Towards Distributed Task-based Visualization and Data Analysis

---

Vom Fachbereich Informatik der Technischen Universität Kaiserslautern zur Verleihung des akademischen Grades Doktor der Naturwissenschaften (Dr. rer. nat.)

genehmigte Dissertation

von

Kilian Werner

Datum der wissenschaftlichen Aussprache: May 19, 2022
Dekan: Prof. Dr. Jens Schmitt
Berichterstatter: Prof. Dr. Christoph Garth
Berichterstatter: Dr. Julien Tierny

DE-386

Kilian Werner
Towards Distributed Task-based Visualization and Data Analysis
Dissertation, May 19, 2022
Reviewers: Prof. Dr. Christoph Garth and Dr. Julien Tierny
Supervisors: Prof. Christoph Garth **TU Kaiserslautern**
Scientific Visualization Lab
Computer Science Department
Gottlieb-Daimler-Straße 47
67663 Kaiserslautern

Abstract

To support scientific work with large and complex data the field of scientific visualization emerged in computer science and produces images through computational analysis of the data. Frameworks for combination of different analysis and visualization modules allow the user to create flexible pipelines for this purpose and set the standard for interactive scientific visualization used by domain scientists.

Existing frameworks employ a thread-parallel message-passing approach to parallel and distributed scalability, leaving the field of scientific visualization in high performance computing to specialized ad-hoc implementations. The task-parallel programming paradigm proves promising to improve scalability and portability in high performance computing implementations and thus, this thesis aims towards the creation of a framework for distributed, task-based visualization modules and pipelines.

The major contribution of the thesis is the establishment of modules for Merge Tree construction and (based on the former) topological simplification. Such modules already form a necessary first step for most visualization pipelines and can be expected to increase in importance for larger and more complex data produced and/or analysed by high performance computing.

To create a task-parallel, distributed Merge Tree construction module the construction process has to be completely revised. We derive a novel property of Merge Tree saddles and introduce a novel task-parallel, distributed Merge Tree construction method that has both good performance and scalability. This forms the basis for a module for topological simplification which we extend by introducing novel alternative simplification parameters that aim to reduce the importance of prior domain knowledge to increase flexibility in typical high performance computing scenarios.

Both modules lay the groundwork for continuative analysis and visualization steps and form a fundamental step towards an extensive task-parallel visualization pipeline framework for high performance computing.

Zusammenfassung

Wissenschaftliche Visualisierung ist eine Disziplin der Informatik, die durch computergestützte Analyse Bilder aus Datensätzen erzeugt, um das wissenschaftliche Arbeiten mit großen und komplexen Daten zu unterstützen. Softwaresysteme, die dem Anwender die Kombination verschiedener Analyse- und Visualisierungsmodule zu einer flexiblen Pipeline erlauben, stellen den Standard für interaktive wissenschaftliche Visualisierung.

Die hierfür bereits existierenden Systeme setzen auf Thread-Parallelisierung mit expliziter Kommunikation, sodass das Feld der wissenschaftlichen Visualisierung auf Hochleistungsrechnern meist spezialisierten Direktlösungen überlassen wird. An dieser Stelle scheint Task-Parallelisierung vielversprechend, um Skalierbarkeit und Übertragbarkeit von Lösungen für Hochleistungsrechner zu verbessern. Daher zielt die vorliegende Arbeit auf die Umsetzung eines Softwaresystems für verteilte und task-parallele Visualisierungsmodule und -pipelines ab.

Der zentrale Beitrag den die vorliegende Arbeit leistet ist die Einführung zweier Module für Merge Tree Konstruktion und topologische Datenbereinigung. Solche Module stellen bereits einen notwendigen ersten Schritt für die meisten Visualisierungspipelines dar und werden für größere und komplexere Datensätze, die im Hochleistungsrechnen erzeugt beziehungsweise analysiert werden, erwartungsgemäß noch wichtiger.

Um eine Task-parallele, verteilbare Konstruktionsmethode für Merge Trees zu entwickeln musste der etablierte Algorithmus grundlegend überarbeitet werden. In dieser Arbeit leiten wir eine neue Eigenschaft für Merge Tree Knoten her und entwickeln einen neuartigen Konstruktionsalgorithmus, der gute Performance und Skalierbarkeit aufweist. Darauf aufbauend entwickeln wir ein Modul für topologische Datenbereinigung, welche wir durch neue, alternative Bereinigungsparameter erweitern, um die Flexibilität im Einstaz auf Hochleistungsrechnern zu erhöhen.

Beide Module ermöglichen weiterführende Analyse und Visualisierung und setzen einen Grundstein für die Entwicklung eines umfassenden Task-parallelen Softwaresystems für Visualisierungspipelines auf Hochleistungsrechnern.

Contents

1 Introduction

> *[A poet would be] overcome by sleep and hunger before [being able to] describe with words what a painter is able to [depict] in an instant.*
>
> **— Leonardo da Vinci**

1.1 Motivation and Problem Statement

Simulation has established itself as a third pillar of science besides theory and experiment. Both simulation and experimental measurements of contemporary scientific work produce digital data of increasing size and complexity. Understanding and working with this data is becoming an ever larger challenge for domain scientists, and partially automated computational analysis has become essential in this process. This PhD thesis aims to support scientific work with large and complex data.

More specifically, we want to support scientific work with *Scientific Visualization*, which has formed a discipline of computer science and is a form of data analysis that produces visual images. Exploration and understanding of data features and deductions by humans is greatly improved by this. However, with increasing data size both scientific importance and computational difficulty of scientific visualization grow.

The use of *high performance computing (HPC)* in simulations produces data that is typically too large to analyse and visualize without also employing HPC systems. For example HPC computing clusters allow simulations to produce data so large, that it becomes uneconomic to store its entirety. Instead, data analysis and scientific visualization are performed *in-situ* -that is while the simulation data is still in memory- and only the (significantly smaller) analysis data (e.g. images) is stored. Here, the quality of the automated visualization is fundamental, as the largest portion of simulation data has to be discarded without human interaction. At the same time, such scenarios present the analysis with unprecedented input data sizes

breaking the feasible performance limits of conventional algorithms that are often designed for sequential or shared memory systems.

Adapting implementations to HPC clusters often requires a complete reformulation of the algorithms and is typically performed individually for each use case and meticulously tailored for specific hardware systems. The use of the emerging task-parallel programming paradigm proves promising for the creation of more generalized, portable and interoperable implementations.

