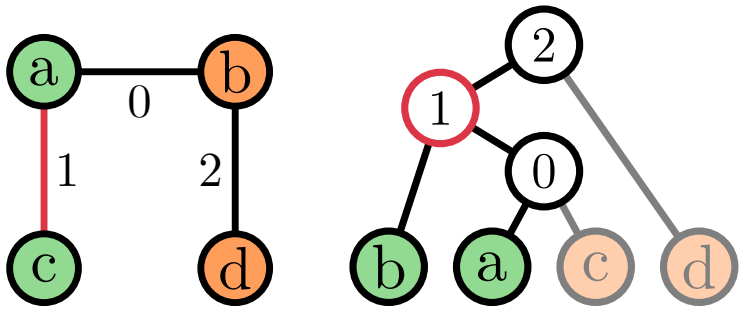
algorithm with the + operator, similarly to the area attribute. The only difference is the initialization: leaves are set to 1 if they contain a marker and 0 otherwise. Analogous to the area attribute, the distribution of C can be computed using the PROPAGATE algorithm with the + operator.

At this point, one may question whether invoking PROPAGATE is necessary for computing the distribution of W. Recall the watershed node criterion: a node is a watershed node if its corresponding region contains more markers than any of its children. In a BPH, each non-leaf node has exactly two children. If the number of markers in a node exceeds that of one of its children, then the number of markers in the other child must be strictly less than that of the parent. Consequently, we can compute W without calling PROPAGATE by simply checking whether a region contains more markers than at least one of its children. This means W can be computed locally for each element of the distribution.

2. Algorithm 18 Lines 6-8. This step involves building the reweighted MST associated with each local hierarchy. Let $\mathcal{B}$ be the BPH built on the graph G, let $X \subset G$ and let $\mathcal{B}_X = \text{select}\,(V(X), \mathcal{B})$. By construction, there exists an injection μ^{-1} that map each edges of the MST of G in X to a corresponding non-leaf nodes in $\mathcal{B}_X$.

This correspondence is guaranteed by the *select* property, which ensures that for any non-leaf node in $\mathcal{B}$ representing a region containing $x \in V(X)$, there exists an occurrence of this region in $\mathcal{B}_X$. Consequently, the attribute $W_{\mathcal{B}_X}$, indicating for each node of $\mathcal{B}_X$ whether of not it is a watershed node, provides all the necessary information to reweight the edges of the MST of G that lie within X.

This injection is illustrated on Figure 5.4. The graph $X = (\{a, c\}, \{(b, d)\})$ is equivalent to its MST. We can see on Figure 5.4b that the edge of the MST $\{a, c\}$



(a) Graph G (b) BPH $\mathcal{B}$ built on G

Figure 5.4: **(a)** edge-weighted graph G partitioned into two tiles $\{a, c\}$ and $\{b, d\}$. **(b)** BPH $\mathcal{B}$ built on the graph G. The hierarchy select $(\{a, c\}, \mathcal{B})$ is in black. The node of altitude 1 corresponding to the unique MST edge within the first tile is outlined in red.

corresponds to the node of altitude 1 in $\mathcal{B}_X$. This confirms that any edge of the MST corresponds to a node in the local hierarchy associated with the tile $\{a, c\}$. Thus, we can reweight the edges of the MST of X considering the subset of nodes associated with an edge in X that are watershed nodes.

However, the graph induced by these edges may not be connected, making previously defined algorithms inapplicable. To solve this, instead of reweighting only the MST in $E(X)$, we reweight edges of the mst X_{mst} computed only on X. This is achieved by assigning weights to each edge $e \in E(X_{\text{mst}})$ according to the following rule:

$$w(e) = \begin{cases} 0, & \text{if } W_{\mathcal{B}_X}(\mu^{-1}(e)) = \text{false} \\ 1, & \text{otherwise} \end{cases} \tag{5.9}$$

The distribution of the reweighted graph can be build using the exact same algorithms we used previously.

3. Algorithm 18 Lines 9-10. Finally, the horizontal-cut can be performed locally on each local hierarchy. In order to maintain consistency between the labels across multiple tiles, each leaf node receives the `nodemap` value its highest parent region R under the cut. This ensures that all vertices in the same subtree share the same label, even if they are located in different local hierarchies, *i.e.*, even if the region spans over multiple tiles. Following this scheme, once the new distribution is computed, we can compute separately the watershed associated with each tile of the original partition.

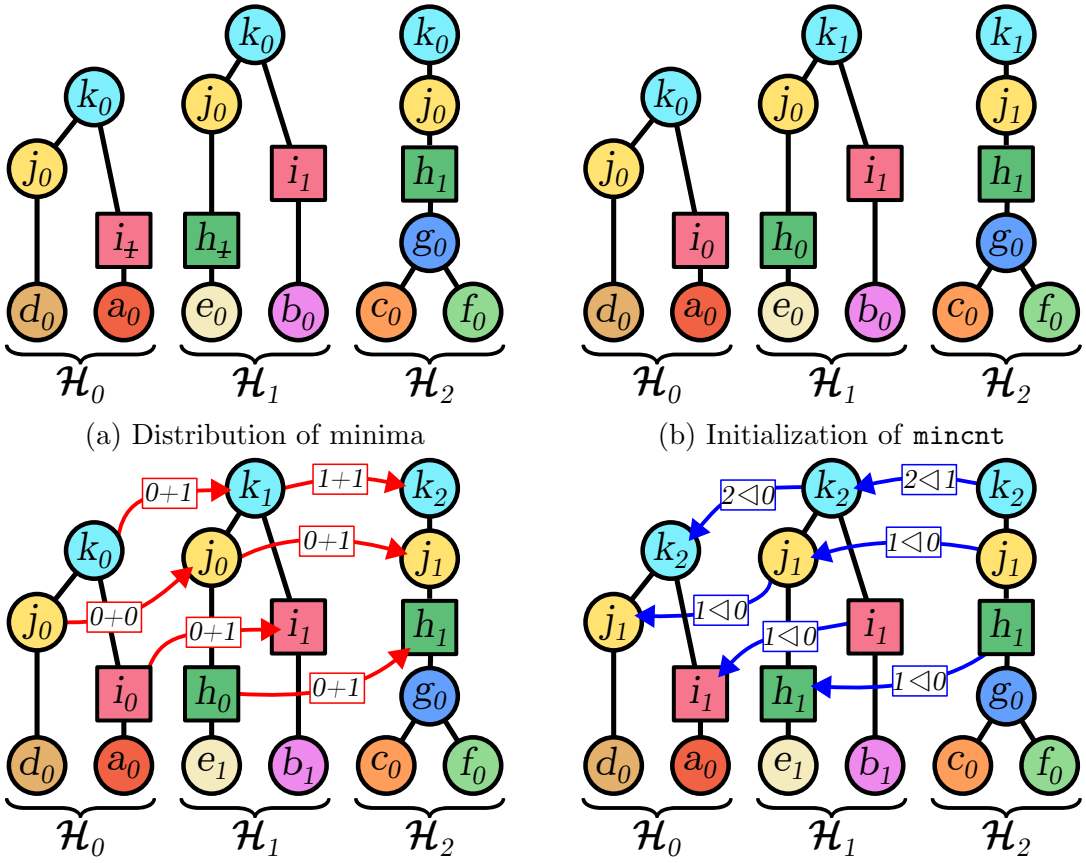
The workflow detailed here is not directly applicable to the computation of the watershed-cut rooted in the regional minima. In the next section, we will describe how to adapt the previously described algorithm to this specific case.

5.5.2 Rooted in Minima

The key difference between marker-based watershed and the minima-based watershed lies in how we count the number of minima within each region of the distribution. While markers are defined on the leaves of the BPH, regional minima are defined on non-leaf regions. This distinction is crucial because it impacts how we propagate minima counts through the distribution.

A naive approach using the same method as for markers by first applying Accumulate and then Propagate with the + operator as for marker count, would lead to an overestimation of the minima count. This issue arises because each leaf belongs to one and only one local hierarchy while a non-leaf node can be replicated in several local hierarchies. For example, in Figure 5.5a, the node h is present in both $\mathcal{H}_1$ and $\mathcal{H}_2$ and i appears in both $\mathcal{H}_0$ and $\mathcal{H}_1$. If we were to use the same propagation strategy as for marker count, a single minimum would then

be counted multiple times. Instead, we must ensure that only one occurrence of each minimum contributes to the computation of the attribute for its ancestors.



(a) Distribution of minima

(b) Initialization of `mincnt`

(c) Causal propagation with $+$

(d) Anti-causal propagation with $\triangleleft$

Figure 5.5: Minima number attribute computation.

To solve this, we introduce the *RightmostSlice* attribute, to help arbitrarily designates the rightmost occurrence of each region across the distribution. This attribute, denoted by `right`, maps to every region R of $\mathcal{H}$, the highest index of a slice containing a vertex of R.

To compute the *RightmostSlice* denoted RM for each region, we locally initialize an attribute $I_{\mathcal{B}_i}(R) = i$ for each region R of $\mathcal{B}_i$ in the distribution $\delta_{\mathcal{H}}$ and then we call Propagate with the supremum operator.

Algorithm 19: RIGHTMOST SLICE DISTRIBUTION

Params: A distribution $\delta_{\mathcal{H}} = \{\mathcal{B}_0, \ldots, \mathcal{B}_k\}$ of hierarchies.
Result: Distribution of rightmost attribute.

```
foreach i from 0 to k do
    foreach region R of B_i do right_{B_i}[R] := i
return PROPAGATE((right_{B_0}, ..., right_{B_k}), δ_H, ∨)
```

We now define the minima number attribute local initialization C^L, which counts the number of minima within each region using the recursive expression:

$$C^L_{\mathcal{B}_i}(n) = \begin{cases} 0, & \text{if } n \text{ is a leaf} \\ 1, & \text{if } n \text{ is a minimum} \\ \sum_{c \in \texttt{children}(n)} C^L_{\mathcal{B}_i}(c) \mid RM_{\mathcal{B}_i}(c) = i, & \text{otherwise} \end{cases} \tag{5.10}$$

We use RM such that only the rightmost occurrence of each minimum contributes to the local count. This is illustrated on Figure 5.5b, region h in $\mathcal{H}_1$ counts as 0 because its rightmost occurrence is located in $\mathcal{H}_2$. With this adjustment, local initialization is performed using ACCUMULATE with the + operator, as shown on Figure 5.5b. The distribution of minima number attribute is then obtained by applying PROPAGATE with the same + operator.

Algorithm 20: MINIMA NUMBER DISTRIBUTION

Params: A distribution $\delta_{\mathcal{H}} = \{\mathcal{B}_0, \ldots, \mathcal{B}_k\}$ of hierarchies.
Result: Distribution of minima number attribute.

```
right := RIGHTMOSTSLICEDISTRIBUTION(δ_H)
mini := MINIMADISTRIBUTION(δ_H)
foreach i from 0 to k do
    Initialize nbmin_{B_i} to 0 for every node
    foreach non-root region R of the each B_i in topological order do
        if mini_{B_i}[R] = true and right_{B_i}[R] = i then
            nbmin_{B_i}[R] := 1
        nbmin_{B_i}[B_i.par[R]] += nbmin_{B_i}[R]
return PROPAGATE((nbmin_{B_0}, ..., nbmin_{B_k}), δ_H, +)
```

The complete out-of-core watershed algorithm is described in Algorithm 21. The unique change compared to the out-of-core marker-based watershed is the computation of the WATERSHEDNODES input which use Algorithm 20.

Algorithm 21: Out-of-core watershed

Params: A distribution $\delta_{\mathcal{H}} = \{\mathcal{B}_0, \ldots, \mathcal{B}_k\}$ of hierarchies.
Result: A series of watershed partition $WS_0, \ldots, WS_k$

```
// Compute minima number distribution
```
$(\texttt{cnt}_{\mathcal{B}_0}, \ldots, \texttt{cnt}_{\mathcal{B}_k}) :=$ MinNbDistribution$(\delta_{\mathcal{H}})$ ▷ Algorithm 20
```
// Compute the watershed nodes
```
foreach i from 0 to k **do**
  $\texttt{ws}_{\mathcal{B}_i} :=$ WatershedNodes$(\mathcal{B}_i, \texttt{cnt}_{\mathcal{B}_i})$ ▷ Algorithm 15
```
// Reweight the MST of each hierarchy
```
foreach i from 0 to k **do**
  $G_i :=$ ReweightMST$(\mathcal{B}_i, \texttt{ws}_{\mathcal{B}_i})$ ▷ Algorithm 16
```
// Compute the distribution on the reweighted MSTs
```
$\left(\mathcal{B}_0^W, \ldots, \mathcal{B}_k^W\right) :=$ Distribution$((G_0, \ldots, G_k))$ ▷ Algorithm 3
```
// Perform horizontal cuts at altitude 1
```
foreach i from 0 to k **do**
  $\mathbf{W}_i :=$ HorizontalCut$(\mathcal{B}_i^W, 1)$ ▷ Algorithm 17
return $(\mathbf{W}_0, \ldots, \mathbf{W}_k)$

5.6 Out-of-Core Hierarchical Watershed

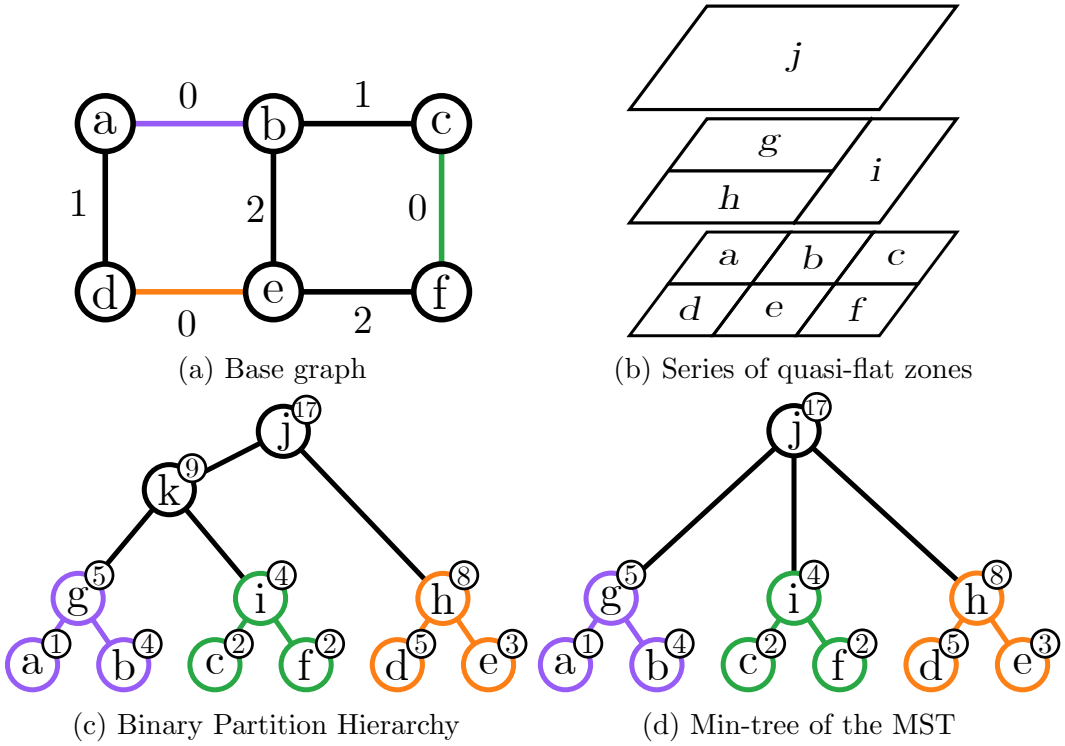
Following the same principles as in the previous section, we will first describe the computation scheme for the classical in-core hierarchical watershed, as outlined in Section 3.6, before addressing the out-of-core case.

5.6.1 In-Core Hierarchical Watershed

In Section 3.6, we introduced the notion of hierarchical watershed based on attributes as a series of nested connected filtering of a graph. Such filtration can be performed on the MST reweighted by the attribute of interest in a similar fashion to the watershed cut.

However, in this setting, the attribute must be preprocessed so that filtering the BPH becomes equivalent to filtering the min-tree of the MST [28].

The issue arises when certain regions appear in the BPH but not in the min-tree of the MST also called *Quasi-flat Zones*. For example in Figure 5.6, the region k of area 9 appears in the BPH but not in the min-tree. This occurs because edges $\{a, b\}$ and $\{c, f\}$ have the same weight, causing all three minima to merge into a single region at altitude 1. As a result, the connected component $\{a, b, c, d\}$ does not truly exist as an independent structure in a level set as outlined in the series of partition induced by the min-tree of the MST in Figure 5.6b.



(a) Base graph (b) Series of quasi-flat zones

(c) Binary Partition Hierarchy (d) Min-tree of the MST

Figure 5.6: **(a)** base graph with three minima outlined in color, **(b)** partition hierarchy corresponding to the min-tree, **(c)** BPH built on the graph, **(d)** min-tree of the MST. In both trees, the area associated with each region is indicated in the top right corner.

This leads to inconsistencies when filtering components based on their attribute values. Suppose we remove all components of area≤ 5, both regional minima $\{a, b\}$ and $\{c, f\}$ would be eliminated. However, the region k of area 9 would persist in the BPH, despite no longer containing any minima.

To solve this issue, we will have to re-evaluate the attribute associated with each node whose altitude is equal to the one of its parent, *i.e.*, regions that does not correspond to any connected component of a level set. For such nodes, we assign the maximum attribute value among their children. In our example the region k will take the value 5 being the area of its most important minima g. This ensures that the region k will be removed from the BPH when the largest minimum it contains is filtered out, in this case $\{a, b\}$.

Then attribute values has to be updated according to canonized regions of the quasi-flat zones. This attribute canonization procedure is depicted in Algorithm 22. The algorithm traverses the BPH in topological order and act as follows:

- If a node has the same altitude as its parent, it represents an artificial region, and we must take the maximum attribute value among its children.
- If a node has a different altitude from its parent, it corresponds to an actual level-set component. Its value is set to the original attribute value.

Once the attribute is corrected, we can compute the persistence associated with each region in the BPH. This is done by applying the ACCUMULATE algorithm with the $\wedge$ operator. The MST is then reweighted by setting the weight of the edges of the MST to the persistence of the corresponding regions in the BPH.

Similarly to the watershed cut, we build a new BPH on the reweighted MST and perform a horizontal cut at level k to obtain the watershed partition of $\Phi_k(G)$, the filtration of the graph G at level k. In the next section, we will give the methodology to compute the hierarchical watershed in an out-of-core setting by computing the persistence and the corrected attribute in the same manner.

5.6.2 Out-of-Core Persistence

As seen previously, horizontal cuts can be performed locally on each local hierarchy. Then, the challenge behind the out-of-core computation of hierarchical watershed lies in the computation of the corrected attribute and the persistence.

Both attributes require knowledge of each region's children, followed by a call to PROPAGATE to distribute the corrected attribute and compute persistence accordingly.

For the corrected attribute, we apply PROPAGATE with the $\vee$ operator to ensure that the highest attribute value across all occurrences of a region is retained throughout the distribution.

In contrast, the persistence attribute is computed differently. Since persistence must reflect the minimum attribute value among a region's children, we first apply

Algorithm 22: QFZ ATTRIBUTE

Data: A hierarchy $\mathcal{B}$, an attribute to canonize `attr`.
Result: An updated attribute `canonized`.

```
Initialize an array canonized of size |B| with 0
foreach non-root region R of B in topological order do
    if B.alt[n] ≠ B.alt[B.par(R)] then
        canonized[n] := attr[n]
    canonized[B.par(R)] := max(canonized[R], canonized[B.par(R)])
canonized[B.root] := attr[B.root]
return canonized
```

ACCUMULATE with the $\wedge$ operator for the local initialization. The final distribution of persistence is then obtained by propagating this value using PROPAGATE with the same $\wedge$ operator. The complete workflow of the out-of-core computation of the persistence is given in Algorithm 23.