The Visualization Toolkit (VTK) [Sch+06] provides a framework for scientific visualization that allows to build user specified pipelines from individual algorithmic modules. This model finds wide spread use in the scientific community and is a great support to scientific work of different fields. However, its applicability to HPC clusters is limited as most general purpose modules have no corresponding implementation for such systems.

This led to work in our research group that aims to create a similar visualization pipeline framework targeted towards high performance computing clusters with the use of a task-parallel programming paradigm. Multiple modules have been created in collaboration with the author and collegial work on the framework is ongoing at the time of writing.

During this work it became apparent, that the concept of *topological simplification* would become a prerequisite for further work. Topological analysis and simplification is often the first module of visualization pipelines, as noise and complexity of the data need to be reduced to produce understandable images [EJ09; Pas+11; Hei+16; Tie18]. Especially for the large and complex data sets that are the object of investigation for the scientific work we want to support, this aspect appears fundamental.

This leads us to the main contributions of this thesis. We lay the groundwork for a framework for task-parallel visualization and data analysis pipeline management targeted at HPC cluster hardware. To this end we introduce a novel, task-parallel and distributed Merge Tree construction algorithm, which required to completely revisit Merge Tree construction and derive novel insights into properties of inner nodes of Merge Trees. We also provide a module for topological simplification based on the Merge Tree that is applicable for HPC and in-situ scenarios. To increase flexibility in such scenarios -where iterative human interaction and prior domain knowledge might not be applicable- we propose novel parameters to guide topological simplification in a more predictable manner. The use of these parameters

requires a self-correcting, statistical estimation, and we explore different possibilities of its realization and the resulting accuracy.

1.2 Thesis Structure

Chapter 1

A brief motivation describes the scientific work we want to support and establishes task-parallel, distributed modules for Merge Tree construction and topological simplification as our main contributions.

Chapter 2

A detailed introduction to parallel programming is given to establish the terms task-parallel and distributed. The utilized parallel framework HPX is introduced and complemented with what we learned regarding the corresponding design principles and algorithmic properties of this setting.

Chapter 3

A detailed, theoretical introduction to Contour Trees, Merge Trees and topological simplification is given. We extend this established field by a modified definition for Contour Trees, that allows us to unify different aspects of Contour Trees that emerged in related work. This foundation is important to understand both the motivation and realization of our work.

Chapter 4

A unified perspective on Contour Tree construction is established and extended by a novel insight that guides the construction method of this thesis. Additionally, an extensive survey and comparison of contemporary Contour Tree construction methods is given. This also highlights that the construction method of this thesis contributes to the state of the art by filling a gap in contemporary construction capabilities.

Chapter 5

The novel, task-parallel and distributed Merge Tree construction method is presented. Building on both previous chapters, the reader should be able to understand the complex algorithm in detail. We discuss implementation details and optimization opportunities along with the necessary adaption of the algorithm to distributed settings.

Chapter 6

The simplification module based on the Merge Trees and the necessary algorithms to employ alternative, more flexible parameters for this simplification are introduced. Setting a percentage p or absolute number N of tree edges to keep after simplification allows for a more direct control of the simplification effects without prior domain knowledge.

Chapter 7

The results of benchmarks and evaluations performed on the modules are presented and strengths and limitations of the novel algorithms are discussed. A concluding summary is followed by an integration of this thesis into ongoing collegial work and suggestions for future work.

2 Parallel Programming with HPX

The scientific work we want to support requires us to adapt algorithms to allow for efficient and scalable parallel and distributed execution. Of course, this is a rather open problem statement. In this section, the used terms for and relevant aspects of both the conceptual and hardware setting targeted by our algorithms are introduced to adequately describe the problem. On the other hand, the utilized parallel paradigm and software framework, along with associated design goals and principles and common pitfalls and limitations are introduced to adequately describe the chosen solution.

2.1 The Parallel Setting

The term concurrency refers to portions of a process not being limited to a certain order of execution or mutual exclusivity in simultaneous execution. Such portions are often called independent from each other. A concurrent process or algorithm has at least some concurrently independent portions. A concurrent problem is a problem that can be solved by a concurrent process. In contrast, a problem or algorithm that exposes no concurrency is called sequential.

In reality, problems and algorithms are neither purely sequential nor consist solely of concurrent portions. Concurrency and sequentiality are therefore often used as comparative, sometimes quantifiable properties. The full extent of concurrent potential of a problem is hard to grasp mathematically and many gradual improvements of parallel programs in the literature came from the identification of more and more concurrent aspects in the problem.

Not all concurrent portions of such processes need to be pairwise independent. If a problem can be solved by solving two completely independent, purely sequential problems, it would still be a concurrent problem. Nonetheless, one of the most common abstractions for concurrent algorithms, called Amdahl's law [Amd07] models them as a purely sequential portion and a portion that is fully concurrent.

This model optimistically assumes the complete mutual concurrency of the latter portion.

The term parallelism refers to portions of processes actually happening simultaneously. Concurrency can be seen as the conceptual possibility for parallelism. Parallel programming therefore is concerned with assessing the concurrency in a problem, deriving an algorithm that exposes that concurrency and deriving a parallel implementation that realizes the concurrency. In reality, it is practically impossible to perfectly realize concurrency. Realizations of parallel concepts in hardware often require the introduction of extra work growing with the number of parallel workers.

We will refer to the maximal number of parallel workers a hardware system can execute simultaneously as *hardware threads*. To utilize effects such as CPU instruction pipelining, the operating system of most hardware systems obfuscates the number of hardware threads and instead exposes a (typically larger) number of *operating system threads (OS-threads)*. The programmer typically has no control over this process and assumes the number of OS-threads to be the number of parallel workers available to the program.

2.2 Task-Parallel Paradigm

A conventional form of parallelism realizes concurrency with respect to the OS-threads. Parallel portions of the program are exposed by specifying *software threads*, with each thread being a distinct sequential process. The number of these software threads is often based on -if not identical- to the number of OS-threads. We call this form of parallelism *thread-parallel*. In contrast, *task-parallelism* bases the parallel portions of the program primarily on the concurrent portions of the algorithm ("tasks") with little regard for the number of available OS-threads. Instead of leaving the allocation of software threads onto OS-threads to the operating system (often aiming for a one to one association), tasks are allocated, suspended and overall managed by a runtime scheduler (often running a thread-parallel back-end).

The term task-parallel has been used in the literature to represent complete parallel programming paradigms. These paradigms include additional commitments to parallel approaches and priorities. The work in this thesis is based on the task-parallel programming paradigm ParalleX and the realizing framework HPX [KBS09]. To understand the different aspects that form this paradigm, we will introduce additional classification of parallel approaches.