Algorithm 23: OUT-OF-CORE PERSISTENCE

Data: A distribution $(\mathcal{B}_0, \ldots, \mathcal{B}_k)$, an attribute function FUNC, and the associated operator $\oplus$.
Result: The distribution of the persistence $(P^{\downarrow}_{\mathcal{B}_0}, \ldots, P^{\downarrow}_{\mathcal{B}_k})$.

```
for i ∈ {0, ..., k} do
    A↑_{B_i} := Func(B_i)
(A↓_{B_0}, ..., A↓_{B_k}) := Propagate((A↑_{B_0}, ..., A↑_{B_k}), ⊕)
for i ∈ {0, ..., k} do
    C↑_{B_0} := QFZAttribute(A↓_{B_0}, C↓_{B_0})          ▷ Algorithm 22
(C↓_{B_0}, ..., C↓_{B_k}) := Propagate((C↑_{B_0}, ..., A↑_{B_k}), ∨)
for i ∈ {0, ..., k} do
    P↑_{B_i} := Accumulate(C↓_{B_i}, ∧)
(P↓_{B_0}, ..., P↓_{B_k}) := Propagate((P↑_{B_0}, ..., P↑_{B_k}), ∧)
```

Given the persistence $\rho_{\mathcal{B}_X}(R)$ of a region R in the local hierarchy $\mathcal{B}_X$ associated with the tile X, we can reweight the edges of the MST of X by setting the weight of the edge corresponding to R to $\rho_{\mathcal{B}_X}(R)$. This is done by using the map μ^{-1} from edges of the global MST included in X to non-leaf regions in $\mathcal{B}_X$ (described in Section 5.5.1).

However, as stated previously for the sake of out-of-core marker-based watershed cut the graph induced by these edges may not be connected, rendering previously defined algorithms inapplicable. To resolve this, instead of reweighting only edges of the global MST included in X, we reweight all edges $E(X_{\text{mst}})$ of the MST of X. This is achieved by assigning weights to each edge $e \in E(X_{\text{mst}})$ according to:

$$w(e) = \begin{cases} \rho_{\mathcal{B}_X}(\mu^{-1}(e)), & \text{if } \mu^{-1}(e) \text{ exists} \\ \infty, & \text{otherwise} \end{cases} \tag{5.11}$$

Thus, edges that don't have an image thought μ^{-1}, *i.e.*, edges that does not belong to the global MST, are weighted to infinity. Other edges are weighted to the persistence of the region they correspond to.

In the exact same manner as for the watershed cut, the λ-scale of the hierarchical watershed associated with a tile can be obtained locally with an horizontal

cut. The consistency of labels is also ensured by associating the `nodemap` of the biggest region R under the cut to each child of R.

5.7 Conclusion

In this chapter, we have presented a comprehensive overview of the Propagate algorithm, which serves as a foundation for a generic and efficient out-of-core attribute computation. This methodology relies on a linear complexity algorithm and requires having only the information about two adjacent tiles in main memory at any step of the algorithm.

On this basis, we give classical attributes such as area, volume, and local minima. It should be noted that the use of this methodology is not limited to the presented attributes, but can easily be used to compute other attributes such as bounding boxes or mean grayscale.

We also show how to adapt the Propagate algorithm to compute watershed segmentation and its hierarchical counterpart in an out-of-core setting.

We gave algorithms to apply this methodology on a causal partition, however from a theoretical point of view, the Propagate algorithm can be applied on a generic partition of the space. Following the same idea as the BPH distribution construction, we would use the Global Border Tree. The adaptation of this methodology to a generic partition is left as future work.

Chapter 6

Superpixel Based Out-of-Core Watershed

6.1 Introduction

In this chapter, we propose a more application-oriented workflow based on our out-of-core framework, which leverages superpixel computation to support marker-based watershed segmentation.

Rather than working at the finest granularity of pixel-level partitions, it is often more practical both in terms of memory usage and execution time, to first group together pixels with similar characteristics. This initial reduction of data complexity leads to the concept of superpixels: perceptually meaningful clusters of pixels that share similar visual properties, such as color, texture, or spatial proximity.

Superpixels offer a compact and semantically relevant representation of images, facilitating various downstream high-level vision tasks including segmentation, object recognition, and tracking. As a result, the generation of superpixels has become a well-established topic in computer vision, with a wide array of approaches developed over the years. We can find popular methods such as SLIC [1] and graph-based techniques like SICLE [11], which is an IFT-based method, or watershed-based methods such as Waterpixels [64].

If we consider our complete out-of-core workflow, the distribution computation presented in Chapter 4 constitutes the bottleneck. Then from an applicative point of view, computing the distribution of the BPH built on superpixels appears to be an interesting alternative to reduce memory consumption and execution time. To ensure full compatibility with our hierarchical segmentation framework, we adopt a specific strategy for superpixel generation based on quasi-flat zones.

After given an overview of the proposed workflow, we formally define quasi-flat

zones and illustrating their connection to a well-known structure in mathematical morphology: the quasi-flat zone hierarchy, also known as the α-tree. We then highlight key properties that make this hierarchical representation particularly suitable for the exact and efficient computation of marker-based watershed segmentation under light constraint. Finally, we describe the construction process of the region adjacency graph distribution and present a series of experiments to demonstrate the practical advantages of our approach.

6.2 Overview

Before describing the proposed workflow, it is useful to briefly recall the pipeline introduced in the previous chapters. In that setting, each tile of the input image is processed independently to compute a partial BPH. The construction of the complete distributed representation then relies on the use of the *select*, *join*, and *insert* operations, which complete and correct the nested region relations of each local hierarchy. As a result, each local tree becomes part of the BPH distribution. Subsequently, using the out-of-core attribute computation techniques introduced earlier, various regional attributes can be computed on these hierarchies to label pixels according to specific segmentation schemes such as the watershed. In this context, pixel labeling is straightforward, since the leaves of each local hierarchy are in bijection with the pixels of the corresponding tile.

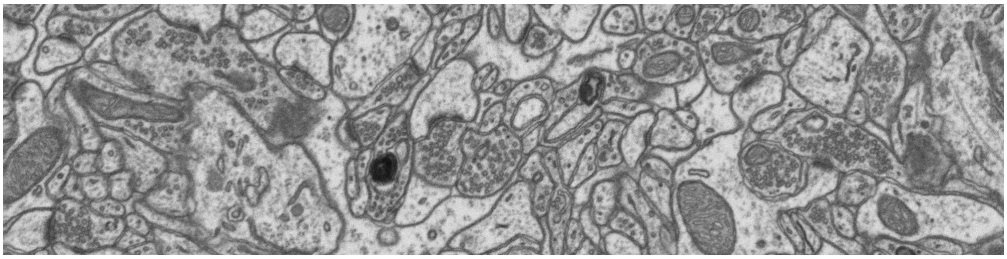
In this chapter, we focus on improving the first stage of the workflow. More specifically, we aim to reduce the number of input vertices and edges per tile, thereby decreasing the overall number of regions generated in the hierarchies and consequently lowering memory usage and computational load. To this end, we propose grouping pixels into superpixels by constructing a Region Adjacency Graph (RAG) within each tile.

This data-structure consists of a graph representing adjacency relationship between adjacent regions. Vertices of the RAG represent regions of the initial graph and edges of the RAG represent the adjacency relation between two regions. In other words, if an edge link two vertices assigned to two distinct region in the initial graph, then this edge might belong to the RAG.

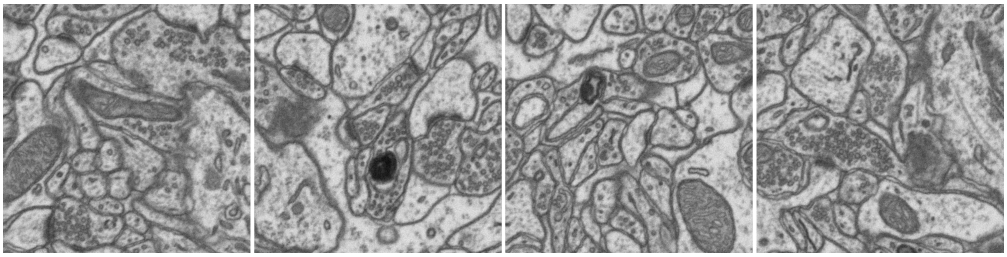
However, to further reduce complexity, we retain only a single edge between any two adjacent superpixels. Specifically, among all edges linking pixels in different superpixels, only the one belonging to the MST is kept. This strategy preserves the essential connectivity between regions while minimizing the total number of edges in the RAG, which contributes to both memory and performance efficiency.

The proposed approach consists in replacing the pixel-level graph in each tile with its corresponding RAG prior to BPH construction. The initialization step of this alternative workflow is illustrated in Figure 6.1: starting from the input image

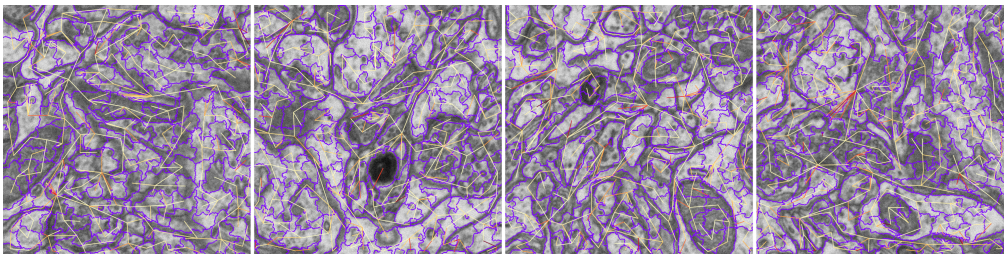
(6.1a), the domain is divided into tiles (6.1b). A RAG is then computed for each tile (6.1c), where purple boundaries denote superpixels and colored lines represent inter-superpixel MST edges, with color indicating edge weights. Finally, partial



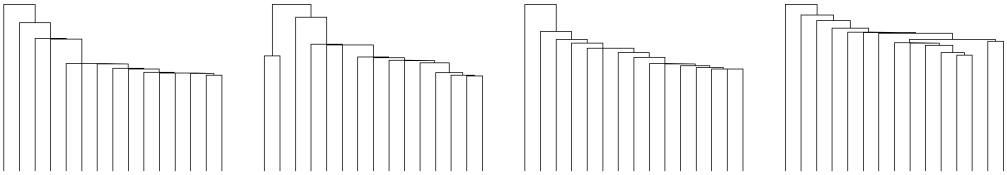
(a) Input image



(b) Tiles



(c) Region Adjacency Graphs



(d) Partial BPHs

Figure 6.1: Illustration of the proposed workflow. **(b)** The input image is partitioned into tiles. **(c)** A region adjacency graph is built for each tile. Frontiers between superpixel in purple and lines represent edges of the minimum spanning tree linking two neighboring superpixels. Edge color represents their weights from light yellow to dark red. **(d)** The partial BPHs are then constructed from these RAGs.

BPHs are computed from these RAGs (6.1d).

This initialization step does not impact the high level calculi Algorithms 2 and 3. However, regarding the return to the realm of pixels after the distribution construction, we need to ensure that the pixel-to-superpixel mapping is maintained. Indeed, leaves of the elements of the distribution are no longer in bijection with pixels, but rather with superpixels. To address this, we introduce a mapping structure that associates each pixel to its corresponding superpixel. This mapping is crucial for projecting the segmentation results back onto the original image grid, ensuring that the final output remains consistent and accurate at pixel resolution.

Before delving into the details of the proposed workflow, we first introduce the concept of quasi-flat zones, which serves as the foundation for our superpixel generation strategy. We then show how these quasi-flat zones can be used to compute the marker-based watershed segmentation efficiently.

6.3 Quasi Flat Zones for Marked Watershed

In this section we introduce first the notion of α-flat zone and show their properties for marked watershed.

6.3.1 Quasi Flat Zones and Hierarchies

Let $G = (V, E, F)$ be an edge-weighted graph and let $\alpha \in \mathbb{R}$. We define the *α-level set* of G as the graph $G_\alpha = (V, E_\alpha, F)$ such that $E_\alpha = \{u \in E \mid F(u) \leq \alpha\}$. Then the *$\alpha$-flat zone* of a vertex x is the connected component of G_α that contains x noted $CC_\alpha(x)$. The set $\mathbf{P}_\alpha = \{CC_\alpha(x) \mid x \in V\}$ of all α-flat zones is a complete partition of V. The imbrication of the partitions for any α in the definition domain result in another popular hierarchy:

Definition 6.3.1 (Quasi-Flat Zone Hierarchy)**.** Let $\{\alpha_1, \ldots, \alpha_n\}$ be the set of all weights of the edges in E in increasing order. The series $(\mathbf{P}_{\alpha_1}, \ldots, \mathbf{P}_{\alpha_n})$ is the *Quasi-Flat Zone Hierarchy* of F.

This structure is also called *α-tree* in the literature [85]. Several variations of this hierarchy were proposed such as the α-ω-tree within the framework of constrained connectivity [110].

The QFZ hierarchy is equivalent to the min-tree of the MST [28] establishing a tight link with the BPH. Najman et al. [81] proposed an algorithm to compute the QFZ hierarchy consisting in a post-processing of the BPH. The proposed algorithm is very simple as it is sufficient to canonize the BPH, *i.e.*, removing any node with a parent of equal altitude, making the procedure linear in the number of nodes.

Then, we can directly build α-flat zones from the BPH in an efficient manner performing an horizontal cut at level α. However, in order to minimize memory consumption and execution time, we will use a dedicated Union-find procedure allowing us to compute the α-flat zones and the induced RAG in a single pass over the input graph.

6.3.2 Marked Watershed on Superpixels

In the previous section, we described how to compute the α-flat zones within the BPH framework. In this section, we extend this approach to the computation of the marker-based watershed, leveraging the concept of watershed nodes introduced in Section 5.5.1.

The key idea is as follows: computing the BPH over a graph composed of α-flat zones is equivalent to first computing the BPH on the original pixel-level graph and then collapsing every subtree entirely contained within a single α-flat zone. This simplification preserves the hierarchical structure while reducing the number of regions.

Now consider the placement of markers. If two markers are located in distinct α-flat zones, their lowest common ancestor in the BPH, referred to as the watershed node, will necessarily have an altitude greater than α. Since this node spans multiple flat zones, it is preserved during the simplification process. Therefore, the watershed node identified in the superpixel-wise BPH (constructed from α-flat zones) will be identical to the one found using the original pixel-wise BPH.

On the other hand, if two markers are placed within the same α-flat zone, then the edge corresponding to their watershed node will lie entirely within that α-flat zone and will be simplified away belonging to a collapsed subtree. In this case, the watershed cut cannot be computed reliably, as the hierarchical structure does not retain sufficient information to separate the markers.

This observation leads to a simple yet important constraint: each marker must be placed in a distinct α-flat zone. Fortunately, this condition is easy to satisfy in practice. It can either be enforced as a pre-processing step by assigning each α-flat zone the marker label that covers the majority of its pixels or directly integrated into the user interface, as illustrated in Figure 6.2.

6.4 Distribution of Region Adjacency Graph

In this section we describe how to compute the distribution of the RAG built from the α-flat zones. We first present an algorithm to compute the α-flat zones and the induced RAG from an edge-weighted graph. This algorithm is based on a union-find data structure. Then we describe how to adapt the distribution computation

presented in Chapter 4 for this RAGs.

6.4.1 From Quasi-Flat Zones to Region Adjacency Graph

As previously discussed, several strategies exist for constructing α-flat zones. In this section, we present a unified algorithm that simultaneously builds the α-flat zones and the induced RAG. This process is formally detailed in Algorithm 24.

Let $G = (V, E, w)$ be an edge-weighted graph, let α be a predefined threshold and let $G_{\text{rag}} = (V_{\text{rag}}, E_{\text{rag}}, w_{\text{rag}})$ be the RAG induced by the α-flat zones of G. The first step involves identifying the α-flat zones, which correspond to the connected components of the thresholded graph $G_\alpha = (V, E_\alpha)$. To this end, we initialize a union-find data structure Q_T, which is used to iteratively merge vertices connected by edges of weight less than α. This operation is performed in increasing order of edge weights, effectively segmenting the graph into its α-flat zones.

Once all the connected components have been identified, we construct a label array L that maps each vertex $v \in V$ to the index of its corresponding α-flat zone

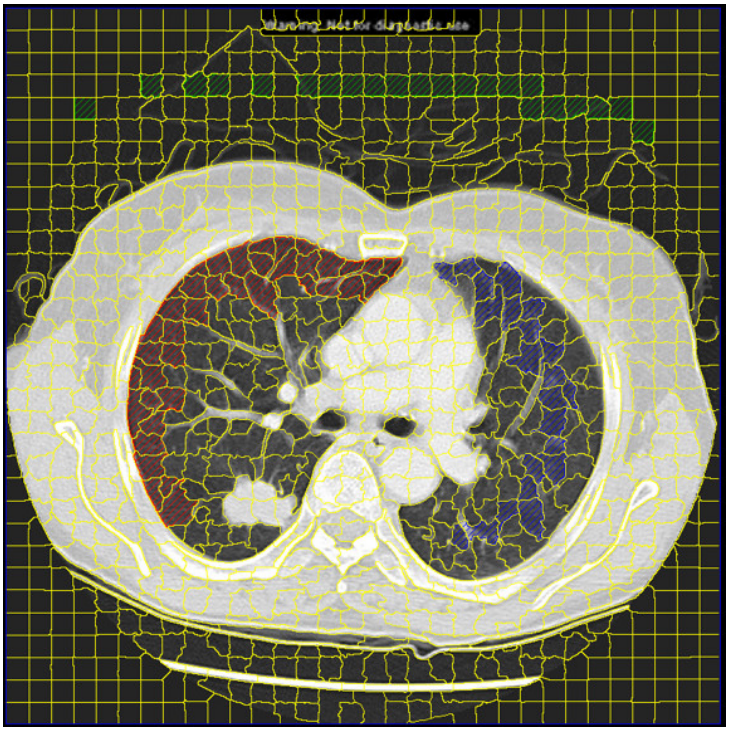


Figure 6.2: Example of superpixel obtained with Waterpixels [64] in Amira software. Yellow lines defines the boundaries of the superpixels. Red markers are placed in the left lung, blue markers in the right lung and green markers in the background.

Algorithm 24: α-FLAT ZONES REGION ADJACENCY GRAPH

Data: A weighted graph $G = (V, E, w)$ and a value α.
Result: A weighted RAG $G_{\text{rag}} = (V_{\text{rag}}, E_{\text{rag}}, w_{\text{rag}})$ induced by α-flat zones and a label array L.

$E :=$sort(E)
foreach *vertex* v_i *of* V **do** Q_T.MakeSet(i)
$ncc := |V|;\ i := 0$
while $i < |E|$ **and** $w[E[i]] < \alpha$ **do**
    $\{u, v\} := E[i]$
    $c_u := Q_T$.FindCanonical(u); $c_v := Q_T$.FindCanonical(v)
    if $c_u \neq c_v$ **then**
        Q_T.Union(c_u, c_v)
        $ncc := ncc - 1$
    $i := i + 1$
for $j := 0$ **to** $n - 1$ **do** $L[i] := Q_T$.Find(j)
Remap L to consecutive labels $0, \ldots, ncc$ - 1 ▷ $O(|V|)$
$V_{\text{rag}} := \{0, \ldots, ncc$ - $1\}$
Initialize two arrays E_{rag} and w_{rag} with ncc elements
while $ncc > 1$ **do**
    $\{u, v\} := E[i]$
    $c_u := Q_T$.FindCanonical(u); $c_v := Q_T$.FindCanonical(v)
    if $c_u \neq c_v$ **then**
        Q_T.Union(c_u, c_v)
        $E_{\text{rag}}[|V_{\text{rag}}| - ncc] := \{L[u], L[v]\}$
        $w_{\text{rag}}[|V_{\text{rag}}| - ncc] := w[E[i]]$
        $ncc := ncc - 1$
    $i := i + 1$
return $G_{\text{rag}} = (V_{\text{rag}}, E_{\text{rag}}, w_{\text{rag}})$, L

$v' \in V_{\text{rag}}$. These labels are remapped to form a contiguous index range facilitating their management regarding the out-of-core workflow.

The second step involves building the RAG that captures the adjacency relationships between the computed α-flat zones. The vertices of the RAG represent the α-flat zones, and its edges are induced from the MST of the original graph. Let $T = (V, E_{\text{mst}})$ be the MST of G. An edge $\{x, y\} \in E_{\text{mst}}$ contributes to the RAG if x and y belong to distinct α-flat zones. Consequently, the RAG edge set can be expressed as:

$$E_{\text{rag}} = \{\{L(x), L(y)\} \mid \{x, y\} \in E_{\text{mst}},\ L(x) \neq L(y)\} = E_{\text{mst}} \setminus E_{\alpha}$$

The construction of E_{rag} is implemented in the second while loop of Algorithm 24. Each edge of the MST linking two distinct α-flat zone is recorded along with its original weight.