Parallel *granularity* is a term that describes the average size (often measured in runtime), or inversely the total number of parallel portions of the program like threads or tasks. A fine granularity refers to many, small portions of the process that are individually modelled and allowed to run in parallel. A coarse granularity refers to few, large portions of the process that partition the total work in parallel regions.

Threads are often comprised of rather coarse-grained portions of work that might be concurrent among themselves. While this discards this internal concurrency, parallel efficiency does not necessarily suffer, since there is a limit for parallelism already given by the number of hardware threads. Thread-parallel programs typically try to minimize the number of context-switches, that is the number of re-allocations of software-threads to OS-threads, along with other sources for parallel overheads.

On the contrary, task-parallel approaches often aim for fine-grained tasks, to expose as much concurrency as possible. The driving design goal is avoiding restrictions of the algorithm and problem concurrency. However, efficiency of this approach is highly dependent on allocation and context-switch costs of tasks to OS-threads.

Synchronous parallelism makes the overall program follow a sequential outline, with parallel regions creating and then terminating threads (*fork & join*) before continuing to the next step after a global barrier. In *asynchronous* parallelism, threads or tasks can be created and terminated at any time, often within some nested hierarchy. The use of global barriers is replaced by local synchronization means for pre-defined objects or between specific threads or tasks.

The communication and thus synchronization between threads or tasks can follow one of three principles. If all parallel workers have access to the same *shared memory* they can communicate information by side effects. Writing and reading on the same memory locations transfers information between threads, but requires synchronization, for example by atomic operation, mutex or similar. If this form of communication is not provided or desired, communication can take the form of explicit *message passing* [93]. For this, each communication participator is assigned a rank. Information and data can be sent by direct messages to a target rank. Synchronization takes the form of active or passive waiting for messages. Lastly, the communication between threads can be limited to parameters and return values of object oriented *method invocation*. For this, objects and callable methods have to be made addressable in a similar way to the ranks of message passing, which often requires some kind of registration with a communication manager before the communication.

Another important distinction in parallelization approaches can be made with respect to the data the algorithm is performed on. A program is called *data-parallel*, if threads perform similar work on different (portions of the) data. At an extreme, data parallel programming is the SIMD model [FR96] of vector processing or GPUs. However, in a wider sense a program is often called data parallel, if a clear distribution onto disjoint portions of data guides the design of the exposed parallelism. This is often done to minimize the performance cost of the communication and synchronization methods described above.

The prevalent parallel programming paradigm for CPU based systems is a coarse-grained, synchronous thread-parallel approach. It is associated with a data-parallel fork & join (divide & conquer) approach. In contrast, the task-parallel programming paradigm facilitates a fine-grained, asynchronous approach. Also, it is often not easy to mix task-parallel and data-parallel principles and let both problem concurrency and data segmentation guide the parallel structure.

The task-parallel paradigm can expose a number of benefits over the conventional approach. Forked threads may have different runtimes due to *load imbalance* between them. This, and external latencies like memory or disk lookup or network communication can lead to hardware threads being idle, while most software threads are waiting. With finer granularity and asynchronous parallelism, such waiting times can be filled with other work. Another aspect is the closer relation of the conventional thread-parallel approach to hardware capabilities. The task-parallel paradigm makes far less assumptions about the available number and nature of hardware threads. The demanding responsibility of thread allocation is separated from the program design and programmers can rely on the quality of the runtime scheduler provided with task-parallel frameworks. Both aspects are of increasing importance with respect to growing complexity and size of contemporary high performance computing hardware.

2.3 Distributed Systems

A distributed memory system is a parallel system where not all parallel workers share the same memory. More specifically, when talking about distributed systems in this thesis, we refer to the setting of computer clusters for high performance computing. In this setting, a number of parallel workers that do have a shared memory are grouped together to form a *locality*. This concept is often called *node*, which we avoid because of the danger of confusing them with graph nodes. These

localities typically are multi-core linux computers in the cluster, with parallel workers associated with OS-threads. The computers are connected by a network layer and can thus communicate. Multiple localities are involved in the execution of a parallel program, distributing the total memory among them.

Local parallel workers can communicate by side effects on shared memory. Communication across localities however has to be done explicitly, by message passing or remote method invocation. In an attempt to treat all communication consistently, such a distributed system can of course emulate a purely non-shared memory, by dividing local memory among local workers and forming virtual localities. Otherwise, the parallel memory setting is often called *hybrid* parallel programming, as shared memory parallelism techniques are combined with explicit communication.

In this setting, data-parallelism typically is a driving factor, regardless of used communication and parallelization techniques. Moving large amounts of data or passing large quantities of network traffic comes with heavy hardware latencies, that can easily become a dominating runtime influence compared to CPU cycles and shared memory communication. Minimizing the amount of communication between localities can often be achieved by moving work to the data, rather than the other way around. Algorithms that expose no parallelism over distinct data regions, but rely strictly on sequential work flows with global data dependencies often need to be reformulated in a more data-parallel way for such settings, if possible.

The task-parallel paradigm is difficult to combine with data-parallel requirements, as an algorithm has to be formulated to expose work that is primarily concurrent over data regions while not sacrificing too much possible concurrency of the problem. Such design is driven by the question what fine-grained concurrent work can be done as locally contained as possible to perform the overall algorithm. This has some striking similarities to ad-hoc network or agent-based algorithms. If successfully applied, the fine-grained task-parallel paradigm can be especially beneficial to performance and scalability in these settings, as the unavoidable network latencies that some tasks will encounter can be filled with work of other tasks.

2.4 Task-Parallel Programming with HPX

They key element of task-parallel programming in HPX is creating asynchronous tasks represented by c++ methods and passively waiting for their return values. At the center of this process is the *future*. A future is created together with a task and represents the promise of its return value. The fulfillment of this future can be

queried, passively waited on or used as a dependency for further task creations in conjunction with other futures. This concept provides a wide variety of possibilities to control when and under which conditions to create a task and when and where to wait for or utilize its results.

Tasks may have side effects in shared memory, including the synchronization with *local control objects (LCOs)* like mutices, semaphores and events. However, the main tool for communication and synchronization between tasks (especially for distributed settings) is parameters of method invocations and the futures of the corresponding return values, see Figure 2.1. Futures and thus tasks are created in HPX with a call to the function *async*. This is a form of synchronization, as the system guarantees that work of a task can start only after the corresponding call to async. Without the use of LCOs, no further guarantees on the timing and synchronization of the work of a task is made. It might be suspended from and allocated to different OS-threads multiple times during its lifetime. It may run in parallel to the original task (that called async) and can even still run after the calling task terminated.