It is interesting to note that at the end of Algorithm 24, edges of the RAG are sorted in increasing order of their weights. Thus, it is not necessary to sort the edges prior the BPH computation. Plus, looking forward watershed and hierarchical watershed computation there is no need for sorting these edges in order to reweight them after the watershed node or persistence computation.

6.4.2 Out-of-core Region Adjacency Graph

The construction of the distribution of the RAG does not fundamentally differs from the one at pixel level. Roughly, the only step to be changed is the computation of the partial BPH. Indeed, we need to build a RAG for each tile of the initial partition using the methodology described in the previous section. For each individual RAG, a partial BPH is computed using *Playing with Kruskal* algorithm. These hierarchies now represents the merging of α-flat zones rather than individual pixels.

However, we need to make sure that the three data structures `map` and `weight` used to build the partial BPH are maintained from the original graph to its associated RAG. First, `weight` which stores the weight of edges should associate to each edge in the RAG the weight of the corresponding edge in the original graph. Second, `map` behave as a mapping from leaves regions to their corresponding superpixel and from non-leaf regions to their corresponding edge in the RAG. While index of edges in the original graph are preserved, the mapping from superpixels to pixels is updated to reflect the new superpixel representation. Each superpixel in the RAG is associated with a unique index which is decided arbitrarily during the construction of the RAG.

Note that we can use the strategy described in Remark 6.4.1 to maintain the total ordering on edges and get rid of the `map` structure.

> **Remark 6.4.1.** *In order to maintain the initial total ordering on edges, the ordering is explicitly encoded using a combination of edge weights and* `map` *as tie-breaking value which is the index of the edge in the base graph. It is not mandatory to do so but from a practical point of view, it is more convenient to reduce the number of array to maintain between the different steps of the algorithm. To do so we encode both values into a single 64-bit value. Taking the example of 8-bits weights, we reserve the first 8 bits to encode the edge weight, and the remaining 56 bits to store the tie-breaking index. The new weight $cw(w, i)$ associated with an edge of weight w and index i is computed*

as follows:

$$cw(w, i) = (w \ll 56) \mid (i \,\&\, 0x00FFFFFFFFFFFFFF)$$

This compact representation ensures that the total order is maintained throughout the workflow while simplifying comparisons. Plus, it does not involve redefining insert algorithm, we can decode the original values when needed or mask the encoded weights to retrieve the value of interest.

Since these data structures are maintained, there is no modification needed for the operations *select*, *join*, and *insert*. However, *join* can be the object of an optimization relying on a simple heuristic to reduce the number of edges of the common neighborhood of two slices. Indeed, the common neighborhood F of two slices is defined as the set of edges linking the two slices. Initially, as we are working either on 4- or 6-connectivity, there exists at most one edge between two vertices. However, when we build the RAG on each tile, we may have multiple edges between two superpixels. As we are interested into edge of the minimum spanning tree, for each pair of superpixels we can only retain the edge of the minimum weight in order to reduce the number of edges to consider during the Union-find procedure. Thus for any two distinct tiles S_i and S_j and for any two vertices $X \in S_i$ and $Y \in S_j$, we define the new common neighborhood of S_i and S_j as:

$$F' = \left\{ e = \{X, Y\} \;\middle|\; w(e) = \min_{\substack{e' \in E \\ e' = \{X,Y\}}} w(e') \right\}$$

Finally, in addition the tree data structures, we also need maintain the mapping from each pixel to its corresponding α-flat zonefor each tile. This mapping enables us to project the segmentation result, computed on the collection of RAG, back onto the original image grid. Thus, although the computation is performed on a coarser representation, the final output remains fully consistent and accurate at pixel resolution.

However, it is important to note that the distribution obtained through this process is not a strict distribution of the BPH built from the α-flat zones. Due to tiling, some QFZs may be split across tile boundaries, resulting in an over-segmentation of the α-flat zones. To enforce a strict BPH distribution over α-flat zones, it would be necessary to canonize regions whose ranks is less than equal to α. This canonization step result in merging any regions that does not belong to the α-flat zones.

However, implementing this canonization would also require updating the pixel-to-superpixel map, which incurs I/O overhead. Given that our objective is to minimize memory usage while keeping execution time low, the memory savings

from canonization do not justify the additional I/O cost. We therefore accept this mild over-segmentation as a trade-off for a more efficient and practical computation pipeline.

In the next section, we will show performances of the distribution construction and the impact of the α parameter on the performance of the distribution construction.

6.5 Experiments

6.5.1 Databases and Test Images

All experiments were conducted on a workstation equipped with an Intel Xeon W-2255 3.70 GHz CPU and 256 GB of RAM. The dataset used for evaluation is a Focused Ion Beam Scanning Electron Microscopy (FIB-SEM) volume comprising 3.3 gigavoxels [62]. Given the characteristics of neuron boundaries, edge weights were defined as the minimum extremity intensity of the inverted image.

For the study of α-flat zone impact, four different α values were tested: 66, 72, 82, and 111. These thresholds were selected to preserve meaningful boundaries while progressively coarsening the image structure. Figure 6.3 illustrates 2D slices of the volume showing the α-flat zone superpixels obtained for these values. Each superpixel is represented using the average grayscale of the pixels it contains.

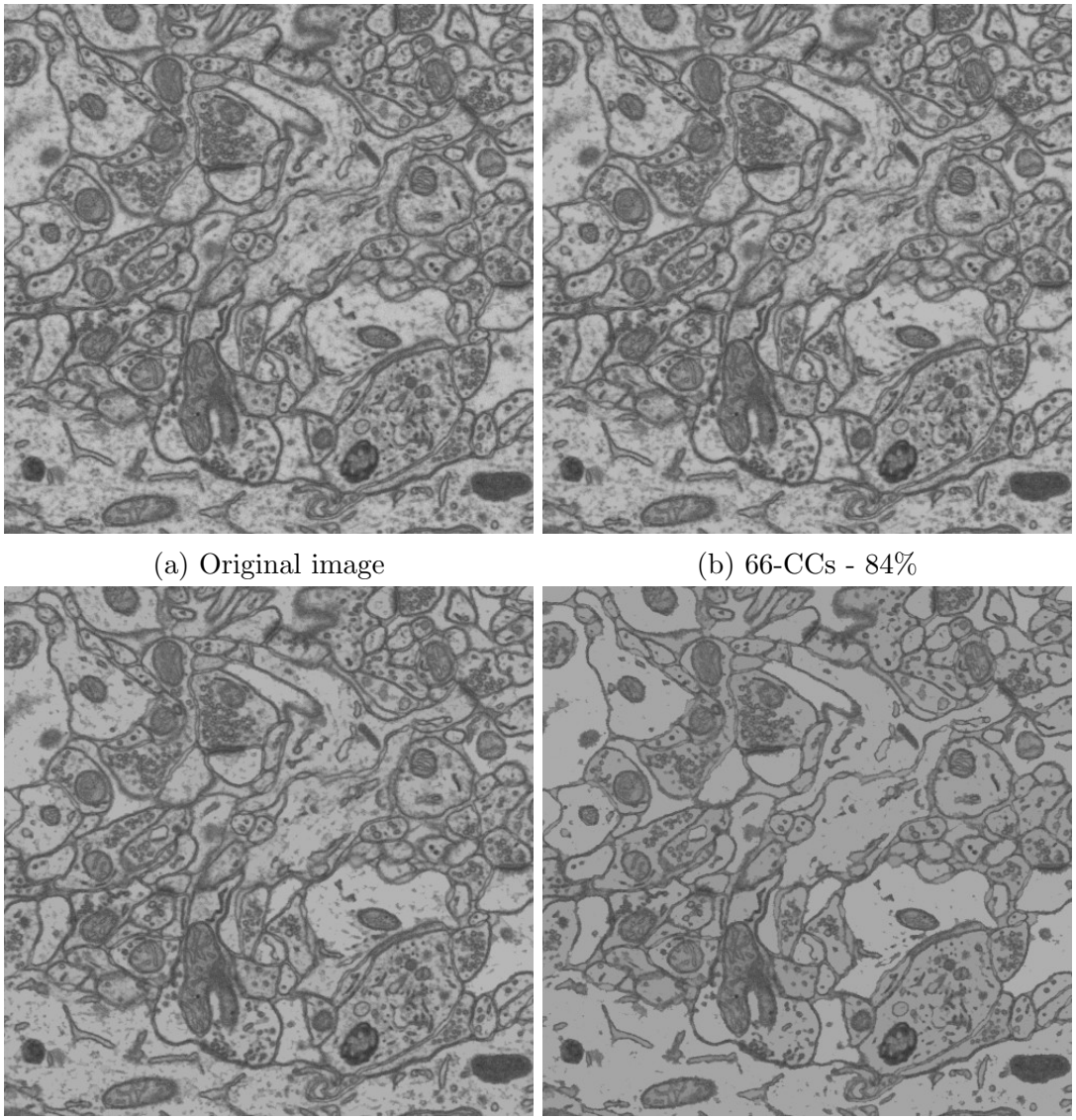
6.5.2 Evaluated Methods

Two configurations were compared for the construction of distributions within the out-of-core framework developed at Thermo Fisher Scientific.

In the `Classic` configuration, the distribution is computed directly at the pixel level using a grid-like partition and the Global Border Tree scheme. This baseline represents the standard implementation of the out-of-core distribution construction, where each pixel is treated as an individual element in the hierarchy.

In the `QFZ` configuration, the same computation is applied to a causal partition where pixels are grouped into superpixels defined by the α-flat zone method. The resulting framework therefore operates at the level of quasi-flat zones rather than individual pixels, which reduces the effective number of elements in the graph and consequently the memory footprint.

In both cases, the pipeline follows the principles described in Chapter 4, but relies here on an in-house implementation.



(a) Original image (b) 66-CCs - 84%

(c) 82-CCs - 63% (d) 111-CCs - 33%

Figure 6.3: Visualization of the α-flat zones obtained for different α values. The number of α-flat zones and the percentage of remaining regions compared to the original pixel-level partition are indicated below each image.

6.5.3 Metrics

Three main quantities were measured to assess performance:

1. *Peak memory consumption*, corresponding to the maximum resident memory used during computation.

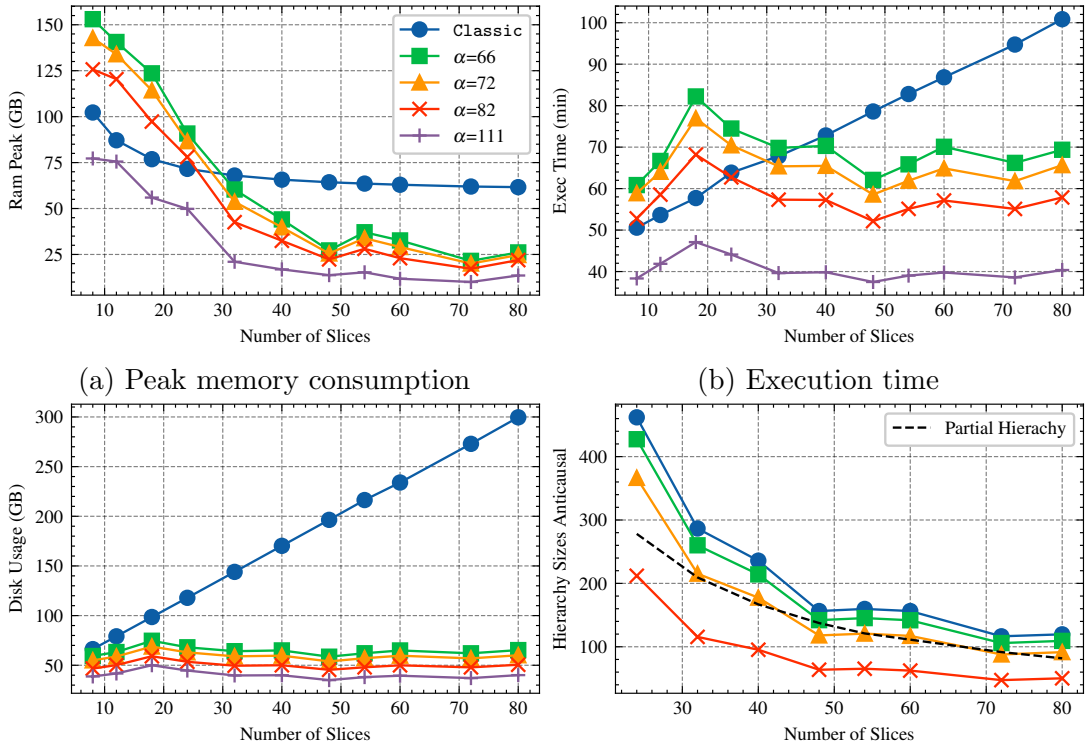
2. *Disk footprint*, defined as the maximum volume of temporary data written to disk during the out-of-core execution.

3. *Execution time*, recorded as the total processing duration from start to completion.

4. *Hierarchy size*, expressed as the number of regions of the largest element of the distribution, was analyzed and compared to the number of regions of a partial hierarchy *i.e.*, the BPH built on one tile.

6.5.4 Evaluation Results

Figure 6.4 summarizes the comparative results obtained for the `Classic` configuration and for the `QFZ`-based computation with different α values.



(a) Peak memory consumption

(b) Execution time

(c) Disk footprint

(d) Hierarchy sizes

Figure 6.4: Comparative analysis between pixel-level (`Classic`) and α-flat zone-based (`QFZ`) distribution construction for various α values.

The results show that the `QFZ` configuration significantly reduces peak memory usage compared to the pixel-level baseline. The minimum RAM peak decreases by

65%, 67%, 72%, and 84% for $\alpha = 66, 72, 82$, and 111 respectively. This reduction becomes more pronounced for larger α values, where fewer superpixels lead to a more compact representation of the data. For instance, with $\alpha = 111$, the minimum peak memory corresponds to roughly three times the original image size in memory.

The disk footprint remains nearly constant across different numbers of slices. This stability results from the dominance of the pixel-to-superpixel mapping structure, whose size does not depend on the number of slices. Similarly, execution times vary only slightly with input size, since the computational load shifts from the pixel-wise *insert* phase to the construction of partial hierarchies over the RAG, which scales linearly with the number of pixels.

This is particularly advantageous for processing large-scale volumes. It enables the best performance in terms of peak memory consumption to be reached without incurring prohibitive costs in disk I/O or execution time.

The influence of α on the hierarchy size further explains these observations. As shown in Figure 6.4d, the number of regions per hierarchy decreases as α increases. This is consistent with the reduction in the number of quasi-flat zones and directly affects both memory and time consumption. For $\alpha = 111$, the number of regions in the hierarchy is consistently lower than the baseline BPH in black dashed, confirming that the *insert* step remains the dominant factor for memory usage.

Quantitatively, for a volume of 80 slices, disk usage decreases from approximately 299 GB for the `Classic` configuration to 65 GB and 40 GB for $\alpha = 66$ and $\alpha = 111$ respectively. Execution time follows the same trend, dropping from 100 minutes to 69 minutes and 40 minutes. These improvements confirm that the `QFZ` approach effectively balances segmentation precision and computational efficiency.

In summary, the `QFZ`-based approach offers substantial reductions in memory usage, disk I/O, and execution time while maintaining scalability and robustness for large-scale volumetric processing. Adjusting the α parameter provides a practical means to trade off segmentation granularity against performance, making the method particularly well suited for out-of-core hierarchical computations on massive datasets.

6.6 Conclusion

In this chapter, we have presented a practical workflow for out-of-core marker-based watershed segmentation using superpixels. By leveraging the concept of α-flat zones, we have shown how to efficiently compute the distribution of the region adjacency graph to enforce low memory consumption and execution time. We first gave a warranty that it is possible to compute an exact watershed on this simplified

distribution up to the light constraint of placing markers in distinct α-flat zones. We then gave an algorithm for the construction of the region adjacency graph distribution and presented a series of experiments to demonstrate the practical advantages of this approach.

The results indicate that our method significantly reduces peak memory consumption down to 3 time the image size in the main memory, enabling the processing of larger volumes without incurring excessive costs in disk I/O or execution time. Additionally, we highlighted the ability to tune the parameter α to balance segmentation granularity with processing cost, further enhancing the scalability of the method.

Chapter 7

Incremental Interactive Watershed Segmentation

Related Publications

- Q. Lebon, J. Lefèvre, J. Cousty, and B. Perret. Interactive segmentation with incremental watershed cuts. *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, page 189–200, Nov. 2023. 10.1007/978-3-031-49018-7_14
- Q. Lebon, J. Lefèvre, J. Cousty, and B. Perret. Incremental watershed cuts: Interactive segmentation algorithm with parallel strategy. *Pattern Recognition Letters*, 189:256–263, Mar. 2025. 10.1016/j.patrec.2024.12.005

7.1 Introduction

While the previous chapters focused on designing an out-of-core framework to enable large-scale hierarchical segmentation, our research also explored a complementary direction: interactive segmentation. This investigation was conducted in a different context but shares the same overarching goal, making image segmentation both scalable and user-friendly.

Among the many segmentation techniques, marker-based watershed has proven particularly effective in interactive settings. By replacing traditional minima with user-defined markers, this approach simultaneously addresses the problem of over-segmentation and introduces semantic priors into the segmentation process. As a result, the output is not only structurally coherent but also aligned with user intent.

Typically, an initial segmentation with manually defined markers, the user often iteratively refine the segmentation by adding, removing, or adjusting markers

until a satisfactory result is achieved. However, in many practical scenarios, especially when dealing with large 2D images or volumetric data such as MRI or FIB-SEM scans, the interactive process can become prohibitively slow. This is largely due to the fact that classical marker-based watershed algorithms do not support incremental updates. Each change in the markers configuration, even if it affects only a few pixels, requires the entire segmentation to be recomputed from scratch, significantly hindering user experience.

To address this limitation, the Differential Image Foresting Transform (DIFT) was introduced in [35] as an extension of the Image Foresting Transform (IFT) [36]. DIFT allows for efficient, incremental updates of shortest-path forest, which can be related to watershed segmentation [27], such that the computation time becomes proportional to the size of the modified regions.

Interactive segmentation has also been explored through other paradigms, including power watersheds [114], random walker algorithms [42], geodesic path models [93]. Additionally, hierarchical representations such as component trees [89, 20, 83] and segmentation hierarchies [98] have been leveraged to enable efficient user-driven segmentation. In these cases, the hierarchy is computed once, and the segmentation is updated after each user interaction without needing to recompute the full structure. This can be seen as a form of incremental segmentation, where the hierarchy is used to guide the segmentation process. However, these methods often require a complete recomputation of the output after each user interaction, which can be computationally expensive and time-consuming.

In this chapter, we propose an new way to compute marker-based watershed cut on the BPH. We show that this method is particularly compliant with an incremental process exploiting the causality within the interactive sessions to achieve remarkable performances, significantly improving responsiveness. The following is structured as follows:

- Section 7.2 we introduce an algorithm to compute marker-based watershed segmentation using the BPH of the input image. This procedure consists of two main steps: marker propagation within the BPH and connected component labeling.

- Section 7.3.1 we show how to efficiently update both marker propagation and connected component labeling when the markers are modified, making the method suitable for real-time interactive segmentation.

- Section 7.3.2 we propose a parallel version of the labeling step to further accelerate the update process.

The proposed approach is evaluated on several 2D and 3D images under both simulated and real user interactions, demonstrating that our algorithm supports

updates in time proportional to the number of affected pixels. Run-time comparisons with both incremental and non-incremental state-of-the-art methods confirm the significant performance advantage of our method in interactive settings. To facilitate reproducibility and further research, an open-source C++ implementation of the proposed techniques is publicly available[1].

This chapter is directly adapted from the journal publication Lebon et al. [56], with minimal modifications to fit the structure of the thesis.

[1] `https://github.com/lebonq/incremental_watershed`

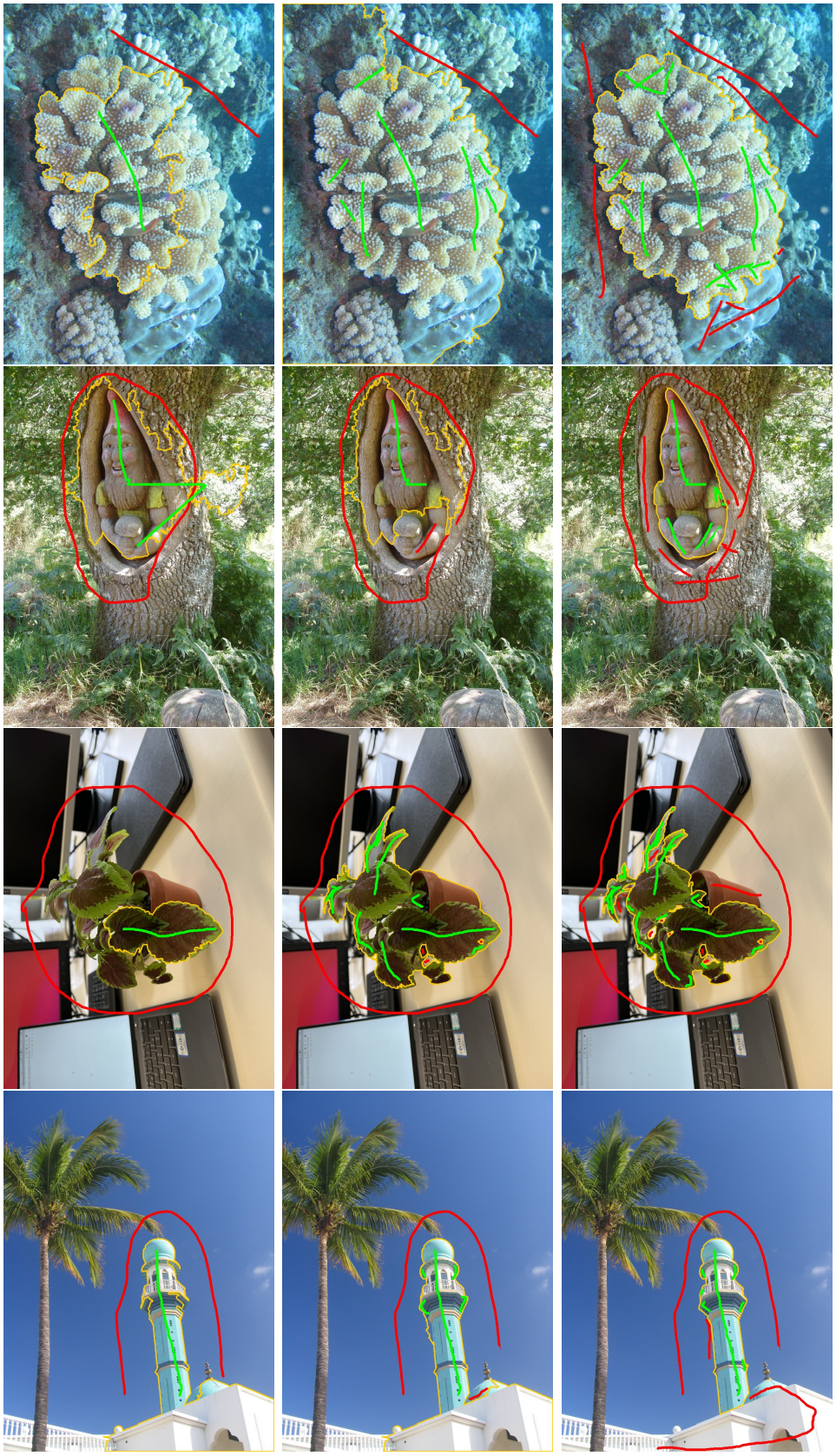


Figure 7.1: Three stages in an interactive segmentation session. The red lines represent background seeds, the green lines represent object seeds, and the yellow lines represent the segmentation produced by the current set of seeds.

7.2 A New Computation of Watershed Cut Based on Binary Partition Hierarchy

This section introduces a new efficient algorithm to compute a WS cut from user-provided markers, and to update its result based on user's feedbacks given in the form of successive deletions and additions of markers.

The overall workflow is presented in Figure 7.2. It comprises three main parts:

1. Computation of an MST and of an associated BPH;
2. Selection of the watershed edges associated to the markers provided by the user. These edges, which are found by browsing the BPH, correspond to the edges that must be deleted from the MST to obtain a minimum spanning forest rooted in the provided markers; and
3. Partitioning and labeling of the graph vertices according to the connected components (CC) of the forest obtained at step 2. This labeling is the resulting WS cut segmentation.

Step 1 is performed using the algorithm presented in [81]. An efficient algorithm for step 2 is given in discusses Section 7.2.1, Section 7.2.2 discusses the labeling involved at step 3, and Section 7.2.3 shows that the proposed algorithm can be used to incrementally update the results of the workflow based on user's feedback.

7.2.1 Tree-node Marking

First we consider marker addition. To this end, we present Algorithm 25 that returns the watershed edges given a set of marked vertices. The algorithm works incrementally, marker by marker. For each marker, taken in an arbitrary order, one edge of the initial MST must be removed to cut the region of this marker

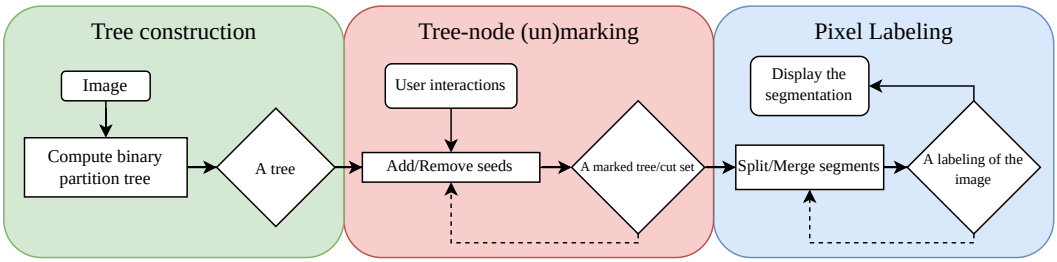
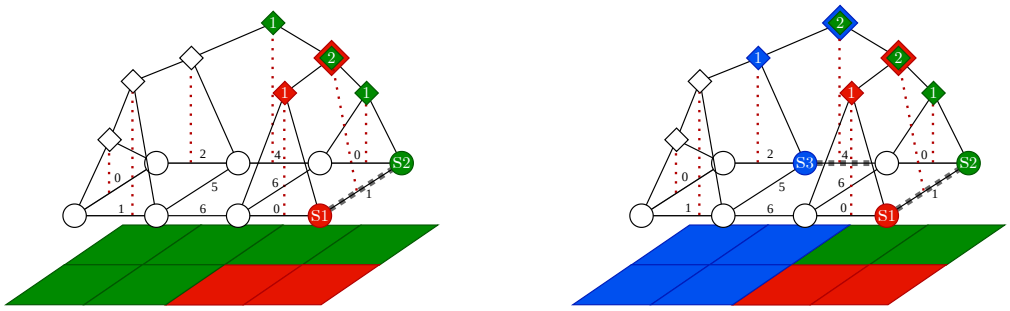


Figure 7.2: Overview of our workflow of the interactive incremental watershed segmentation

from the regions of the current partition (obtained with the previously processed markers). To find this edge, we search for the lowest common ancestor of the new marker and any already processed marker in the BPH. This node corresponds to the lowest edge of the MST that merges a region containing the new marker with a region containing an already processed marker. In order to "prevent" this merging and to cut the region of the new marker from the rest of the partition, the MST edge associated to this node is tagged as a "WS edge" and added to the set of edges to be deleted from the MST.



(a) (b)

Figure 7.3: Integer labels on non-leaf nodes represent the value of `visitCount`. (a) Initialization with S_1 and S_2 as markers. (b) Update after the addition of S_3 as an additional marker.

In order to implement this scheme, we consider an array `visitCount` which tracks, for each node in the BPH, how many times it has been visited during successive marker traversals. Initially, `visitCount` is set to 0 for every node.

Then, for each leaf node of the tree corresponding to a marker, we visit its ancestors by browsing the tree from the leaf to the root, and we update `visitCount` accordingly. Let `p` be an ancestor of the considered marker node found during the traversal. If `visitCount[p]` $= 0$, it is incremented, and the traversal continues upward to `p`'s parent. However, if `visitCount[p]` $= 1$, this indicates that `p` has already been reached by a different marker. In this case, `p` becomes a watershed node. Its corresponding edge in the MST separate two distinct catchment basins.

The value `visitCount[p]` is thus incremented to 2, and the MST edge associated with `p`, given by $\mathcal{H}$`.mstEdge[p]`, is added to the set of watershed edges W, as described in line 7 of Algorithm 25.

Specifically, nodes with `visitCount` $= 2$ account for the watershed nodes; nodes with `visitCount` $= 1$ account for the regions containing at least one marker; and nodes with `visitCount` $= 0$ account for the regions that do not contain any marker.

The traversal halts once a node with `visitCount` $= 2$ is reached, as the separa-

Algorithm 25: ADD MARKERS

Data: $\mathcal{H}$: a BPH, `markers`: a set of markers and `visitCount`;
Result: `ws` a set of edges to be removed and `visitCount` updated.

```
1 ws ← ∅
2 foreach leaf n of markers do
3     while n ≠ H.root and visitCount[n] ≠ 2 do
4         n := H.parent[n]
5         visitCount[n] := visitCount[n] + 1
6         if visitCount[n] = 2 then
7             ws := ws ∪ {H.mstEdge[p]}
```

tion induced by the addition of the new marker has been found. Therefore, adding a new marker only requires visiting the nodes along the path from that marker to its nearest watershed node. This behavior is illustrated in Figure 7.3(b), where the insertion of marker S_3 involves visiting only two nodes. During a call or a succession of calls to Algorithm 25, each node is visited at most twice, leading to an overall linear-time complexity with respect to the number of vertices.

7.2.2 Pixel labeling

Once the set of watershed nodes is computed, their corresponding edges are removed from the MST. The resulting graph is a minimum spanning forest (MSF) relative to the markers. The connected components of this graph are then labeled using a Breadth First Search (BFS) algorithm. This latter is performed in linear time with respect to the number of vertices and edges of the MSF.

7.2.3 Incremental Management of User Interaction

The workflow presented in the previous sections is suited to work in an incremental way, where one considers successive addition or removal of markers and the differential update of the resulting WS cut and intermediary structures. Let us detail the two cases corresponding to the addition and removal of a marker.

Marker Addition. When a new marker is introduced, it induce the creation of a new watershed edge within the initial MST. This identification can be carried out incrementally by calling Algorithm 25, under the condition that the `visitCount` array from the previous state is retained and that only the newly added markers are provided as input.

The algorithm outputs the set of new watershed edges to be removed from the current MSF. Each such edge $\{a, b\}$ split the connected component containing both

edges into two new components: one containing vertex a, and the other containing vertex b. To reflect this separation, a relabeling of the connected components is required. This can be handled by performing a BFS from each endpoint of the removed edge.

Marker Removal. Conversely, when a marker is deleted, one watershed edge must be restored to the MSF. This operation can also be computed incrementally by modifying Algorithm 25 to produce Algorithm 26. Instead of incrementing `visitCount` during the bottom-up traversal, the algorithm decrements it on this path.

During this traversal, for each visited ancestor `p`, the counter `visitCount[p]` is decremented. The traversal stops as soon as `visitCount[p]` $= 1$ after decrementation, indicating that the region previously marked as a watershed is no longer shared between distinct markers. The corresponding edge is then reintroduced into the MSF.

Once reinserted, each restored edge $\{a, b\}$ merges the two previously separated connected components containing a or b into one single connected component. The resulting unified component must be relabeled, which can be achieved using a BFS starting from either endpoint of the reintroduced edge.

Algorithm 26: Remove Markers

Data: $\mathcal{H}$: a BPH, `markers`: a set of markers and `visitCount`;
Result: `ws` a set of edges to be added and `visitCount` updated.

```
1 ws ← ∅
2 foreach leaf n of markers do
3     while n ≠ H.root and visitCount[n] ≠ 1 do
4         n := H.parent[n]
5         visitCount[n] := visitCount[n] − 1
6         if visitCount[n] = 1 then
7             ws := ws ∪ {H.mstEdge[p]}
```

In this section, we addressed the addition and removal of markers without recomputing the entire watershed segmentation from scratch. This is made possible by maintaining three core data structures throughout the interactive session: BPH, the MSF, and the `visitCount` array. Together, these structures allow for the localized updating of watershed edges and the connected components they affect. While we have briefly discussed the relabeling process required to reflect these updates, the next section will explore in more detail how to efficiently relabel connected components by leveraging both the incremental workflow and the structural properties of the MST.

7.3 Strategies for Minimum Spanning Forest Labeling

7.3.1 Breadth-First Search and Heuristics

As mentioned in the previous section, the connected components of the MSF can be labeled using a BFS algorithm. This algorithm is efficient and works in linear time with respect to the number of vertices and edges of the MSF. However, it can be further optimized by optimizing the merging of the connected components. The idea is to memorize the size of the connected components during the labeling. By doing so, when merging two connected component, we spread the label of the larger component to the smaller one hence minimizing the number of visited vertices for each marker removal.

It is not possible to directly transpose this strategy to the case of marker addition, since the size of the connected components is not known in advance. It is possible to maintain a pair associated to each edge indication the number of vertices in the connected components of the two vertices of the edge. This pair is updated during the relabeling process. However, such approach might introduce a significant overhead both in computation and memory. Alternatively, we chose to simply pick randomly one of the two vertices of the edge as starting point of the BFS.

7.3.2 A Parallel Strategy

As demonstrated in the experiments reported in Section 7.4.4, the majority of the computation time required to update a watershed cut segmentation in our differential framework is spent on connected component labeling. This step accounts up to 99.5% of the total execution time in the case of 3D images. To further enhance interaction speed and responsiveness when processing large volumes, we propose a parallel algorithm that reduces the time spent on labeling by leveraging multi-core CPU architectures.

Our method, detailed in Algorithm 27 and Algorithm 28, extends the parallel strategy introduced in [128]. It takes advantage of the fact that the connected components being processed form trees and therefore contain no cycles. This property enables the algorithm to propagate root labels instead of computing distances, thereby simplifying the traversal.

Before diving in the details of the parallel algorithm, we first recall the concept of a level set in a forest. Let the distance $D_{x_0}(X_k)$ from a vertex x_0 to any other vertex x_k be the number k of element on the path $\langle x_0, \ldots, x_k \rangle$ distinct from x_0. The λ-*level-set of* x_0 is the set $\{x \mid D_{x_0}(x) = \lambda\}$ of vertices at distance λ from x_0.

Algorithm 27: Parallel Breadth-First Search

Data: $\mathcal{F}$: a forest, *roots*: a set of root vertices to search from, each of which being in a distinct tree of $\mathcal{F}$, p: the number of processor;
Result: `label`: the updated segmentation.

```
1  E ← roots
2  label[r]:= new unique label
3  while E ≠ ∅ do
4      if |E| > MIN_BREADTH then
5          (E_1, ··· , E_p) := Partition(E, p)
6          foreach processor i in {1, ··· , p} do in parallel
7              S_i:= SuccessorLabeling(F, E_i)
8              foreach ℓ in {1, ... ,PAR_DEPTH} do
9                  S_i:= SuccessorLabeling(F, S_i)
10         E ← ⋃_{i=1}^{i=p} {S_i}
11     else E:= SuccessorLabeling(F, E)
```

The labeling proceeds through successive iterations over level sets rooted at the initial labeled vertices. Each level set, associated with one vertex, is traversed in parallel. Because connected components are trees, no vertex can be reached from multiple neighbors. As a result, no synchronization mechanism is needed when accessing or updating the label array, allowing us to use a standard shared array without conflict. Then, the main problematic is to ensure that the workload is balanced across the available processors. To achieve this, prior to this parallel step (Algorithm 27, line 6), the current level set is partitioned (line 5) so that each processor receive the same amount of vertices to process.

Algorithm 28: Successor Labeling

Data: $\mathcal{F}$: a forest, E: a set of vertices to explore;
Result: S: the set of explored vertices

```
1  S ← ∅
2  foreach vertex v in E do
3      foreach vertex w adjacent to v in F do
4          if label[w] ≠ label[v] then
5              S := S ∪ {w}
6              label[w] := label[v]
```

Additionally, we introduce two parameters to fine-tune the degree of paral-

lelization and the frequency of synchronization. The first parameter, `MIN_BREADTH`, enables the algorithm to dynamically switch between sequential and parallel execution depending on the size of the current level set, which corresponds to the active propagation front. The second parameter, `PAR_DEPTH`, controls how many propagation steps are performed in parallel before synchronizing the processors (the union step at line 10). This parameter stand from the range $\lambda \in [1, \ldots, \texttt{PAR_DEPTH}]$ of λ-*level-sets* to consider. While synchronization is necessary to ensure a fair workload distribution, it incurs a computational cost. Completely avoiding synchronization may result in unbalanced partitioning and a drop in parallel efficiency. These parameters allow us to strike a balance between minimizing synchronization overhead and maintaining effective parallelism.

Additionally, we introduce two parameters to fine-tune the degree of parallelization and the frequency of synchronization. The first parameter, `MIN_BREADTH`, enables the algorithm to dynamically switch between sequential and parallel execution depending on the size of the current level set, which corresponds to the active propagation front. The second parameter, `PAR_DEPTH`, controls how many propagation steps are performed in parallel before synchronizing the processors (the union step at line 10). This parameter defines the range $\lambda \in [1, \texttt{PAR_DEPTH}]$ of λ-*level sets* to process between two synchronizations. While synchronization is necessary to ensure fair workload distribution, it introduces additional computational overhead. Thus, if the level set is not important enough, this overhead might not be worth the effort, and the algorithm can be run sequentially. On the other hand, avoiding synchronization entirely may lead to unbalanced partitioning and degraded parallel efficiency.

These parameters are therefore key to optimizing the relabeling step: they must be carefully tuned to strike the right compromise between minimizing synchronization costs and preserving a balanced workload across processors, which is essential for achieving both high performance and scalability in large-scale image segmentation tasks. Such parameter tuning will be discussed in Section 7.5.

7.4 Experimental Results

7.4.1 Tested Methods

In addition to our proposed methods, we evaluated five state-of-the-art implementations of seeded watershed segmentation:

1. **OpenCV** [72, 18]: a highly optimized image processing library implementing a traditional seeded watershed algorithm.

Method	IWS	NIWS	OpenCV	DIFT	HIGRA	ITK	Amira
Pre-processing	BPH	BPH	✗	Parent Array	✗	✗	✗
Incremental	✓	✗	✗	✓	✗	✗	✗
Parallel	✓	✓	✗	✗	✗	✗	✗
WS paradigm	Drop Of water	Drop Of Water	Flooding	IFT	Drop of Water	Flooding	Flooding
Algorithm type	Kruskal	Kruskal	Region Growing	Dijkstra	Kruskal	Region Growing	Region Growing
Union-find	✓	✓	✗	✗	✓	✗	✗
Priority queue	✗	✗	✓	✓	✗	✓	✓
Sorting	✓	✓	✗	✗	✓	✗	✗
2D experiences	✓	✓	✓	✓	✓	✗	✗
3D experience	✓	✓	✗	✗	✗	✓	✓

Table 7.1: Overview of the main features of the tested watershed implementation sand their inclusion in the presented experiments.

2. **HIGRA** [91]: a generic framework for hierarchical graph analysis that includes a watershed method based on the watershed cut approach from [26], adapted to compute a MSF rooted in markers instead of minima.

3. **DIFT** [35]: an incremental version of the IFT-based watershed. We used the implementation provided by the authors[2].

4. **ITK** [10]: the Insight Toolkit, a widely adopted library for medical image processing, which provides an implementation of the marker-based watershed algorithm.

5. **Amira** [113]: a commercial software suite developed by Thermo Fisher Scientific, used in life science imaging for visualization and analysis.

We also evaluated three variants of our proposed method:

6. **NIWS**: a non-incremental variant where the BPH is computed once, and for each user interaction, the watershed edges are recomputed using only Algorithm 25, followed by a full relabeling of connected components.

7. **IWS**: an incremental version that updates the `visitCount` array and the labeling after each interaction by processing only the modified markers (Algorithms 25 and 26).

8. **IWS_PAR**: a parallelized extension of IWS, where connected component labeling is performed concurrently, following the strategy detailed in Section 7.3.2.

[2] `https://github.com/tvspina/ift-demo`

Execution times were measured on an Intel Xeon W7-2495X[3] processor with 24 cores and 48 threads at 4.8GHz, running GNU Linux. Each time is reported as the average over five runs.

The experiment in Section 7.4.4 was conducted using 11 threads, with parameters `PAR_DEPTH` and `MIN_BREADTH` set to 3 and 165, respectively.

7.4.2 Experiment with user generated seeds

In this first experiment, a user was asked to segment four high-resolution images of size 2048×1536 pixels. Three of the images were selected from the INRIA Holidays dataset [49], while the fourth was provided by ourself (being our coleus plant). The segmentation task was performed using a graphical interface that allowed the user to interactively edit markers. Green markers were used to indicate the object of interest, and red markers were used to indicate the background. In case of mistakes, an eraser tool enabled the user to remove existing markers. Each interaction was logged as a batch of added or removed green and red markers.