This is where the future comes into play. It allows to capture the fleeting chunk of work that is the task, by representing its end point. A call to *is_ready()* on a future returns true if the associated task has finished its work. A call to *get()* on a future returns the return value of the associated task. get() will block until the associated task is done and the return value is ready. With this, the lifetime of task is guaranteed to be between the call to async and the first time a get() on its future returns.

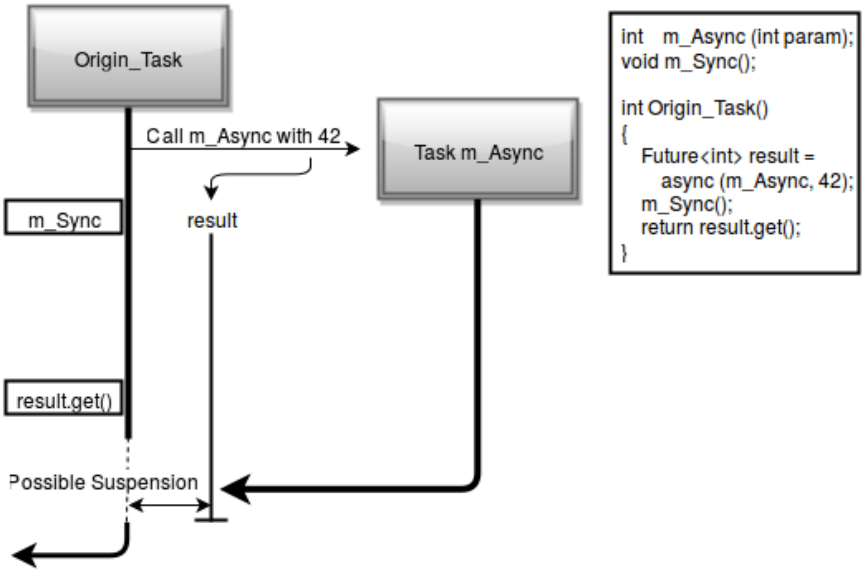


Fig. 2.1: Sequence diagram of a simple, direct use of a future. Both tasks can run in parallel, if two OS-threads are available and the HPX scheduler allocates them accordingly.

This principal functionality of tasks also exists in the c++ standard. Apart from providing a custom scheduler and powerful tools for hybrid programming, HPX extends the capabilities of futures. The async function has to be called from an active task to create a new task. It does so immediately, resulting in the allocation of resources (e.g. the stack) for the new task. Since dependencies between tasks (meaning the timings of futures to become ready) are unpredictable, a growing number of active but suspended tasks waiting for futures may arise.

The extensions of HPX allow the scheduling of tasks to be tied to futures becoming ready. The most basic function *then* allows to attach a task call to a future. A future to the attached task is returned immediately, but the actual scheduling of the task is delayed until the future it is attached to becomes ready. When_all and for_each are methods that allow for the composition and bundled treatment of futures in a similar way. The dataflow method is similar to then, but allows tasks to depend on multiple futures. The shared_future is a future that can be queried multiple times allowing arbitrary n to m dependencies between tasks finishing and starting their work. With this, the main method (and initial task) of an HPX application may consist solely of creating tasks and attaching tasks as continuations to the resulting futures. This allows to model entire algorithms as a *task graph*, mapping the concurrency of the algorithm at compile time and leaving the actual parallel realization to the scheduler, see Figure 2.2 .

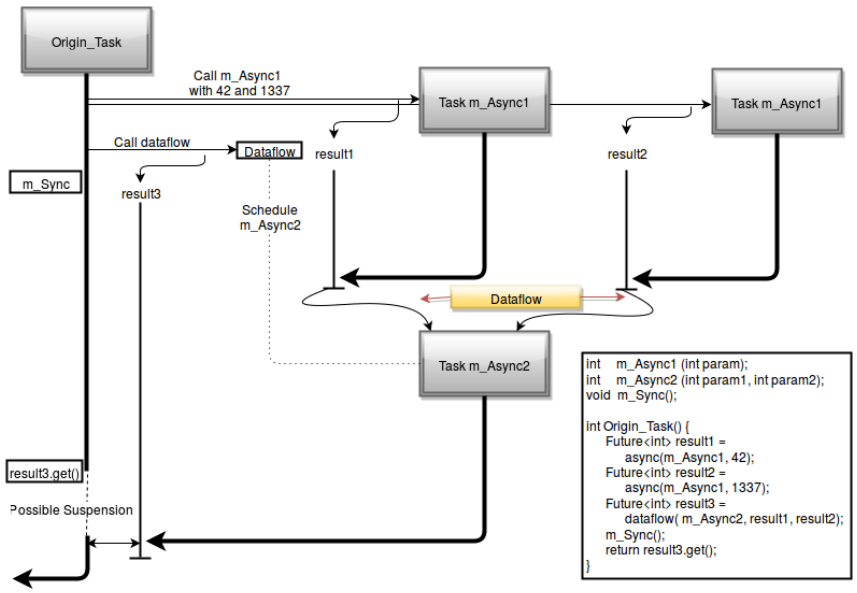


Fig. 2.2: Sequence diagram of a simple dataflow. The main task schedules two tasks and a continuation once both of them are done.

However, this conceptual method is hard to fully realize. Task dependencies often arise from intermediate results and the task graph has to be built on the fly. Even with the minimized context switch cost the HPX scheduler aims for, parallel granularity can only become so fine before scheduling overheads exceed actual work time per task. This means that not every synchronization dependency can be modelled by tasks and futures. Instead, conventional synchronization with LCOs becomes necessary, eroding the meaningfulness of the task graph. Lastly, leaving the details of parallelization entirely to the scheduler will not result in optimal performance. This conceptual responsibility of the scheduler had to be alleviated by manually optimized scheduling orders in the projects presented in this thesis to achieve optimal performance.

Under the hood, the HPX scheduler maintains queues for OS-threads to hold scheduled tasks. Tasks undergo a series of state changes. At some point, they become active, being allocated resources (including a fixed size stack) that stay fully occupied until the tasks eventually terminate. The nature and relation of the queues can be configured using scheduling policies, OS-thread pools and executors, ranging from FIFO queues and a round-robin OS-thread assignment of tasks to NUMA aware work stealing schemes on priority queues. Working with the system, it became clear that other configurable aspects like work stealing parameters, the maximal amount of simultaneously active tasks and stack sizes can have a dominant impact on performance. Especially memory consumption can reach extreme values if the number of active tasks is not limited and stack sizes are not carefully managed (reminiscent of GPU warp stack management). If the number of active tasks is limited however, one has to be careful not to create deadlocks. If all tasks that already have allocated resources wait for work by some tasks that have not yet allocated resources, the program cannot continue. Ultimately, the choices for these configurations fall to the programmer and may be based on assumptions about the hardware and algorithm.

Another limitation of this programming model that results in increased implementation complexity for the developer is termination handling. A strain of parallel computation is only guaranteed to have terminated once all involved futures are ready. To validate this, all involved futures have to be accumulated in a single control flow. In a distributed setting this often requires complex reference counting and heavy use of LCOs. This is an innate problem to the unpredictable order and timing of asynchronous parallelism, however it is particularly in conflict with design goals of the task-parallel paradigm.

2.5 Hybrid Programming with HPX

HPX supports task-based parallelism by asynchronously invoking methods and handling futures to their results. HPX also incorporates a solution to hybrid programming, managing multiple localities. This solution stayed as close to the governing principles of local parallelism as possible. Not any c++ method can be called by async right away. It has to be registered as an *action* using a macro at compile time. This registration also sets the action up for remote method invocation. A target locality can be specified with an additional parameter to async, creating the task for the action on the specified locality. The future is created on the calling locality which allows task dependencies and synchronization to span across localities.

To incorporate object oriented programming, the same mechanism can be applied for member functions of classes. For that, the class in question has to be registered as a *component* with a macro at compile time as well. Component instances are assigned global addresses similar to locality ranks and class member functions can be called remotely by specifying these addresses to async when calling component actions. Components are subject to garbage collection based on existing copies of their address and can also be actively migrated to other localities, with all existing copies of their address remaining valid. The mechanism that manages registration, lifetime and addresses of components is called active global address space (AGAS).

AGAS allows local and remote method invocations to be treated identically by the programmer. In fact, whether an action is called on a component that is in shared memory or resides on a different locality is not necessarily known at compile time. Dependencies of the task graph and the associated data can span across localities, as if working in one large shared memory. However, methods to evaluate whether a component is co-located are available and component migration is not automated. The degree of abstraction from hardware localities therefore lies in the hands of the programmer. The experience gained during the work on the algorithms in this thesis led us to a very limited use of AGAS capabilities. Often, only one component instance of an overall manager type class existed per locality with no component migration performed.

The main reason for this is that frequent data migration comes with harsh performance penalties and an efficient data-parallel design often eliminated the need for it completely. Additionally, interaction between local processes often can be optimized by the use of shared memory side effects and LCOs. Such optimizations are lost, if interactions are actually programmed to be agnostic of locality affiliation.

3 An Introduction to Data Analysis and Simplification with the Contour Tree

3.1 Domain and Scope

The scientific work we want to support concerns itself with scalar functions that represent natural phenomena. These may be for example temperature, density or even complex attributes reflected in some arithmetically achieved metric. We try to use topological features of these functions to allow for some semi-automated analysis of this data, like segmentation and simplification. To grasp these topological features we need some mathematical structures from algorithmic topology and Morse theory.

We consider the domain on which the scalar function we are interested in lives to be a 3-manifold M that is homeomorphic to the 3-sphere. This is a limitation to compact, (simply-)connected, not self-intersecting domains without boundary in 3D space. We consider the function itself to be a smooth, real-valued Morse (see below) function f on the domain M.

With these definitions come some assumptions about and limitations of the domain and attributes of the function. These limitations are necessary for two distinct reasons.

The first reason is to make it more easy to formally talk about the domain, data and algorithm. Topology is a very formal construct. The studied natural phenomena and especially the questions about them and the expected results of computational analysis are often far less formal. To map the measurements, simulations and insights of real world data to the topological structure of the Contour Tree we have to begin with continuous Morse theory, adapt to a piecewise-linear setting and ultimately formulate our algorithm input and output on graphs. The unavoidable ambivalence in this field manifests itself in the very name and definition of the Reeb Graph and Contour Tree. They are named graph and tree but defined as continuous quotient spaces.

On the other hand, data acquisition is often just as ambiguous, producing artifacts and noise. In most cases data sources are point based, so that the very topology of the data (e.g. regular grids vs. triangulations) is an arbitrary choice.

Mitigating some of this complexity therefore is the first reason to introduce assumptions and limitations in our formal description. This however is not a technical limitation and to our knowledge the presented algorithms may as well produce expected (although not necessarily formally correct) results for non-smooth, non-morse functions, higher dimensions, domains with boundary or even non-manifold domains.

Technical limitations exist however and form the second reason for the assumptions in the definition. The Contour Tree is a special case of the Reeb Graph. The latter may include cycles and thus is not only well-defined for manifolds that are homeomorphic to a sphere. This however requires special care during construction. For that, more complex and computationally expensive data structures and methods have to be introduced. Additionally, the concept of trunk skipping that can drastically reduce runtime is only applicable to Contour Trees. While the authors of [Gue+19] manage to partially replace that concept within their algorithm - providing the fastest Reeb Graph construction algorithm known to us at the time of writing - the achieved performance (measured in runtime per vertex) still falls short of their own Merge Tree variant of the same algorithm [Gue+17] by an order of magnitude.

The limitation to domains that are homeomorphic to spheres therefore is a limitation of our work to Contour Tree construction over Reeb Graph construction. This allows for a faster and more scalable algorithm. Most real world data sets, especially the prevalent scalar fields, do not include "holes". A conversion by "filling" holes with some base function values might produce desired results for additional data sets.

Morse Function The definition of a smooth Morse function is not crucial in our work, since we will be concerned with its piecewise linear equivalent introduced below. For the sake of completeness the brief definition will follow. A point p on M is *critical* if the differential df, p of f at p is zero. It is further non-degenerate if the matrix of second order partial derivatives (the Hessian) $H(p)$ is non-singular.

f is called a *Morse function*, if all critical points are non-degenerate and $f(p) \neq f(q)$ for all points $p \neq q$ that are critical.

3.2 Contour Trees as Quotient Spaces

Contour Trees can briefly be described as a contraction of each connected area of points with identical function values. With this, they are a tool that allows us to identify such areas that are "insignificant", to set such areas in a hierarchical context and to make batchwise operations on them instead of individual points.

Going further it is clear that we will need a notion for "points with identical function value". A *level set* is the pre-image $f^{-1}(h)$ of a given *level* $h \in \mathbb{R}$.