Method	Init	Average	Worst	Accumulated
IWS	200.64	<u>6.73</u>	<u>54.26</u>	<u>398.81</u>
NIWS	200.64	55.20	57.83	2091.46
DIFT	<u>12.17</u>	266.36	317.33	8997.28
OpenCV	$\emptyset$	133.47	142.64	4518.08
HIGRA	$\emptyset$	388.95	402.59	12572.92

Table 7.2: Computation time (in milliseconds) of the different methods with user generated seeds. The second column indicates the initialization time, while the third and fourth columns show the average and worst computation time for all user interactions. The last column represents the total computation time, which includes both the initialization and the sum of all user interactions

The execution times are reported in Figure 7.4 and in Table 7.2. We observe that the initialization cost, corresponding to the BPH construction, is quickly amortized over the course of the segmentation process for both NIWS and IWS. These two versions of our method consistently achieve significantly lower average execution times compared to the other approaches.

Moreover, the execution time of IWS is proportional to the number of pixels affected by marker updates, in line with the theoretical expectations. The highest execution time typically occurs during the first user interaction, when all pixels

[3]For licensing reasons, Amira was run on a different system, an Intel Xeon W-2245 with 8 cores and 16 thread at 4.7GHz.

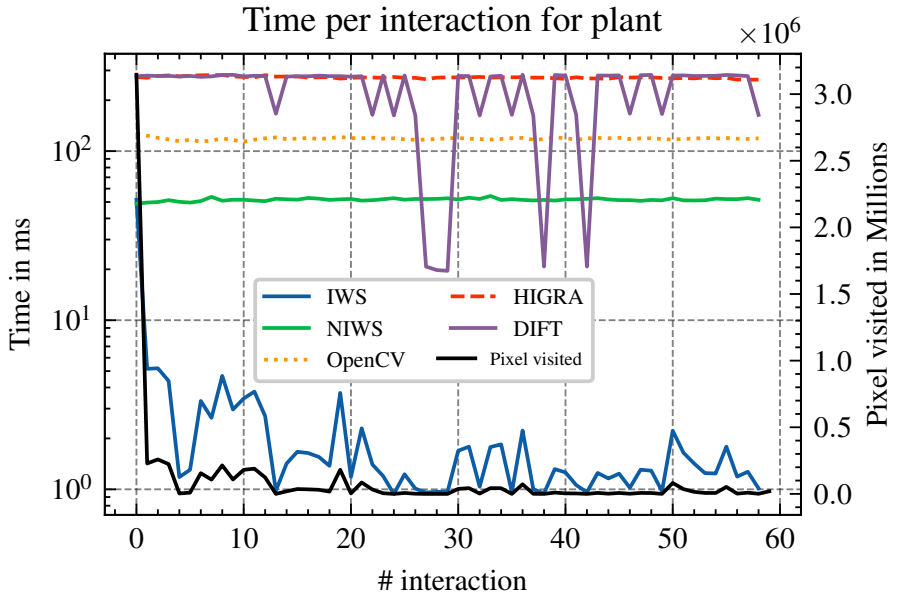


Figure 7.4: Computation time in ms during an interactive session (user generated markers). The plain black curves show the number of pixels updated at each user interaction (right axis).

are labeled for the first time. This initial step is therefore equivalent to the non-incremental NIWS variant. Additional spikes in computation time may appear mid-process or toward the end, particularly when a user modifies markers implying changes within a large connected component, which triggers a substantial number of pixel updates.

Finally, our results show no significant difference in execution time between the addition and removal of markers.

7.4.3 Experiment with randomly generated seeds

To evaluate the methods on a larger and more diverse dataset, we selected the BIG dataset introduced in [22], which consists of high-resolution natural images ranging in size from 2048×1600 to 5000×3600 pixels. We used the full set of 150 images and associated each image with a sequence of 70 markers. These markers are evenly split between two classes: object and background. Each marker is defined as a disk of radius 11, centered on a randomly selected pixel from the eroded version of the object (or background) mask, where erosion is performed using a disk of radius 12. This ensures that each marker is well contained within its respective class.

The experimental protocol follows the same incremental strategy described in

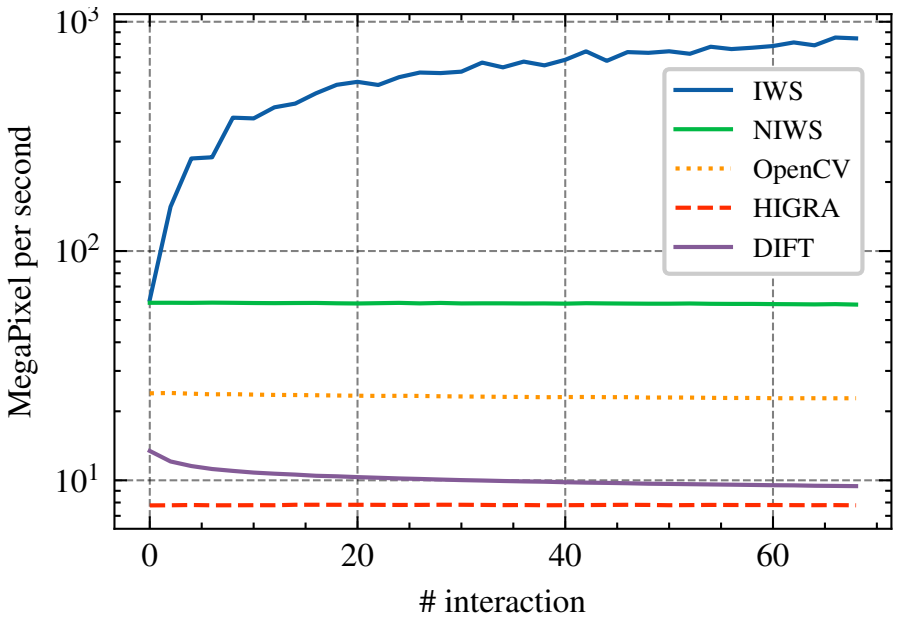


Figure 7.5: Average throughput in megapixel per second on the BIG dataset with simulated interactions. In order to avoid noisy visualization, the curve is smoothed by averaging values in a window of size 2.

Section 7.4.2. Markers are introduced one after the other, alternating between object and background, and only additions are considered throughout the process. Marker removal is not included in this evaluation.

Method	Init	Average	Worst	Accumulated
IWS	14.39	596.17	58.21	2.87
NIWS	14.39	59.09	53.06	0.80
DIFT	233.79	10.20	9.22	0.14
OpenCV	∅	23.23	21.07	0.33
HIGRA	∅	7.80	7.21	0.11

Table 7.3: Throughput in megapixel per second averaged on all images of the BIG dataset with simulated interactions.

The results are shown in Figure 7.5 and Table 7.3. For each interaction, the throughput is averaged across all 150 images. The trend observed in Section 7.4.2 is confirmed: while the throughput of other methods remains constant across iterations, our incremental method exhibits a clear increase in throughput as the process progresses. This is explained by the decreasing number of pixels affected by each interaction.

Additionally, the results demonstrate that our incremental approach significantly reduces response time compared to other methods, leading to a more efficient and responsive user experience during interactive segmentation.

7.4.4 Experiment on 3D images

To evaluate the method on larger volumetric data, we used the liver segmentation dataset from [111], which contains 3D images acquired via CT scans. From this dataset, we selected five volumes with dimensions ranging from $512 \times 512 \times 124$ to $512 \times 512 \times 260$ voxels. The experimental protocol follows the methodology described in Section 7.4.3, using the provided liver masks as ground truth. For each volume, we generated 60 sets of markers, with each set consisting of 30 randomly positioned markers covering 100 pixels each.

Since our algorithm is designed to operate on arbitrary graphs, it does not require any modification to handle 3D data, and can process volumetric images directly.

Execution times are reported in Figure 7.6 and Table 7.4. The trends observed in Sections 7.4.3 and 7.4.2 are confirmed: the throughput of ITK, NIWS, and Amira remains constant throughout the interactions, while the two differential approaches, IWS and IWS_PAR, show a significant improvement in responsiveness. In particular, their throughput increases after the initial iterations, as new markers lead to smaller refinements in the segmentation and thus fewer pixel-level changes.

On average, across all evaluated 3D images, the per-interaction execution times for ITK, NIWS, IWS, IWS_PAR, and Amira are 9.26, 2.48, 1.18, 0.34, and 6.80 seconds respectively. This results in a speedup factor of 27 when comparing the fastest method, IWS_PAR, to the ITK baseline. Furthermore, parallelization in IWS_PAR yields a 3.47-fold speedup over the standard IWS method, excluding initialization time.

Method	Init	Average	Worst	Accumulated
IWS	7.92	86.18	26.78	0.54
NIWS	7.92	24.72	23.71	0.32
IWS_PAR	7.34	216.78	96.14	0.89
Amira[3]	∅	8.14	7.38	0.13
ITK	∅	6.49	6.22	0.11

Table 7.4: Throughput in megapixel per second on all 5 3D images with simulated interactions.

To better understand the computational behavior of each method, we analyzed the proportion of time spent in the main steps across all iterations. For NIWS,

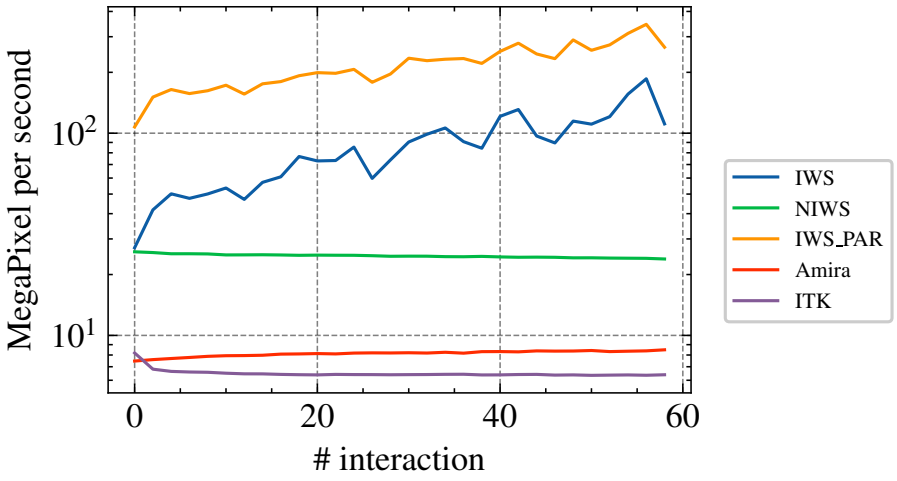


Figure 7.6: Throughput in megapixel per second on 3D images with simulated interactions. An average smoothing with a window size of 2 is used to prevent noisy aspects.

95.7% of the total execution time is dedicated to pixel labeling, while node marking and unmarking account for 0.66%, and BPH construction takes 3.64%. In the case of IWS, pixel labeling remains the dominant operation, representing 92.3% of the execution time, with 0.5% spent on node updates and 7.2% on BPH construction. For IWS_PAR, which benefits from parallelization, 82.8% of the time is spent on pixel labeling, 1.03% on node updates, and 16.17% on building the BPH.

If we focus on a single interaction step, the breakdown changes. For NIWS, the proportion of time allocated to each task remains essentially the same, as the BPH is reconstructed at each step. In contrast, for IWS and IWS_PAR, the hierarchy is built only once at initialization and reused, effectively eliminating the cost of BPH construction during subsequent interactions. In this context, over 99.5% of the time is spent on connected component labeling for both incremental methods, highlighting the importance of optimizing this step. This observation supports our decision to introduce a parallel labeling strategy, as detailed in Section 7.3.2.

7.5 Assessment of Parallel Breadth-First Search

7.5.1 Parameter Tuning

Figure 7.7 presents the execution times of the Parallel Breadth-First Search algorithm as both the `MIN_BREADTH` parameter and the number of threads are varied within the ranges $[0; 3000]$ and $[1; 48]$, respectively. Each data point corresponds to the average execution time of the algorithm across all 60 interactions on one of

the 3D images from the liver segmentation dataset described in Section 7.4.4.

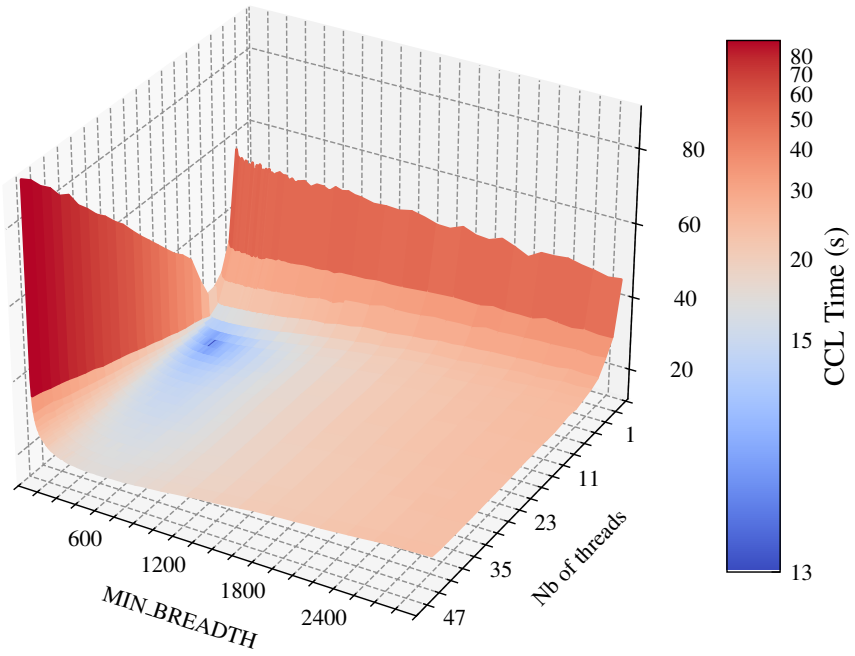


Figure 7.7: Connected component labeling time for an interactive session on patient2 with respect to the parameter `MIN_BREADTH` and to the number of threads

From the plot, we identify a clear global minimum corresponding to 11 threads and a `MIN_BREADTH` value of 165. These optimal values were adopted in the experiments of Section 7.4.4 for comparing the performance of IWS_PAR against other methods.

The plot also reveals two key trends:

1. For a fixed number of threads, increasing the `MIN_BREADTH` value initially leads to a sharp decrease in execution time, which reaches a minimum before gradually increasing again. This behavior reflects the balance between the overhead of parallelism on small workloads and the benefit of parallel processing on sufficiently large propagation fronts.

2. For a fixed `MIN_BREADTH` value, increasing the number of threads results in a reduction in execution time up to a certain point, beyond which the overhead introduced by thread management and synchronization begins to outweigh the gains from parallelism, causing the execution time to increase.

The first observation confirms that parallel traversal of small level sets leads to performance degradation compared to a sequential approach. This justifies the introduction of the `MIN_BREADTH` parameter, which enables the Parallel Breadth-First Search algorithm to switch from sequential to parallel execution only when the current level set contains more than `MIN_BREADTH` elements. This adaptive mechanism ensures that parallelism is applied only when the workload is sufficient to justify its overhead.

The second observation clearly demonstrates the efficiency of the proposed parallelization strategy, but also its limitations. As the number of threads increases beyond a certain point, the number of vertices assigned to each thread (that is, the number of elements in the set S_i in Algorithm 27) becomes too small relative to the overhead required to manage each thread. This imbalance leads to reduced performance. In the most extreme case, some threads may receive an empty subset, resulting in unnecessary invocation of the `SuccessorLabeling` procedure on $S_i = \emptyset$.

When the optimal parameters are applied, the proposed parallel method achieves a speedup of 3.63 over the sequential version. Figure 7.8a illustrates the speedup as a function of the number of threads, with `MIN_BREADTH` fixed at 165.

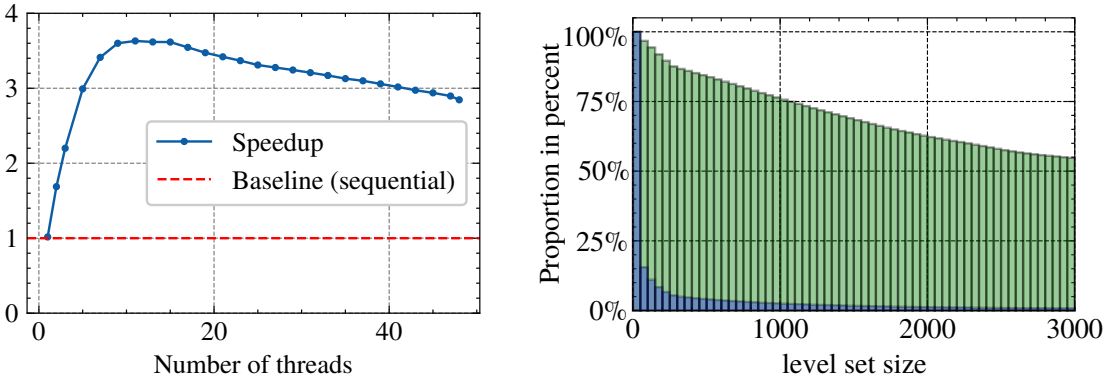
7.5.2 Discussion on Parallelization Rate

The optimal speedup of 3.63 is reached using 11 threads, corresponding to a parallel efficiency of approximately 33%. To further analyze the degree of parallelization achieved by the method, Figure 7.8b presents two complementary metrics as functions of the `MIN_BREADTH` parameter:

1. The proportion of level sets that are processed in parallel (blue curve); and
2. The proportion of vertices labeled during those parallel traversals (green curve).

With the optimal value of `MIN_BREADTH`, only 10% of the level sets are processed in parallel. However, these few iterations account for 93% of the total labeled vertices. This result highlights that most of the computational effort is concentrated in a small number of large level sets, which are efficiently handled in parallel. It also confirms the relevance of our adaptive strategy: enabling parallelism selectively where it has the highest impact ensures that resources are used effectively without incurring the overhead of parallel execution on smaller, less demanding workloads.

We also analyzed the time distribution across the various steps of the algorithm, identifying the proportion of total execution time spent in each phase:



(a) (b)

Figure 7.8: (a) Speedup of parallel over sequential breadth-first search. (b) Proportion of parallel iterations (blue) and of vertices traversed in a parallel iteration (green) with Parallel Breadth-First Search with respect to `MIN_BREATH` values.

1. **Sequential time (14.5%)**: time spent performing sequential exploration of level sets, as described in Algorithm 27, line 11.
2. **Distribution time (2%)**: time dedicated to partitioning the current level set among threads, corresponding to line 5 in Algorithm 27.
3. **Parallel time (33%)**: time used for parallel exploration of the level set, which takes place from lines 7 to 9 in Algorithm 27.
4. **Union time (8%)**: time spent merging the results from all threads after the parallel traversal, corresponding to the union step at line 10.
5. **Wait time (9%)**: time during which some threads remain idle while waiting for others to complete their parallel tasks.
6. **System time (33.5%)**: time consumed by the operating system for thread management and synchronization once all threads have completed their parallel exploration.

7.6 Conclusion

In this chapter, we introduced a novel interactive marker-based watershed segmentation method designed to operate incrementally. By leveraging the causal nature of user interactions, this method significantly improves responsiveness and reduces redundant computation, particularly when applied to large-scale data such as 3D medical images.

To further enhance performance, especially during the initial interactions when the segmentation is still coarse, we proposed a parallel labeling algorithm tailored for minimum spanning forests. This method exploits the acyclic structure of connected components to safely distribute labeling tasks across multiple threads. With appropriate tuning of parallelization parameters, the proposed approach achieves up to 95-fold speedup in cumulative execution time compared to high-performance baseline implementations.

The effectiveness of the parallel algorithm is influenced by the choice of the parameters `MIN_BREADTH` and `PAR_DEPTH`, which control the trade-off between parallel workload distribution and synchronization overhead. In this work, we provided a detailed empirical analysis of these parameters and demonstrated that a carefully selected static configuration leads to substantial performance gains. This analysis not only validates the robustness of the proposed parallel scheme but also offers practical guidance for applying it to real-world data. Building on this foundation, future work will explore strategies for dynamically adapting parameter values during execution to further improve responsiveness and efficiency throughout the entire interactive segmentation process.

Chapter 8

Conclusion

In this thesis, we have presented an out-of-core framework for hierarchical segmentation, with a particular focus on the computation and distribution of the Binary Partition Hierarchy (BPH). Designed to overcome the limitations of main memory when dealing with large-scale image data, our contributions span both theoretical foundations and practical algorithms, enabling efficient and scalable hierarchical image analysis.