Level values have been associated with height and time in the literature and we will use terms like lower, higher, before and after as if talking about a rising level, that is as if considering level sets for increasing values for h over time. A popular image is the comparison to a rising water level. If the domain were subject to rain and "up" and "down" regarding the water flow would be defined by the function values of f, then a level set would be a straight water surface at a given time (and corresponding height h).

Note that for Morse functions, a level set includes at most one critical point.

A *sub-level set* $f^{-1}_{-\infty}(h)$ is the pre-image of the interval $(-\infty, h]$ and a *sur-level set* $f^{-1}_{+\infty}(h)$ is the pre-image for the interval $[h, \infty)$ respectively. Intuitively, the sub-level set is everything under water and the sur-level set is everything dry (ignoring the falling rain itself).

When restricting the domain M of f to a level set, each connected component in the restricted domain is called a *contour*. We denote the contour that contains a point p by $f^{-1}(h)_p$. Note that the contours form a partition of M and thus $f^{-1}(h)_p$ is unique.

One can define the equivalence relation $\sim$ on points $p_1, p_2 \in M$ as: $p_1 \sim p_2 \Leftrightarrow p_2 \in f^{-1}(f(p_1))_{p_1}$. That is two points are equivalent if they belong to the same contour.

The *Reeb Graph* of M w.r.t f is the quotient space $R(f) = M/\sim$. For our chosen attributes of M, $R(f)$ will be loop-free and thus is called *Contour Tree*. With this, we have arrived at the brief description above. However,we need two more structures that are easier-to-construct, intermediate steps for actual Contour Tree construction.

The *Join Tree* of M is the quotient space $J(f) = M/\sim_J$, with $p_1 \sim_J p_2 \Leftrightarrow f(p_1) = f(p_2) \wedge p_2 \in f^{-1}_{-\infty}(f(p_1))_{p_1}$. Similarily, the *Split Tree* of M is $S(f) = M/\sim_S$, with $p_1 \sim_S p_2 \Leftrightarrow f(p_1) = f(p_2) \wedge p_2 \in f^{-1}_{+\infty}(f(p_1))_{p_1}$.

Together, Join and Split Tree are called the *Merge Trees*. Intuitively, they contract all points with identical function values, that belong to the same body of water (and grotto of air respectively) just like the Contour Tree contracts contours.

3.3 PL Morse Theory and Contour Trees as Graphs

Contour Trees are defined as a quotient space regarding continuous functions and domains. Data produced, measured, simulated and used in the scientific work we want to support might not always represent continuous phenomena however. More importantly, data representations for computational analysis are not continuous and general infinite precision is technically impossible. This requires us to find discrete analogies for the above definitions.

Let K be a simplical complex that triangulates M. In this discrete setting, we consider continuous, real-valued Morse functions f, that are piecewise-linear on K. That means f is linear when restricted to any simplex of K. Such functions are now uniquely defined by their function values on vertices of K [Rou72], as other values can be obtained by linear interpolation. This piecewise-linear (PL) setting fits well with the typically point based data from real world measurements and simulations.

However, the PL setting requires an adjusted definition of critical points (and thus Morse functions), as a piecewise-linear f cannot be smooth w.l.o.g. and with that, the differential and Hessian are not well defined.

PL Morse Function As mentioned above, the piecewise-linear equivalent of Morse functions is fundamental for our work. Critical points have a close relation to nodes of the Contour Tree and the PL definitions for them will be starting point and anchor to many considerations regarding tree construction.

Critical points are defined by the function behaviour in the immediate vicinity of the point. The *star* $St(v)$ of a vertex $v \in K$ is the set of all simplices that contain v as a face. The *link* $Lk(v)$ of a vertex $v \in K$ is the set of all faces of simplices in $St(v)$ that are disjoint from v. The *lower link* $\underline{Lk}(v)$ respectively contains all simplices in $Lk(v)$ for which v is the point with the highest value.

We will follow the definition of critical points in the piecewise-linear context given in [Ede+03]. It is based on the reduced Betti numbers of lower links of vertices in K. A proper introduction of Betti numbers is not profitable here, but it suffices to

observe the following. $\tilde{\beta}_{-1}$ is 1 for empty lower links. $\tilde{\beta}_0$ is the number of connected components of the lower link. $\tilde{\beta}_1$ is the genus of and $\tilde{\beta}_2$ the number of enclosed voids in the lower link.

	$\tilde{\beta}_{-1}$	$\tilde{\beta}_0$	$\tilde{\beta}_1$	$\tilde{\beta}_2$
regular	0	0	0	0
minimum	1	0	0	0
1-saddle	0	1	0	0
2-saddle	0	0	1	0
maximum	0	0	0	1

In other words: A regular point has a topologically trivial smaller valued immediate vicinity. A minimum has no smaller valued vicinity. A maximum tears a void into its smaller valued (thus entire) vicinity. A 1-saddle has two connected components in its smaller valued vicinity. Lastly a 2-saddle has two connected components in its larger valued vicinity, which together with the saddle itself form a tunnel through the smaller valued vicinity. Any other configuration of Betti numbers constitutes a degenerate critical point.

With this, the definition of a PL Morse function can be worded just like in the continuous case. f is called a *PL Morse function*, if all critical points are non-degenerate and $f(p) \neq f(q)$ for all points $p \neq q$ that are critical. Note that critical points are always at vertices of K.

Since the definitions of level sets, sub-level sets and sur-level sets transfer to the PL setting well, Contour Trees and Merge Trees can be defined as quotient spaces on K instead of M accordingly. However, they are still quotient spaces and not graphs, much less trees.

The prevalent definition of Contour Trees [CSA03] as graphs associates nodes with points in the quotient space where the number of connected components changes with respect to a small change of the level h. In this paper however, we will use another property of those points for a simpler, formal transition to graphs: Points at which the number of connected components changes have a local neighborhood that is not topologically equivalent to the interval (0,1). Intuitively, they do not lie on the interior of a line in the quotient space, but at line endings or junctions.

We introduce the skeleton graph $Sk(K)$ of a simplical complex K, that quite trivially represents the 1-simplical skeleton of K. For each 0-simplex v in K there is an associated node $\tilde{v}$ in $Sk(K)$. Two nodes $\tilde{u}, \tilde{v}$ share an edge in $Sk(K)$, iff there exists a 1-simplex in K with u and v as its faces.