We began by introducing two high-level calculi for computing the distribution of the BPH, as detailed in Chapter 4, which models segmentation hierarchies as collections of subtrees whose size is manageable by a single machine. The first calculus is applicable to any spatial partitioning of the data, while the second is optimized for causal partitions. While being efficient in practice, the first one does not satisfy the out-of-core constraint from a theoretical point of view, while we gave properties ensuring that the second one does. These calculi rely on three algebraic operations namely *select*, *join*, and *insert*. Their computational complexity is either linear or linearithmic. Our experiments show that while the first calculus may lack strict out-of-core guarantees, it remains practical for images considering a moderate number of slices and is faster than the second approach. These results laid the groundwork for building a toolkit of out-of-core hierarchical segmentation algorithms.

The next major contribution given in Chapter 5, is the introduction of the Propagate algorithm, a generic approach to compute attributes on a precomputed distribution in an out-of-core context. An attribute is associated with each subtree of the distribution such that filtering a subtree can be done independently of the others. The algorithm operates on causal partitions with only two adjacent tiles in memory, allowing linear-time processing. We showed how this approach can be used to compute classical attributes such as area, volume, and local minima, and highlighted its extensibility to other features such as bounding boxes or mean intensities. Notably, we demonstrated how to adapt Propagate to compute both

marker-based and hierarchical watershed segmentations out-of-core, further consolidating its role as a foundational component of our framework. While this work focused on causal partitions, we also discussed how the method could, in principle, be extended to generic partitions through the use of the Global Border Tree, a direction we identify for future exploration.

From a more applicative point of view, in Chapter 6 we proposed a practical solution for marker-based watershed segmentation using superpixels based on α-flat zones. This technique enables an exact watershed to be computed over a simplified region adjacency graph, provided that markers fall within distinct α-flat zones. We introduced a full pipeline for constructing the region adjacency graph distribution and validated its effectiveness experimentally. The results demonstrate a significant reduction in peak memory usage, down to three times the image size with minimal performance overhead. Furthermore, we studied the impact on the parameter α to control the granularity of α-flat zones, offering a flexible trade-off between segmentation resolution and resource consumption. This approach represents an important advance toward truly scalable and controllable watershed segmentation on very large volumes.

Moving away from memory constraint considerations to focus on execution time, we also proposed an incremental marker-based watershed segmentation algorithm tailored for in-core interactive use cases, particularly in 3D imaging. This method, described in Chapter 7, takes advantage of the causality of user interaction ensuring that the computation time associated with a marker addition or deletion is proportional to the size of the region to relabel. It incorporates a parallel labeling algorithm based on minimum spanning forests to further accelerate processing. Our experiments showed that, with appropriate tuning, this approach can achieve up to 95 fold speedups over high-performance baselines. This line of work demonstrates the versatility of the hierarchical framework and its applicability to interactive, user-driven workflows.

Taken together, the proposed out-of-core hierarchical analysis framework form a coherent and extensible framework for scalable hierarchical image segmentation in out-of-core environments. Still, several open directions remains.

Further divide to conquer

To achieve memory consumption levels that are acceptable for real-world applications, our framework currently relies on the use of superpixels. This strategy effectively reduces the complexity of the data by simplifying the image into a coarser representation. In practice, this means that we deliberately "drop" part of the data in order to maintain the process within memory constraints. While we have shown that this simplification introduces only a mild restriction on markers locations and does not hinder the correctness of out-of-core watershed segmen-

tation, a natural question arises: What fundamentally limits the ability of our framework to operate efficiently at full resolution?

The answer lies in the redundancy of high-level regions that are highly replicated throughout the hierarchy distribution. As the spatial extent of a region increases, so does its likelihood of overlapping multiple chunks. Since each chunk maintains its own local hierarchy, these large-scale structures are duplicated across many local trees, leading to significant memory overhead and reduced efficiency. This issue stems from a deeper structural limitation in our current chunking strategy.

Our current approach follows a rigid spatial decomposition strategy. We begin by dividing the image into spatial chunks before constructing one complete hierarchy per individual chunk so that each local structure contains the necessary context for segmentation, from leaves up to the root. This "*vertical*" chunking strategy, while conceptually straightforward, leads to significant drawbacks. Regions that span multiple chunks are duplicated across many hierarchies, resulting in widespread replication of nodes and substantial memory overhead.

A more flexible alternative would involve a "*horizontal*" approach that builds hierarchies incrementally, scale by scale. Instead of computing full hierarchies for each chunk directly, we would construct them progressively, processing one range of scales at a time, from the leaves up to the root. Specifically, considering the complete range of scales $\{1, \ldots, i, \ldots, j, \ldots, k\}$, we could built a first distribution for the range $\{1, \ldots, i\}$. Next, we would proceed to the next range $\{i+1, \ldots, j\}$ on the basis largest regions previously computed, and so forth, until reaching the root scale k. This method would ensure that at each step, we only maintain the necessary nodes for the current scale range, significantly reducing redundancy.

This strategy would significantly reduce the number of duplicated nodes at each step and could enhance the scalability of our framework by avoiding the need to store complete hierarchies simultaneously across all chunks.

Attributes

While we have proposed a generic methodology for computing attributes in an out-of-core setting, one cornerstone of hierarchical analysis remains absent: the computation of the lowest common ancestor (LCA). The LCA plays a central role in many hierarchical operations, as it captures the smallest region enclosing any pairs of regions and it gives the ultrametric distance between any two nodes.

Integrating an efficient out-of-core LCA computation would extend the analytical capabilities of our framework to other hierarchical representations and important geometric attributes. In particular, it would enable the computation of saliency maps. This concept introduced by [80] is also known as ultrametric contour maps [5, 6]. This map associate with each contour of a region in a hierarchy,

the scale at with the region disappears. Taking the specific case of hierarchical watershed, the saliency will correspond to the altitude of the saddle point between two catchment basins. These maps, closely related to quasi-flat zones hierarchies are often derived from the LCA.

Beyond saliency maps, the LCA can also be used to compute other geometric attributes such as perimeter, circularity or compactness. These features often depend not only on a region's immediate geometry but also on how it merges with its neighbors in the hierarchy. Which is why the LCA is crucial for capturing these relationships.

By integrating an efficient out-of-core LCA computation, we would significantly enhance the analytical capabilities of our framework toward the objective of large-scale, flexible, and rich segmentation pipelines.

Incremental Out-of-Core Marker-Based Watershed

We proposed an out-of-core (hierarchical) watershed pipeline which allows for an exact computation of these segmentations. However, a current limitation of our approach is its non-incremental nature. Each time a new marker is added, the entire distribution of the attribute must be recomputed from scratch. This is acceptable for hierarchical watershed and minima-based watershed, as it mimics the in-core approach. However, it lacks flexibility when dealing with marker-based watershed. While this design is acceptable in the context of hierarchical watershed and watershed from minima, it proves to be less suitable for marker-based watershed, where interactivity and responsiveness are key.

To address this limitation, we should aim to adapt the incremental in-core watershed strategy, introduced in Chapter 7, to the out-of-core setting. In the in-core version, the watershed is computed incrementally as new markers are added, allowing for immediate feedback and efficient exploration of segmentation results. Extending this paradigm to the out-of-core domain would require the initial distribution of markers to be computed once and updated incrementally, in a manner that respects the memory and locality constraints of our framework.

Then two main challenges arise. First, when a new marker is added or removed, the tree values are updated to reflect the new marker's influence. Regions of the hierarchy associated to the chunk containing the new marker must be updated accordingly, while ensuring that occurrences of the same regions in different hierarchies are kept consistent. This requires a mechanism to propagate these changes only to the trees that are affected by the new marker, rather than propagating changes across the complete distribution.

Second, the pixel labeling stage must be adapted to handle incremental updates even though a single minimum spanning forest can span over multiple chunks. When new markers are added, the labeled regions must be updated accordingly,

while ensuring consistency across chunk boundaries. This introduces the need for a robust out-of-core minimum spanning forest connected component labeling algorithm, capable of updating region labels progressively without full recomputation. Parallel or distributed existing algorithms should be adapted to work in this out-of-core setting.

Developing this capability would bring our out-of-core framework closer to a truly interactive and scalable segmentation system, where both hierarchical and marker-based approaches coexist efficiently.

Parallel and distributed computing

Another promising direction for future work is to explore the parallel and distributed potential of our framework. This work was initially inspired by existing strategies for distributed computation of component trees, which we have extended to the out-of-core computation of the Binary Partition Hierarchy. In our current implementation, memory constraints require that image chunks be processed sequentially. However, the underlying operations in our framework are inherently parallelizable. Each partial hierarchy can be computed independently for its corresponding chunk, and the merging of neighboring hierarchies can also be conducted concurrently.

To the best of our knowledge, no existing algorithm has been proposed for computing the BPH in a parallel or distributed setting within the domain of mathematical morphology. Interestingly, such strategies do exist in the context of single linkage clustering, which is theoretically equivalent to the BPH. Notable examples include the recent works by Dhulipala et al. [33] for parallelism and Sao et al. [104] targeting massively parallel architectures. It is interesting to note that contrary to our framework, these approaches take as input minimum spanning trees. One major difference between single linkage clustering and mathematical morphology communities is that while the first is interested in dense graph, the second is interested into images model as regular sparse graphs.

This distinction highlights a unique opportunity: adapting our out-of-core framework for parallel or distributed computation without relying on prior MST computation. Doing so would address the specific needs of morphological image analysis and segmentation, where the spatial regularity of the image graph structure can be leveraged for more efficient processing.

Regarding other popular hierarchical representations, the proposed *join* algorithm for merging two hierarchies has the distinctive property that its complexity is independent of the image dynamic, unlike existing parallel and distributed algorithms. This makes it potentially suitable for high dynamic range scenarios. Given the strong relationship between the BPH and quasi-flat zone hierarchies, the algorithm could also be adapted to merge two quasi-flat zone hierarchies. However,

extending it to component trees is less straightforward and would require further investigation.

Appendix: Proofs

8.1 Proof of Property 4.4.1

Before proving Property 4.4.1, let us first introduce some lemmas.

Lemma 8.1.1. *Let A and B be two disjoint subsets of V and let A' and B' be two subsets of A and B, respectively, then we have:*

$$\begin{aligned} &\mathit{select}\,(A' \cup B', \mathcal{B}_A^{\prec} \sqcup \mathcal{B}_B^{\prec}) \\ &= \mathit{select}\,(A', \mathcal{B}_A^{\prec}) \sqcup \mathit{select}\,(B', \mathcal{B}_B^{\prec}) \end{aligned}$$

Lemma 8.1.2. *Let A and B be two disjoint subsets of V and Z be a set such as $Z \cap A \neq \emptyset$ and $Z \cap B \neq \emptyset$. Then we have:*
$\mathit{select}\,(Z, \mathit{join}\,(\mathcal{B}_A^{\prec},\ \mathcal{B}_B^{\prec})) = \mathit{join}\,(\mathit{select}\,(Z \cap A, \mathcal{B}_A^{\prec}),\ \mathit{select}\,(Z \cap B, \mathcal{B}_B^{\prec}))$

Proof of Lemma 8.1.2. Let $\mathcal{H} = \text{select}\,(Z, \text{join}\,(\mathcal{B}_A^{\prec},\ \mathcal{B}_B^{\prec}))$. We are going to prove that $\mathcal{H} = \text{join}\,(\text{select}\,(Z \cap A, \mathcal{B}_A^{\prec}),\ \text{select}\,(Z \cap B, \mathcal{B}_B^{\prec}))$. We have

$$
\begin{aligned}
\mathcal{H} &= \text{select}(Z, \mathcal{B}_A^{\prec} \sqcup \mathcal{B}_B^{\prec} \boxplus \gamma(A,B)) \\
&\quad \text{by definition of join} \\
&= \text{select}(Z, \text{insert}(\text{select}(Z, \mathcal{B}_A^{\prec} \sqcup \mathcal{B}_B^{\prec} \boxplus \gamma(A,B)), \\
&\quad \mathcal{B}_A^{\prec} \sqcup \mathcal{B}_B^{\prec})) \\
&\quad \text{by Lemma 13 of [30]} \\
&= \text{insert}(\text{select}(Z, \mathcal{B}_A^{\prec} \sqcup \mathcal{B}_B^{\prec} \boxplus \gamma(A,B)), \\
&\quad \text{select}(Z, \mathcal{B}_A^{\prec} \sqcup \mathcal{B}_B^{\prec})) \\
&\quad \text{by Lemma 12 of [30]} \\
&= \text{select}(Z, \mathcal{B}_A^{\prec} \sqcup \mathcal{B}_B^{\prec}) \boxplus \gamma(A,B) \\
&\quad \sqcup \text{select}(Z, \mathcal{B}_A^{\prec} \sqcup \mathcal{B}_B^{\prec})
\end{aligned}
$$

From Property 7.3 of [30], we have

$$
\begin{aligned}
\mathcal{H} = &\text{select}\,(Z, \mathcal{B}_A^{\prec} \sqcup \mathcal{B}_B^{\prec}) \sqcup \text{select}\,(Z, \mathcal{B}_A^{\prec} \sqcup \mathcal{B}_B^{\prec}) \\
&\boxplus \gamma(A,B)
\end{aligned}
$$

Since $Z \cap A \subseteq A$ and $Z \cap B \subseteq B$, by Lemma 8.1.1 we deduce that:

$$
\begin{aligned}
\mathcal{H} &= \text{select}\,(Z \cap A, \mathcal{B}_A^{\prec}) \sqcup \text{select}\,(Z \cap B, \mathcal{B}_B^{\prec}) \\
&\quad \boxplus \gamma(A,B) \\
&= \text{join}\,(\text{select}\,(Z \cap A, \mathcal{B}_A^{\prec}),\ \text{select}\,(Z \cap B, \mathcal{B}_B^{\prec})) \\
&\quad \text{by definition of join}
\end{aligned}
$$

□

Proof of Property 4.4.1. At initialization step of Algorithm 1, we have $\mathcal{M}_0 = \text{select}\left(\gamma^{\bullet}_{V \setminus S_0}(S_0), \mathcal{B}_{S_0}^{\prec}\right)$. Then at the first iteration we have

$$
\begin{aligned}
\mathcal{M}_1 &= \text{join}\left(\text{select}\left(\gamma^{\bullet}_{V\setminus S_1}(S_1), \mathcal{B}^{\prec}_{S_1}\right)\right) \\
&\qquad \left(\text{select}\left(\gamma^{\bullet}_{V\setminus S_0}(S_0), \mathcal{B}^{\prec}_{S_0}\right)\right) \\
&= \text{select}(\gamma^{\bullet}_{V\setminus S_0}(S_0) \cup \gamma^{\bullet}_{V\setminus S_1}(S_1), \\
&\qquad \text{join}\left(\mathcal{B}^{\prec}_{S_0},\ \mathcal{B}^{\prec}_{S_1}\right)) \\
&\qquad \text{by Lemma 8.1.2} \\
&= \text{select}\left(\gamma^{\bullet}_{V\setminus S_0}(S_0) \cup \gamma^{\bullet}_{V\setminus S_1}(S_1), \mathcal{B}^{\prec}_{S_0\cup S_1}\right) \\
&\qquad \text{by Property 9 of [30]}
\end{aligned}
$$

By recurrence, we have

$$
\begin{aligned}
\mathcal{M}_i &= \text{select}\left(F, \mathcal{B}^{\prec}_{\cup\{S_j | j\in\{0,\dots i\}\}}\right) \\
\text{with} \quad F &= \bigcup_{j=0}^{i} \left\{\gamma^{\bullet}_{V\setminus S_j}(S_j)\right\}
\end{aligned}
$$

Then, if we consider a partition into $k+1$ regions, when $i = k$ we have

$$
\begin{aligned}
\mathcal{M}_k &= \text{select}\left(F, \mathcal{B}^{\prec}_{V}\right) \\
\text{with} \quad F &= \bigcup_{j=0}^{k} \left\{\gamma^{\bullet}_{V\setminus S_j}(S_j)\right\}
\end{aligned}
$$

This latter rewriting fits Definition 4.4.1. □

8.2 Proof of Lemma 4.4.1

Proof of statement 1. At each step of the first loop of Algorithm 3, we have $\mathcal{M}^{\uparrow} = \text{join}\left(\text{select}\left(\gamma^{\bullet}_{i}(S_{i-1}), B^{\uparrow}_{i-1}\right),\ \text{select}\left(\gamma^{\bullet}_{i-1}(S_i), B^{\prec}_{S_i}\right)\right)$. Theorem 17 of [30] states that $B^{\uparrow}_{i-1} = \text{select}\left(S_i, \mathcal{B}^{\prec}_{\cup\{S_j | j\in\{0,\dots i\}\}}\right)$. Then, since $\delta^{\bullet}(\gamma(S_{i\text{-}1}, S_i)) \cap \gamma^{\bullet}_{i}(S_{i-1}) \subseteq \gamma^{\bullet}_{i}(S_{i-1})$ and $\delta^{\bullet}(\gamma(S_{i\text{-}1}, S_i)) \cap \gamma^{\bullet}_{i-1}(S_i) \subseteq \gamma^{\bullet}_{i-1}(S_i)$, by Lemma 8.1.2 we can deduce that $\mathcal{M}^{\uparrow} = \text{select}\left(\delta^{\bullet}(\gamma(S_{i\text{-}1}, S_i)), \mathcal{B}^{\prec}_{\cup\{S_j | j\in\{0,\dots i\}\}}\right)$ □

Proof of statement 2. Let A, B and C be three pairwise disjoint subsets of V such that $\gamma(A, C) = \emptyset$. Let $\mathcal{M}^{\uparrow} = \text{join}\left(\text{select}\left(\gamma^{\bullet}_{B}(A), \mathcal{B}^{\prec}_{A}\right),\ \text{select}\left(\gamma^{\bullet}_{B}(A), \mathcal{B}^{\prec}_{A}\right)\right)$ and $\mathcal{M}^{\downarrow} = \text{insert}(\text{select}\left(\gamma^{\bullet}_{B}(A), \mathcal{B}^{\prec}_{A\cup B\cup C}\right), \mathcal{M}^{\uparrow})$. Then we have

$$
\begin{aligned}
\mathcal{M}^{\downarrow} &= \text{join}(\text{select}\left(\gamma^{\bullet}_{B}(A), \mathcal{B}^{\prec}_{A}\right), \\
&\qquad \text{select}\left(\gamma^{\bullet}_{A}(B), \mathcal{B}^{\prec}_{B\cup C}\right)) \\
&\qquad \text{by Lemma 15 of [30]} \\
&= \text{select}\left(\gamma^{\bullet}_{B}(A) \cup \gamma^{\bullet}_{A}(B), \text{join}\left(\mathcal{B}^{\prec}_{A},\ \mathcal{B}^{\prec}_{B\cup C}\right)\right) \\
&\qquad \text{by Lemma 8.1.2} \\
&= \text{select}\left(\gamma^{\bullet}_{B}(A) \cup \gamma^{\bullet}_{A}(B), \mathcal{B}^{\prec}_{A\cup B\cup C}\right) \\
&\qquad \text{by definition of join}
\end{aligned}
$$

Since $\gamma^{\bullet}_{B}(A) \cup \gamma^{\bullet}_{A}(B) = \delta^{\bullet}(\gamma(A, B))$, we have $\mathcal{M}^{\downarrow} = \text{select}\left(\delta^{\bullet}(\gamma(A, B)), \mathcal{B}^{\prec}_{A\cup B\cup C}\right)$. □

8.3 Proof of Property 4.4.2

Proof. In order to establish Property 4.4.2, we first introduce the following lemma:

Lemma 8.3.1. *Let $\mathcal{H}$ be a BPH on V. Let $A \subseteq V$ and $B \subseteq V$ such that $A \subseteq B$. The we have $\textit{select}\,(A, \mathcal{H}) \sqsubseteq \textit{select}\,(B, \mathcal{H})$.*

Then, we proceed in two steps:

(1) $\mathcal{M}_i^{\uparrow} \sqsubseteq \mathcal{M}_i$; and

(2) $\mathcal{M}_i^{\downarrow} \sqsubseteq \mathcal{M}$.

(1) As stated in Section 4.4.1, $\mathcal{M}_i$ built on $\mathbf{P}$ is defined as follows

$$
\mathcal{M}_i = \text{select}\left(F, \mathcal{B}^{\prec}_{\cup\{S_j | j\in\{0,\ldots i\}\}}\right) \tag{8.1}
$$

where

$$F = \bigcup_{j=0}^{i} \left\{ \gamma^{\bullet}_{V \setminus S_j}(S_j) \right\} \tag{8.2}$$

From the definition of $\delta^{\bullet}$, γ and $\gamma^{\bullet}$, it can be seen that

$$F = \bigcup_{j=0}^{i} \left\{ \gamma^{\bullet}_{V \setminus S_j}(S_j) \right\} \tag{8.3}$$

$$= \bigcup_{j=1}^{i} \{ \delta^{\bullet}(\gamma(S_{j\text{-}1}, S_j)) \} \tag{8.4}$$

As stated in Property 4.4.1, we have

$$\mathcal{M}_i^{\uparrow} = \text{select} \left(C, \mathcal{B}^{\prec}_{\cup \{S_j | j \in \{0, \dots i\}\}} \right) \tag{8.5}$$

where

$$C = \delta^{\bullet}(\gamma(S_{i\text{-}1}, S_i)) \tag{8.6}$$

We can deduce that for any $i \in \{1, \dots, k\}$ we have $C \subseteq F$. Therefore, given Lemma 8.3.1 we have $\mathcal{M}_i^{\uparrow} \sqsubseteq \mathcal{M}_i$.

(2) For this second statement, we can express $\mathcal{M}$ and $\mathcal{M}_i$ with C from Equation (8.6) and F from Equation (8.4).

$$\mathcal{M} = \text{select}\,(F, \mathcal{B}^{\prec}_V) \tag{8.7}$$

$$\mathcal{M}_i^{\downarrow} = \text{select}\,(C, \mathcal{B}^{\prec}_V) \tag{8.8}$$

We have $C \subseteq F$, and with Lemma 8.3.1 we have $\mathcal{M}^{\downarrow} \sqsubseteq \mathcal{M}$.

□

8.4 Proof of Property 4.4.3

Proof. Given $n_i = |\text{select}\,(S_{i\text{-}1}, \mathcal{H})| + |\text{select}\,(S_i, \mathcal{H})|$, we will prove statements of Property 4.4.3 in the following order:

(1) $|\mathcal{M}_i^{\downarrow}| \leq n_i$; and

(2) $|\mathcal{M}_i^{\uparrow}| \leq n_i$.

(1) From Property 4.4.1, we have $\mathcal{M}_i^{\downarrow} = \text{select}\,(F, \mathcal{H})$ where $F = \delta^{\bullet}(\gamma(S_{i\text{-}1}, S_i))$. We can see that $F \subseteq (S_{i\text{-}1} \cup S_i)$ which gives us Equation (8.9). Each region R of $\text{select}\,(S_{i\text{-}1} \cup S_i, \mathcal{H})$ either intersects an element of $S_{i\text{-}1}$ or S_i or both. As a result, R belongs either to $\text{select}\,(S_{i\text{-}1}, \mathcal{H})$ or $\text{select}\,(S_i, \mathcal{H})$ or both. As a result, we have Equation (8.10).

$$|\mathcal{M}_i^{\downarrow}| \leq |\text{select}\,(S_{i\text{-}1} \cup S_i, \mathcal{H})| \tag{8.9}$$
$$\leq |\text{select}\,(S_{i\text{-}1}, \mathcal{H})| \; + |\text{select}\,(S_i, \mathcal{H})| \tag{8.10}$$

(2) Given $C = \delta^{\bullet}(\gamma(S_{i\text{-}1}, S_i))$, we can write the two structures as follows:

$$\mathcal{M}_i^{\downarrow} = \text{select}\,(C, \mathcal{B}_V^{\prec}) \tag{8.11}$$
$$\mathcal{M}^{\uparrow} = \text{select}\left(C, \mathcal{B}^{\prec}_{\cup\{S_j | j \in \{0,\ldots k\}\}}\right) \tag{8.12}$$

Considering Lemma 8.4.1, we have $|\mathcal{M}_i^{\uparrow}| \leq |\mathcal{M}_i^{\downarrow}|$. Thus we have $|\mathcal{M}_i^{\uparrow}| \leq |\mathcal{M}_i^{\downarrow}| \leq n_i$

Lemma 8.4.1. *Let $(S_0, \ldots, S_k)$ be a causal partition of V, $A \subseteq V$ and u be a pair of distinct elements of V. For any $i \in \{0, \ldots, k\}$, the following inequalities holds true:*

$$|\mathit{select}\,(A, \mathcal{H})| \leq |\mathit{select}\,(A, \mathcal{H} \boxplus \{u\})| \tag{8.13}$$
$$|\mathit{select}\left(X, \mathcal{B}^{\prec}_{\cup\{S_j | j \in \{0,\ldots i\}\}}\right)| \leq |\mathit{select}\,(X, \mathcal{B}_V^{\prec})| \tag{8.14}$$

□

Bibliography

[1] R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua, and S. Süsstrunk. Slic superpixels compared to state-of-the-art superpixel methods. *TPAMI*, 34 (11):2274–2282, 2012.

[2] C. Allène, J.-Y. Audibert, M. Couprie, and R. Keriven. Some links between extremum spanning forests, watersheds and min-cuts. *Image and Vision Computing*, 28(10):1460–1471, Oct. 2010. ISSN 0262-8856. `10.1016/j.imavis.2009.06.017`.

[3] J. M. Almira, H. Phelippeau, and A. Martinez-Sanchez. Fast normalized cross-correlation for template matching with rotations. *Journal of Applied Mathematics and Computing*, 70(5):4937–4969, June 2024. ISSN 1865-2085. `10.1007/s12190-024-02157-6`.

[4] J. Angulo and D. Jeulin. Stochastic watershed segmentation. *Proc. Int. Symp. Mathematical Morphology ISMM'07*, 1, 2007.

[5] P. Arbeláez, M. Maire, C. Fowlkes, and J. Malik. Contour detection and hierarchical image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(5):898–916, 2011.

[6] P. A. Arbeláez and L. D. Cohen. A metric approach to vector-valued image segmentation. *International Journal of Computer Vision*, 69(1):119–126, Apr. 2006.

[7] L. Arge, A. Danner, H. Haverkort, and N. Zeh. *I/O-Efficient Hierarchical Watershed Decomposition of Grid Terrain Models*, pages 825–844. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-35589-2.

[8] G. Barbosa da Fonseca, R. Negrel, B. Perret, J. Cousty, and S. Jamil F. Guimarães. Hierarchy-based fuzzy segmentation and marker learning layer: theory and algorithms. *Journal of Mathematical Imaging and Vision*, 67(4), June 2025. ISSN 1573-7683. `10.1007/s10851-025-01251-z`.

[9] D. Baum, J. C. Weaver, I. Zlotnikov, D. Knötel, L. Tomholt, and M. N. Dean. High-Throughput Segmentation of Tiled Biological Structures using Random-Walk Distance Transforms. *Integrative and Comparative Biology*, 59(6):1700–1712, 07 2019. ISSN 1540-7063.

[10] R. Beare and G. Lehmann. The watershed transform in itk - discussion and new developments. *The Insight Journal*, June 2006.

[11] F. C. Belem, B. Perret, J. Cousty, S. J. F. Guimaraes, and A. X. Falcao. Towards a simple and efficient object-based superpixel delineation framework. In *2021 34th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, page 346–353. IEEE, Oct. 2021.

[12] G. Bertrand. On topological watersheds. *Journal of Mathematical Imaging and Vision*, 22(2):217–230, 2005. ISSN 1573-7683.

[13] S. Beucher. *Watershed, Hierarchical Segmentation and Waterfall Algorithm*, pages 69–76. Springer Netherlands, Dordrecht, 1994. ISBN 978-94-011-1040-2.

[14] S. Beucher and F. Meyer. The morphological approach to segmentation: The watershed transformation. *Mathematical Morphology in Image Processing*, pages 433–481, 1992.

[15] N. Blin, E. Carlinet, F. Lemaitre, L. Lacassagne, and T. Geraud. Max-tree computation on GPUs. *IEEE TPDS*, 2022.

[16] P. Bosilj, E. Kijak, and S. Lefèvre. Partition and inclusion hierarchies of images: A comprehensive survey. *Journal of Imaging*, 4(2), 2018.

[17] N. Boutry, L. Najman, and T. Géraud. Some equivalence relation between persistent homology and morphological dynamics. *Journal of Mathematical Imaging and Vision*, 64(7):807–824, September 2022.

[18] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[19] E. J. Breen and R. Jones. Attribute openings, thinnings, and granulometries. *Computer Vision and Image Understanding*, 64(3):377–389, 1996.

[20] E. Carlinet and T. Geraud. Morphological object picking based on the color tree of shapes. In *IPTA*, pages 125–130. IEEE, Nov. 2015.

[21] E. Carlinet and T. Géraud. A comparative review of component tree computation algorithms. *IEEE Transactions on Image Processing*, 23(9):3885–3895, 2014.

[22] H. K. Cheng, J. Chung, Y.-W. Tai, and C.-K. Tang. Cascadepsp: Toward class-agnostic and very high-resolution segmentation via global and local refinement. In *CVPR*. IEEE, June 2020.

[23] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '95, page 139–149, USA, 1995. Society for Industrial and Applied Mathematics.

[24] C. Couprie, L. Grady, L. Najman, and H. Talbot. Power watershed: A unifying graph-based optimization framework. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(7):1384–1399, 2011.

[25] J. Cousty and L. Najman. Incremental algorithm for hierarchical minimum spanning forests and saliency of watershed cuts. In *ISMM*, pages 272–283, 2011.

[26] J. Cousty, G. Bertrand, L. Najman, and M. Couprie. Watershed cuts: Minimum spanning forests and the drop of water principle. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(8):1362–1374, 2009.

[27] J. Cousty, G. Bertrand, L. Najman, and M. Couprie. Watershed cuts: Thinnings, shortest path forests, and topological watersheds. *TPAMI*, 32(5): 925–939, May 2010. `10.1109/tpami.2009.71`.

[28] J. Cousty, L. Najman, and B. Perret. Constructive links between some morphological hierarchies on edge-weighted graphs. In *ISMM*, pages 86–97, 2013.

[29] J. Cousty, L. Najman, Y. Kenmochi, and S. Guimarães. Hierarchical segmentations with graphs: quasi-flat zones, minimum spanning trees, and saliency maps. *JMIV*, 60(4):479–502, 2018.

[30] J. Cousty, B. Perret, H. Phelippeau, S. Carneiro, P. Kamlay, and L. Buzer. An algebraic framework for out-of-core hierarchical segmentation algorithms. In *DGMM*, pages 378–390, 2021.

[31] A. Danner, T. Mølhave, K. Yi, P. K. Agarwal, L. Arge, and H. Mitasova. Terrastream: From elevation data to watershed hierarchies. In *ACM Symposium on Advances in Geographic Information Systems*, pages 212–219, 2007.

[32] M. de Goede, E. Johlin, B. Sciacca, F. Boughorbel, and E. C. Garnett. 3d multi-energy deconvolution electron microscopy. *Nanoscale*, 9(2):684–689, 2017. ISSN 2040-3372. 10.1039/c6nr07991a.

[33] L. Dhulipala, X. Dong, K. N. Gowda, and Y. Gu. Optimal parallel algorithms for dendrogram computation and single-linkage clustering. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, page 233–245. ACM, June 2024.

[34] H. Digabel and C. Lantuejoul. Iterative algorithms. *Proceedings of the 2nd European Symposium Quantitative Analysis of Microstructures in Material Science, Biology and Medicine*, pages 85–89, 1978.

[35] A. Falcao and F. Bergo. Interactive Volume Segmentation With Differential Image Foresting Transforms. *IEEE Transactions on Medical Imaging*, 23(9): 1100–1108, Sept. 2004.

[36] A. Falcão, J. Stolfi, and R. de Alencar Lotufo. The image foresting transform: theory, algorithms, and applications. *IEEE TPAMI*, 26(1):19–29, 2004.

[37] E. C. J. R. C. I. for the Protection and the Security of the Citizen. *The alpha-tree algorithm: theory, algorithms, and applications.* Publications Office, LU, 2012. 10.2788/48773. URL https://data.europa.eu/doi/10.2788/48773.

[38] S. Gazagnes and M. H. Wilkinson. Parallel attribute computation for distributed component forests. In *2022 IEEE International Conference on Image Processing (ICIP)*, pages 601–605, 2022.

[39] S. Gazagnes and M. H. F. Wilkinson. Distributed connected component filtering and analysis in 2d and 3d tera-scale data sets. *IEEE Transactions on Image Processing*, 30:3664–3675, 2021.

[40] L. Gigli, S. Velasco-Forero, and B. Marcotegui. On minimum spanning tree streaming for hierarchical segmentation. *PRL*, 138:155–162, 2020.

[41] M. Götz, G. Cavallaro, T. Geraud, M. Book, and M. Riedel. Parallel computation of component trees on distributed memory machines. *TPDS*, 29 (11):2582–2598, 2018.

[42] L. Grady. Random walks for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(11):1768–1783, Nov. 2006.

[43] M. Grimaud. A new measure of contrast: the dynamics. *Image Algebra and Morphological Image Processing*, 1769:292–305, 1992.

[44] R. Grompone von Gioi and G. Randall. A brief analysis of the dense extreme inception network for edge detection. *Image Processing On Line*, 12:389–403, Oct. 2022.

[45] J. Havel, F. Merciol, and S. Lefèvre. Efficient schemes for computing α-tree representations. In *Mathematical Morphology and Its Applications to Signal and Image Processing*, pages 111–122, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38294-9.

[46] J. Havel, F. Merciol, and S. Lefèvre. Efficient tree construction for multiscale image representation and processing. *JRTIP*, 16(4):1129–1146, 2019.

[47] R. Henderson. From electron crystallography to single particle cryoem (nobel lecture). *Angewandte Chemie International Edition*, 57(34):10804–10825, July 2018.

[48] M. Hučko and M. Šrámek. Streamed watershed transform on gpu for processing of large volume data. In *Proceedings of the 28th Spring Conference on Computer Graphics*, SCCG '12, page 137–141, New York, NY, USA, 2012. Association for Computing Machinery.

[49] H. Jegou, M. Douze, and C. Schmid. *Hamming Embedding and Weak Geometric Consistency for Large Scale Image Search*, page 304–317. Springer Berlin Heidelberg, 2008.

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

[51] C. Jordan. On the crest lines and synclical lines. *C. R. Acad. Sci., Paris*, 74:1457–1459, 1872.

[52] S. K C, J. Aryal, and D. Ryu. Automated delineation of the agricultural fields using multi-task deep learning and optical satellite imagery. In *IGARSS 2023 - 2023 IEEE International Geoscience and Remote Sensing Symposium*, pages 2795–2798, 2023.

[53] J. J. Kazemier, G. K. Ouzounis, and M. H. Wilkinson. Connected morphological attribute filters on distributed memory parallel machines. In *ISMM*, pages 357–368, 2017.

[54] B. Le Moigne, C. Rault, F. Guiotte, D. J. Thomas, and T. Dewez. RIDIM: Unveiling Rock Instabilities through Hierarchical Segmentation of 3D Point Clouds. 14th International Symposium on Landslides, July 2024. Poster.

[55] Q. Lebon, J. Lefèvre, J. Cousty, and B. Perret. Interactive segmentation with incremental watershed cuts. *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, page 189–200, Nov. 2023. 10.1007/978-3-031-49018-7_14.

[56] Q. Lebon, J. Lefèvre, J. Cousty, and B. Perret. Incremental watershed cuts: Interactive segmentation algorithm with parallel strategy. *Pattern Recognition Letters*, 189:256–263, Mar. 2025. 10.1016/j.patrec.2024.12.005.

[57] J. Lefèvre, J. Cousty, B. Perret, and H. Phelippeau. Join, select, and insert: Efficient out-of-core algorithms for hierarchical segmentation trees. In *Discrete Geometry and Mathematical Morphology*, pages 274–286. Springer, 2022. 10.1007/978-3-031-19897-7_22.

[58] J. Lefèvre, J. Cousty, B. Perret, and H. Phelippeau. Out-of-core attribute algorithms for binary partition hierarchies. *Discrete Geometry and Mathematical Morphology*, pages 298–311, 2024. 10.1007/978-3-031-57793-2_23.

[59] J. Lefèvre, J. Cousty, B. Perret, and H. Phelippeau. Out-of-core algorithms for binary partition hierarchies. *Journal of Mathematical Imaging and Vision*, 67(2):18, 2025. 10.1007/s10851-025-01234-0.

[60] Y.-C. Lin, Y.-P. Tsai, Y.-P. Hung, and Z.-C. Shih. Comparison between immersion-based and toboggan-based watershed image segmentation. *IEEE Transactions on Image Processing*, 15(3):632–640, 2006.

[61] P. Lindstrom and V. Pascucci. Terrain simplification simplified: a general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, 2002.

[62] A. Lucchi, K. Smith, R. Achanta, G. Knott, and P. Fua. Supervoxel-based segmentation of mitochondria in EM image stacks with learned shape features. *IEEE Transactions on Medical Imaging*, 31(2):474–486, 2012.

[63] O. Lézoray. Stochastic permutation ordering watershed. In *2021 29th European Signal Processing Conference (EUSIPCO)*, pages 726–730, 2021.

[64] V. Machairas, M. Faessel, D. Cardenas-Pena, T. Chabardes, T. Walter, and E. Decenciere. Waterpixels. *IEEE Transactions on Image Processing*, 24 (11):3707–3716, Nov. 2015.

[65] D. S. Maia, M.-T. Pham, and S. Lefèvre. Watershed-based attribute profiles with semantic prior knowledge for remote sensing image analysis. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 15:2574–2591, 2022.

[66] F. Malmberg and C. L. Luengo Hendriks. An efficient algorithm for exact evaluation of stochastic watersheds. *Pattern Recognition Letters*, 47:80–84, 2014. Advances in Mathematical Morphology.

[67] A. Martinez-Sanchez, U. Homberg, J. M. Almira, and H. Phelippeau. *Tensorial Template Matching for Fast Cross-Correlation with Rotations and Its Application for Tomography*, page 19–35. Springer Nature Switzerland, Nov. 2024. ISBN 9783031733833. 10.1007/978-3-031-73383-3_2.

[68] 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 *Advanced Concepts for Intelligent Vision Systems*, pages 230–241, Berlin, Heidelberg, 2008. Springer. ISBN 978-3-540-88458-3.

[69] J. C. Maxwell. L. on hills and dales: To the editors of the philosophical magazine and journal. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 40(269):421–427, Dec. 1870.

[70] J. Mendes Forte, N. Passat, and Y. Kenmochi. *Building the Topological Tree of Shapes from the Tree of Shapes*, page 271–285. Springer Nature Switzerland, 2024. ISBN 9783031577932. 10.1007/978-3-031-57793-2_21. URL http://dx.doi.org/10.1007/978-3-031-57793-2_21.

[71] J. Mendes Forte, N. Passat, and Y. Kenmochi. *Designing Connected Operators Using the Topological Tree of Shapes*, page 195–209. Springer Nature Switzerland, Nov. 2025. ISBN 9783032095442. 10.1007/978-3-032-09544-2_14. URL http://dx.doi.org/10.1007/978-3-032-09544-2_14.

[72] F. Meyer. Color image segmentation. In *1992 International Conference on Image Processing and its Applications*, pages 303–306, 1992.

[73] F. Meyer. Minimum spanning forests for morphological segmentation. In *Proceedings of the Second International Conference on Mathematical Morphology and its Applications to Image Processing*, pages 77–84, September 1994.

[74] F. Meyer. The dynamics of minima and contours. In *ISMM*, pages 329–336, 1996.

[75] F. Meyer and P. Maragos. Morphological scale-space representation with levelings. In *International Conference on Scale-Space Theories in Computer Vision*, pages 187–198, 1999.

[76] A. Miles, J. Kirkham, M. Durant, J. Bourbeau, T. Onalan, J. Hamman, Zain Patel, Shikharsg, M. Rocklin, R. Dussin, V. Schut, E. S. D. Andrade, R. Abernathey, C. Noyes, Sbalmer, Pyup.Io Bot, T. Tran, S. Saalfeld, J. Swaney, J. Moore, J. Jevnik, J. Kelleher, J. Funke, G. Sakkis, C. Barnes, and A. Banihirwe. zarr-developers/zarr-python: v2.4.0, 2020. URL `https://zenodo.org/record/3773450`.