The Contour Tree as a graph $\tilde{R}(f)$ is the skeleton graph of a triangulation $K(R(f))$ of the Contour Tree as a quotient space $R(f)$. This definition covers an infinite amount of different graphs, one for each possible triangulation. And in fact, graphs have been called Contour Trees in the literature for different triangulations. Triangulations were always limited to contain only vertices for levels with vertices in the original domain. Sometimes, the limitation was tightened to contain only vertices for levels with Morse critical vertices. In this paper we will refer to such forms as non-canonical Contour Trees. The canonical Contour Tree of this thesis stems from a triangulation $K(R(f))$ that has a minimal number of 0-simplices. This does not allow the graph to contain vertices with degree 2 and coincides with the prevalent intuition described above. Definitions for Merge Tree graphs follow similarly from their respective quotient spaces.

It is important to note, that a skeleton graph is always a graph in the strict sense. That is, the edges form a set of tuples of vertices. However, the Reeb Graph as a quotient space may contain circular structures and thus can only be represented by a multigraph. That is, the edges form a set of distinct symbols and an incidence function maps those to tuples of vertices.

The definition above therefore relies on a loop-free quotient space and thus is applicable only for Contour Trees. When applied to Reeb Graph quotient spaces that are not also Contour Trees, triangulations (even with minimal simplex count) may introduce "auxiliary nodes" that have a degree of two and are not part of the Reeb Graph's conventional definition. However, with an additional step, all nodes of degree two can be removed, instead introducing a new edge between their neighbors. This may produce double edges and necessitates a switch to the multigraph model, but produces the conventional Reeb Graph.

Sometimes Contour Tree and Merge Tree graphs are represented as directed graphs, with edges being assigned a direction based on the function values. In this thesis they will go from small to large values for Join and Contour Trees and from large to small values for Split Trees. This makes Join and Split Trees directed rooted in-trees. That is, all edges are pointing towards the root.

Now we have defined the Contour Tree -our algorithm output- as a structure that can be computationally represented, a graph. Let us do the same with the input. Remember that PL functions are uniquely defined by their function values on vertices. Also, we chose M to be homeomorphic to a sphere and so is its triangulation K, which makes it uniquely defined by its 0- and 1-simplices (if all faces of a higher order simplex exist, the simplex exists). This allows us to represent all relevant

information of K and f within the skeleton graph $Sk(K)$ of K, by assigning the nodes $\tilde{v}$ scalar values corresponding to $f(v)$.

This graph representation of the input saves us the work of actually computing Betti numbers for lower links and instead observe the following properties of critical points.

Let the lower/upper link for scalar-valued nodes v in a Graph G be a subgraph of G that contains all smaller/larger valued neighbors of v and all edges in G between those neighbors. Consider the node $\tilde{m}$ in $Sk(K)$ for a vertex m in K.

m is a minimum, iff the lower link of $\tilde{m}$ is empty. m is a 1-saddle, iff the lower link of $\tilde{m}$ consists of two connected components. Similarily m is a maximum, iff the upper link of $\tilde{m}$ is empty and m is a 2-saddle, iff the upper link of $\tilde{m}$ consists of two connected components. m is regular, iff both the upper and lower link of $\tilde{m}$ consist of one connected component. m is a degenerate critical point if either the upper or lower link of $\tilde{m}$ consist of more than two connected components.

There is a number of nested relations between Contour Tree graph nodes and points on K. Going forward we will facilitate a notation, that allows us to shortcut these nested relations and to identify related entities with each other.

A graph node $\tilde{v}$, in a Contour Tree $\tilde{R(f)}$ corresponds to a point $\hat{v}$ on the respective quotient space $R(f)$. This point represents an equivalence class $[v]$ that corresponds to $f^{-1}(f(v))_v$. v could be chosen arbitrarily from any point in that equivalence class. For ease of notation however, we will always choose v to be the only such point, that is critical, if it exists. Remember, that such a v will always be a vertex in K.

All graph nodes $\tilde{v}$ correspond to quotient space points $\hat{v}$ that in turn correspond to level set components $[v]$ which include a critical point v of K. However, the opposite direction is not guaranteed. All points p in K belong to a level set component $[p]$ and are represented by a quotient space point $\hat{p}$, but not for all of them exists a graph node $\tilde{p}$. This is even the case if $[p] = [v]$ for some critical v (e.g. p itself is critical).

More intuitively, every Contour Tree node $\tilde{v}$ "is" a critical point v in the domain, but not every critical point (let alone regular points) "appears in" the graph. The relations for Join and Split Tree nodes are similar. More specifically, every node in a Join Tree is either a minimum or a 1-saddle or the global maximum. Similarly, every node in a Split Tree is either a maximum or a 2-saddle or the global mimimum.

Augmented Contour Tree The above description and definition of canonical Contour Trees as graphs does not assign a graph node to every vertex v (much less every point p) in K (not even to every critical one). While this reduction of represented information is part of the abstraction that the Contour Tree is used for, the lost information is still needed for segmentation, simplification and other global modifications of K.

The term augmented Contour Tree has been defined [V P03] as a (here non-canonical) representation of the tree, with all Morse critical vertices in the domain being assigned a graph node in the tree. This guarantees that all contours represented by one tree edge share the same topology w.r.t. Betti numbers. These Betti numbers were computed and added as meta information (augmented) to the tree edges. Because of this property, this representation was later termed the Contour Topology Tree [Chi+05].

We use the term augmented to describe a Contour Tree with all vertices of the input being assigned (augmented) to a tree edge like [CSA03]. This augmented Contour Tree encodes the information of "where in the tree" each vertex v lies. From this, the Betti numbers can be computed in post processing and additional applications based on the implied data segmentation become available. This tree is often represented by another (here non-canonical) representation of the tree, with all vertices in the domain being assigned a graph node in the tree.

In this paper we will define the augmented Contour Tree as a tuple of the (canonical) Contour Tree and a second structure called augmentation (of the tree). The augmentation is a map that assigns each vertex v in K to an edge of the (canonical) Contour Tree, if there exists no $\tilde{v}$ for v in the tree. This edge corresponds to the 1-simplex in the triangulation of the Reeb Graph space that contains $\hat{v}$. Similarily, Merge Trees can be tupled with an augmentation to form Augmented Merge Trees.

3.4 Applications of the Contour Tree in Visual Analysis

As described above, the Contour Tree forms a kind of skeleton of a function on geometry. This high level representation has a wide variety of applications for (semi-)automated data analysis and modification. A central capability of the contour tree is to grasp advanced features and often extract previously unknown points of interest in large and complex, possibly time-variant data.

This allows for interactive and hierarchical visual exploration of data and scientific results [Wid+12; Bre+11; Ros+17]. The Augmented Contour Tree additionally allows associating these topological features with a meaningful segmentation of the data, allowing for additional visual exploration and clustering techniques [WBP07], like automated and sped up volume rendering [Web+06; BG15].