[77] P. Monasse and F. Guichard. *Scale-Space from a Level Lines Tree*, page 175–186. Springer Berlin Heidelberg, 1999. ISBN 9783540482369. `10.1007/3-540-48236-9_16`. URL `http://dx.doi.org/10.1007/3-540-48236-9_16`.

[78] L. Najman. Ultrametric watersheds. In M. H. F. Wilkinson and J. B. T. M. Roerdink, editors, *Mathematical Morphology and Its Application to Signal and Image Processing*, pages 181–192, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-03613-2.

[79] L. Najman and J. Cousty. A graph-based mathematical morphology reader. *Pattern Recognition Letters*, 47:3–17, 2014. Advances in Mathematical Morphology.

[80] L. Najman and M. Schmitt. Geodesic saliency of watershed contours and hierarchical segmentation. *TPAMI*, 18(12):1163–1173, 1996.

[81] L. Najman, J. Cousty, and B. Perret. Playing with Kruskal: algorithms for morphological trees in edge-weighted graphs. In *ISMM*, pages 135–146, 2013.

[82] P. Neubert and P. Protzel. Compact watershed and preemptive slic: On improving trade-offs of superpixel segmentation algorithms. In *2014 22nd International Conference on Pattern Recognition*. IEEE, Aug. 2014. `10.1109/icpr.2014.181`. URL `http://dx.doi.org/10.1109/ICPR.2014.181`.

[83] M. Ô. V. Ngoc, E. Carlinet, J. Fabrizio, and T. Géraud. The dahu graph-cut for interactive segmentation on 2d/3d images. *PR*, 136:109–207, 2023.

[84] H. T. Nguyen, M. Worring, and R. van den Boomgaard. Watersnakes: energy-driven watershed segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(3):330–342, 2003.

[85] G. K. Ouzounis and P. Soille. Pattern spectra from partition pyramids and hierarchies. In P. Soille, M. Pesaresi, and G. K. Ouzounis, editors, *Mathematical Morphology and Its Applications to Image and Signal Processing*, pages 108–119. Springer, 2011. ISBN 978-3-642-21569-8.

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

[87] P. V. V. Paiva, C. K. Cogima, E. Dezen-Kempter, and M. A. G. Carvalho. Historical building point cloud segmentation combining hierarchical watershed transform and curvature analysis. *Pattern Recognition Letters*, 135: 114–121, 2020.

[88] N. Passat and Y. Kenmochi. *A Topological Tree of Shapes*, page 221–235. Springer International Publishing, 2022. ISBN 9783031198977. 10.1007/978-3-031-19897-7_18. URL http://dx.doi.org/10.1007/978-3-031-19897-7_18.

[89] N. Passat, B. Naegel, F. Rousseau, M. Koob, and J.-L. Dietemann. Interactive segmentation based on component-trees. *Pattern Recognition*, 44 (10–11):2539–2554, Oct. 2011.

[90] N. Passat, C. Kurtz, and A. Vacavant. Editorial — virtual special issue: "hierarchical representations: New results and challenges for image analysis". *Pattern Recognition Letters*, 138:201–203, Oct. 2020.

[91] B. Perret, G. Chierchia, J. Cousty, S. J. F. Guimarães, Y. Kenmochi, and L. Najman. Higra: Hierarchical graph analysis. *SoftwareX*, 10:100335, 2019.

[92] H. Phelippeau, C. Levien, and T. Franke. Automated and robust method for recording nm-resolution 3d image data from serial ultra-sections of life-sciences samples with electron microscopes, Dec 2020. Publication of US20200410697A1; priority date 2019-06-28; filed 2020-06-26.

[93] B. L. Price, B. Morse, and S. Cohen. Geodesic graph cut for interactive image segmentation. In *CVPR*, 2010.

[94] K. R, K. S, W. W, P. J, and K. A. Chlamydomonas cryo-slice and view on thermo scientific helios 5 hydra pfib, Oct. 2022. URL http://dx.doi.org/10.6019/EMPIAR-11275.

[95] S. Ram and J. J. Rodriguez. Random walker watersheds: A new image segmentation approach. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, page 1473–1477. IEEE, May 2013. 10.1109/icassp.2013.6637896. URL http://dx.doi.org/10.1109/ICASSP.2013.6637896.

[96] J. F. Randrianasoa, C. Kurtz, Éric Desjardin, and N. Passat. Binary partition tree construction from multiple features for image segmentation. *Pattern Recognition*, 84:237–250, 2018.

[97] C. Ronse. Partial partitions, partial connections and connective segmentation. *JMIV*, 32(2):97–125, 2008.

[98] P. Salembier and L. Garrido. Binary partition tree as an efficient representation for image processing, segmentation, and information retrieval. *TIP*, 9(4):561–576, 2000.

[99] P. Salembier and J. Serra. Flat zones filtering, connected operators, and filters by reconstruction. *IEEE Transactions on Image Processing*, 4(8): 1153–1160, 1995.

[100] P. Salembier, A. Oliveras, and L. Garrido. Antiextensive connected operators for image and sequence processing. *IEEE Transactions on Image Processing*, 7(4):555–570, 1998.

[101] E. F. Sanmartín, S. Damrich, and F. A. Hamprecht. Directed probabilistic watershed. In *NeurIPS 2021*, NeurIPS '21, Red Hook, NY, USA, 2021. Curran Associates Inc. ISBN 9781713845393.

[102] D. Santana Maia, A. de Albuquerque Araujo, J. Cousty, L. Najman, B. Perret, and H. Talbot. *Evaluation of Combinations of Watershed Hierarchies*, page 133–145. Springer International Publishing, 2017. ISBN 9783319572406. 10.1007/978-3-319-57240-6_11. URL http://dx.doi.org/10.1007/978-3-319-57240-6_11.

[103] D. Santana Maia, J. Cousty, L. Najman, and B. Perret. Properties of combinations of hierarchical watersheds. *Pattern Recognition Letters*, 128: 513–520, Dec. 2019. ISSN 0167-8655. 10.1016/j.patrec.2019.10.009. URL http://dx.doi.org/10.1016/j.patrec.2019.10.009.

[104] P. Sao, A. Prokopenko, and D. Lebrun-Grandie. Pandora: A parallel dendrogram construction algorithm for single linkage clustering on gpu. In *Proceedings of the 53rd International Conference on Parallel Processing*, ICPP, page 908–918. ACM, Aug. 2024.

[105] B. Selig, F. Malmberg, and C. L. L. Hendriks. Fast evaluation of the robust stochastic watershed. In J. A. Benediktsson, J. Chanussot, L. Najman, and H. Talbot, editors, *Mathematical Morphology and Its Applications to Signal and Image Processing*, pages 705–716, Cham, 2015. Springer International Publishing. ISBN 978-3-319-18720-4.

[106] A. G. Silva and R. de Alencar Lotufo. Efficient computation of new extinction values from extended component tree. *Pattern Recognition Letters*, 32(1): 79–90, 2011. Image Processing, Computer Vision and Pattern Recognition in Latin America.

[107] D. J. Silva, W. A. Alves, and R. F. Hashimoto. Incremental bit-quads count in component trees: Theory, algorithms, and optimization. *PRL*, 129:33–40, 2020.

[108] A. P. Soares, D. Baum, B. Hesse, A. Kupsch, B. R. Müller, and P. Zaslansky. Scattering and phase-contrast x-ray methods reveal damage to glass fibers in endodontic posts following dental bur trimming. *Dental Materials*, 37(2): 201–211, 2021.

[109] P. Soille. *Morphological Image Analysis*. Springer Berlin Heidelberg, 1999. ISBN 9783662039397. 10.1007/978-3-662-03939-7.

[110] P. Soille. Constrained connectivity for hierarchical image partitioning and simplification. *TPAMI*, 30(7):1132–1145, 2008.

[111] L. Soler, A. Hostettler, V. Agnus, A. Charnoz, J. Fasquel, J. Moreau, A. Osswald, M. Bouhadjar, and J. Marescaux. 3d image reconstruction for comparison of algorithm database: A patient specific anatomical and medical image database. *IRCAD*, 1(1), 2010.

[112] X. Soria, A. Sappa, P. Humanante, and A. Akbarinia. Dense extreme inception network for edge detection. *Pattern Recognition*, 139:109461, July 2023.

[113] D. Stalling, M. Westerhoff, and H.-C. Hege. *amira: A Highly Interactive System for Visual Data Analysis*, page 749–767. Elsevier, 2005.

[114] C. N. Straehle, U. Köthe, G. Knott, and F. A. Hamprecht. Carving: Scalable Interactive Segmentation of Neural Volume Electron Microscopy Images. In *MICCAI*, pages 653–660, 2011.

[115] H. SUN and Y. LUO. Adaptive watershed segmentation of binary particle image. *Journal of Microscopy*, 233(2):326–330, 2009.

[116] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, Apr. 1975.

[117] N. The HDF Group and Q. Koziol. Hdf5-version 1.12.0, 2020. URL https://www.osti.gov/doecode/biblio/9801.

[118] Thermo Fisher Scientific. *Electron Microscopy Handbook*. Thermo Fisher Scientific, 2025. URL `https://www.thermofisher.com/fr/fr/home/electron-microscopy/learning-center.html`. Accessed: 2025-10-12.

[119] S. Toledo. *A survey of out-of-core algorithms in numerical linear algebra*, page 161–179. American Mathematical Society, USA, 1999.

[120] C. Vachier and F. Meyer. Extinction value: A new measurement of persistence. *IEEE Workshop on Non Linear Signal and Image Processing*, 1: 254–257, 06 1995.

[121] C. Vachier and F. Meyer. The viscous watershed transform. *Journal of Mathematical Imaging and Vision*, 22(2–3):251–267, May 2005.

[122] L. Vincent and P. Soille. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(6):583–598, 1991.

[123] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33(2):209–271, jun 2001.

[124] S. Wang, M. Zhang, K. Yang, K. Chen, S. Ma, J. Jiang, and Y. Wu. Noswalker: A decoupled architecture for out-of-core random walk processing. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 466–482, New York, NY, USA, 2023. ACM.

[125] M. Wilkinson, H. Gao, W. 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*, 30(10): 1800–1813, Oct. 2008.

[126] S. Wolf, C. Pape, A. Bailoni, N. Rahaman, A. Kreshuk, U. Köthe, and F. A. Hamprecht. The mutex watershed: Efficient, parameter-free image partitioning. In V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, editors, *Computer Vision – ECCV 2018*, pages 571–587, Cham, 2018. Springer International Publishing. ISBN 978-3-030-01225-0.

[127] C. S. Xu, K. J. Hayworth, Z. Lu, P. Grob, A. M. Hassan, J. G. García-Cerdán, K. K. Niyogi, E. Nogales, R. J. Weinberg, and H. F. Hess. Enhanced fib-sem systems for large-volume 3d imaging. *eLife*, 6:e25916, may 2017.

[128] I. Youkana, J. Cousty, R. Saouli, and M. Akil. Parallelization strategy for elementary morphological operators on graphs: Distance-based algorithms and implementation on multicore shared-memory architecture. *Journal of Mathematical Imaging and Vision*, 59(1):136–160, June 2017.

List of Figures

1.1 Comparison of the light microscope with SEM and TEM. Illustration used courtesy of Thermo Fisher Scientific [118]. 13
1.2 Schematic view of volume electron microscopy techniques. Images courtesy of Thermo Fisher Scientific. 14
1.3 **(a)** Volumescope 2 SEM for serial block-face imaging system with multi-energy deconvolution. **(b)** Helios Hydra PFIB-SEM for large-volume milling and volumetric imaging. Images courtesy of Thermo Fisher Scientific 15
1.4 Reconstructed cryo-PFIB image and deep learning-based segmentation of *Chlamydomonas reinhardtii* [94]. 16
1.5 Marker-based watershed segmentation of a chocolate bar CT volume. 17
1.6 Example of interactive marker-based watershed segmentation of a nearly isotropic FIB-SEM mammalian cerebral cortex volume [62]. . 18
1.7 Example of a hierarchy of partitions of an image from the BSDS500 dataset [5], represented as a series of nested segmentations and an associated tree of regions. 19
1.8 **(a)** Comparison between the expected watershed outcome and **(b)** the naive block-wise algorithm result. 22

2.1 Watersheds of the North American continent. The watershed lines separate three catchment basins: the Atlantic, Pacific and Arctic oceans. 28
2.2 **(a)** Original image of uranium oxide **(b)** Corresponding topographic relief where watershed segmentation can be applied. 29
2.3 **(a)** Original topographic relief exhibiting many spurious minima **(b)** Homotopy modification with h-minima transform (h=20) **(c)** Watersheds of the original relief **(d)** Watersheds of the h-minima filtered relief. 31
2.4 Watershed segmentation of the uranium oxide image given the set of disk markers. 32

2.5 A simple gray-scale image, the foreground components of each threshold set, also know as peak components, and the resulting component tree, which is referred to as a max-tree in this case. Courtesy of Gazagnes and Wilkinson [39]. 33
2.6 Partitions of the quasi-flat zones hierarchy using absolute difference between 4-adjacent pixels as dissimilarity measure. Courtesy of [37]. 34
2.7 Saliency maps obtained from hierarchical watershed using different attributes to order the minima. 37

3.1 The series (P_0, P_1, P_2, P_3) of partitions of the set $\{1, 2, 3, 4, 5\}$ and its tree representation. Each node is associated with the number of the region it represents. 48
3.2 A BPH built on a 1D edge-weighted graph $\{a, b, c, d, e\}$. The red arrows represent the map μ from non-leaf regions to edges of the MST. The leaves of the BPH correspond to the vertices of the MST. 50

4.1 Left, a BPH $\mathcal{H}$ built on an edge-weighted graph $(weight/index)$ and the visualization of the nested series of partition. Red edges of the graph belong to the minimum spanning tree. Right, the distribution $\{\mathcal{H}_0, \mathcal{H}_1, \mathcal{H}_2\}$ of $\mathcal{H}$ over the causal partition $\{S_0, S_1, S_2\}$. 58
4.2 Illustration of the supremum of hierarchies. The leftmost hierarchy correspond to the BPH built over the graph excluding the rightmost edge. The middle hierarchy correspond to the BPH built over the graph composed of the excluded edge and its extremities. Finally, the rightmost hierarchy is the supremum of the two previous ones. . 60
4.3 *Join* of $\mathcal{X}$ and $\mathcal{Y}$ given their common neighborhood $\{\{d, f\}, \{c, e\}\}$ of ranks 6 and 4. 61
4.4 Insertion of $\mathcal{X}$ into $\mathcal{Y}$ to obtain $\mathcal{Z}$. Each region is represented by the vertices of the input space included in this region in white. The region U of $\mathcal{Y}$ at altitude 6 is discarded from $\mathcal{Z}$ because of its non-null intersection with $gr(\mathcal{X}[6])$ (highlighted in red). 62
4.5 Operators to access specific subsets of the graph. 62
4.6 Illustration of Counterexample 4.4.1. The partition of the space $\{x_0, \ldots, x_7\}$ into 4 tiles is represented by the black rectangles, the BPH is represented by the black and blue tree while the GBT is highlighted in bold blue. 66
4.7 Given a partition of the space in 4 slices, the BPH in black and the merged tree of the last two slices in bold blue. 69

4.8 (a) An edge-weighted graph with edge/vertex index in black/white and edge weights in red, (b) its BPH where each node is associated with a local index, and (c) the data structures used to represent the BPH. 71
4.9 A 6-adjacency grid of size $3 \times 3 \times 3$ with vertices indexed in raster scan order. Green horizontal edges are indexed first, followed by blue vertical edges, and finally red depth edges. 73
4.10 FIB-SEM volume of the CA1 hippocampus region of the brain [62]. The left image shows a 3D rendering of the volume while the right image is a slice of the volume. The image is composed of $2048 \times 1536 \times 1065$ voxels with an isotropic size of 5 nm. 82
4.11 Comparative analysis of peak memory consumption and execution time between the In-Core algorithm and the out-of-core causal given as a function of the image size. The variable w correspond to the slice width. 83
4.12 Comparative analysis across different computation patterns on FIB-SEM volume as a function of the number of slices. 84
4.13 Causal and grid partition into 16 tiles. It is necessary to draw 15 frontiers (in red) for the causal partition and only 9 for the grid partition to reach the same number of slices. 85

5.1 Illustration of the distribution of the area attribute A. Each node is identified by a letter and its area is given as subscript. Nodes with red dashed borders in **(b)** are regions where the attribute differs from the original hierarchy. 90
5.2 Illustration of the PROPAGATE algorithm. Each node is identified by a letter, and its attribute value (area) is indicated as a subscript. Red arrows show the replicated regions detected by MERGE and point to the region whose attribute is updated. Operand order is given by Algorithm 8. 94
5.3 Illustration of the watershed cut using BPH and horizontal cut. The graph G is shown in **(a)** with two markers on a and c, the BPH $\mathcal{B}$ built on G is shown in **(b)** and the watershed node is outlined in cyan, the reweighted MST T is shown in **(c)** with the corresponding watershed edge represented in dashed cyan, and the BPH built on T is shown in **(d)**. 100
5.4 **(a)** edge-weighted graph G partitioned into two tiles $\{a, c\}$ and $\{b, d\}$. **(b)** BPH $\mathcal{B}$ built on the graph G. The hierarchy select $(\{a, c\}, \mathcal{B})$ is in black. The node of altitude 1 corresponding to the unique MST edge within the first tile is outlined in red. 104
5.5 Minima number attribute computation. 106

5.6 **(a)** base graph with three minima outlined in color, **(b)** partition hierarchy corresponding to the min-tree, **(c)** BPH built on the graph, **(d)** min-tree of the MST. In both trees, the area associated with each region is indicated in the top right corner. 109

6.1 Illustration of the proposed workflow. **(b)** The input image is partitioned into tiles. **(c)** A region adjacency graph is built for each tile. Frontiers between superpixel in purple and lines represent edges of the minimum spanning tree linking two neighboring superpixels. Edge color represents their weights from light yellow to dark red. **(d)** The partial BPHs are then constructed from these RAGs. . . . 115
6.2 Example of superpixel obtained with Waterpixels [64] in Amira software. Yellow lines defines the boundaries of the superpixels. Red markers are placed in the left lung, blue markers in the right lung and green markers in the background. 118
6.3 Visualization of the α-flat zones obtained for different α values. The number of α-flat zones and the percentage of remaining regions compared to the original pixel-level partition are indicated below each image. 123
6.4 Comparative analysis between pixel-level (`Classic`) and α-flat zone-based (`QFZ`) distribution construction for various α values. 124

7.1 Three stages in an interactive segmentation session. The red lines represent background seeds, the green lines represent object seeds, and the yellow lines represent the segmentation produced by the current set of seeds. 130
7.2 Overview of our workflow of the interactive incremental watershed segmentation 131
7.3 Integer labels on non-leaf nodes represent the value of `visitCount`. (a) Initialization with S_1 and S_2 as markers. (b) Update after the addition of S_3 as an additional marker. 132
7.4 Computation time in ms during an interactive session (user generated markers). The plain black curves show the number of pixels updated at each user interaction (right axis). 140
7.5 Average throughput in megapixel per second on the BIG dataset with simulated interactions. In order to avoid noisy visualization, the curve is smoothed by averaging values in a window of size 2. . . 141
7.6 Throughput in megapixel per second on 3D images with simulated interactions. An average smoothing with a window size of 2 is used to prevent noisy aspects. 143

7.7 Connected component labeling time for an interactive session on patient2 with respect to the parameter MIN_BREADTH and to the number of threads 144
7.8 (a) Speedup of parallel over sequential breadth-first search. (b) Proportion of parallel iterations (blue) and of vertices traversed in a parallel iteration (green) with Parallel Breadth-First Search with respect to MIN_BREATH values. 146

List of Algorithms

1 Global Border Tree 63
2 GBT-based distributed BPH 64
3 Out-of-core BPH for causal partitioning [30]. 67
4 select 76
5 Join 77
6 Auxiliary functions for join algorithm 79
7 Insert 81

8 Propagate 92
9 Merge 93
10 Accumulate 95
11 Area distribution 96
12 Volume distribution 96
13 Height distribution 97
14 Minima distribution 98
15 Watershed nodes 101
16 Reweight MST 102
17 Horizontal Cut 102
18 Out-of-core marker-based watershed 103
19 Rightmost slice distribution 107
20 Minima number distribution 107
21 Out-of-core watershed 108
22 QFZ Attribute 110
23 Out-of-core Persistence 111

24 α-flat zones Region Adjacency Graph 119

25 Add MArkers 133
26 Remove Markers 134
27 Parallel Breadth-First Search 136
28 Successor Labeling 136