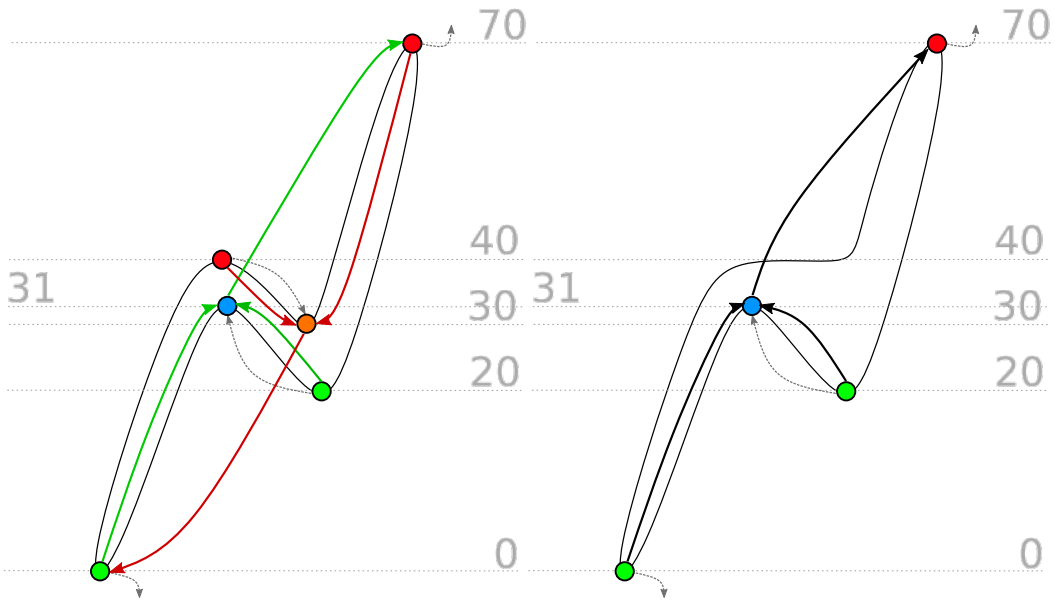
Grasping topological patterns also allows for similarity estimation, pattern matching, finding periodic features and data retrieval [Hil+01; TS05; SSW14; TN14]. It is also often at the heart of continuative data analysis and manipulation like surface parametrization for texture mapping [ZMT05], deformation and animation [TVD06], isosurface seed set extraction [Kre+97] and lastly compression and level of detail control [Sol+18].

The essence of most of these applications is to highlight, extract, determine, maintain or compare what is important and fundamental in the data, while fading, discarding or manipulating what is not important or mutable. As such, almost all of them contain an element of simplification. We want to support scientific work in the form of pipelines made from modular algorithms and so we focus on this aspect directly. The major application of Contour Trees discussed in this thesis therefore is general simplification of scalar data.

3.5 Topological Simplification and the Contour Tree

A very well-defined approach to topological simplification of scalar functions is based on persistence pairs [ELZ02]. A persistence pair is a pair of either a minimum and a 1-saddle, or a maximum and a 2-saddle. The persistence of such a pair is the distance in function value between both points.

From the perspective of Merge Trees, persistence pairs can be defined as pairs of graph nodes, with each node appearing in exactly one pair. Following the directed graph representation, a minimum is paired with the first of its descendants s in the tree, with which it forms a persistence pair with minimal persistence (compared to other minima that s is a decendant of). This leaves the global minimum and global maximum without a pairing. When pairing the global minimum and maximum with each other (and ignoring that they are strictly not a persistence pair), the Merge Trees can be represented as hierarchically nested persistence pairs called *branches* by a so called *branch decomposition* [PCS05].



(a) (b)

Fig. 3.1: (a) Exemplary domain with scalar function set to the height function, indicated by dotted lines. Dotted arrows indicate persistence pairs. Colored arrows indicate join and split tree of domain. (b) $\varepsilon - simplification$ of the domain for $\varepsilon = 10$ and the resulting Contour Tree (identical to Join Tree).

The pairs can be visualized in the so called persistence diagram [CEH07] and form the basis for the *ε-simplification* [EMP06]. It allows for the simplification of the scalar function f, creating a simplified f' by eliminating persistence pairs. To be more precise, persistence pairs are cancelled, meaning that both critical points do not exist in f' while all remaining persistence pairings remain unchanged (though their persistence might have changed). This is achieved by adjusting function values of regions around saddle points, until the paired minimum or maximum is no longer critical, see Figure 3.1. Given a value ε to drive the simplification, the process gives two strong guarantees:

1. all and only those persistence pairs with a persistence of up to ε will be cancelled.

2. the maximal distance in function value (and persistence of remaining pairs) before and after the simplification of a point p is ε: $|f(p) - f'(p)| \leq \varepsilon, \forall p \in \sigma \in K$.

However, it is possible to cancel the same persistence pairs with a smaller maximal impact on function values [BLW12]. This is made possible by changing function values in regions around both saddle and extremum of a persistence pair towards

each other. Since it is not possible to cancel any more persistence pairs while holding the guarantee for a maximal function value change, this simplification is called optimal by the authors. It gives the following guarantees:

1. all and only those persistence pairs with a persistence of up to 2ε will be cancelled.

2. the maximal distance in function value before and after the simplification of a point p is ε: $|f(p) - f'(p)| \leq \varepsilon, \forall p \in \sigma \in K$.

Both of these simplification methods are rather sequential and global by nature. The "movement" of saddles and regions around them modifies function values of points that are augmented to more than one tree edge. Additionally, the simultaneous treatment of 1-saddle-minimum pairs and 2-saddle-maximum pairs (and thus Join Tree branches and Split Tree branches) requires some knowledge about both structures before the function can safely be modified. These requirements do not fit the targeted, distributed memory and parallel setting of high performance computing clusters well. Instead,semi-local operations that can safely be performed without knowledge about the complete data set are preferred.

We therefore focused on another simplification method, that allows the cancellation of persistence pairs by moving only regions around extrema towards their saddles [TP12]. This approach generally allows the elimination of arbitrarily chosen extrema, with some constraints, and is therefore called general by the authors. With this, nested branches may be chosen to persist, while their containing, more presistent branch extrema would be eliminated. This is not a direct cancellation of persistence pairs, as it possibly recombines the remaining pairings. It still eliminates the extremum and saddle and reduces the number of persistence pairs. However, we will limit our application of this technique to cancel persistence pairs based on persistence and ε. This also avoids the necessity to perform more than one iteration of the algorithm.

General simplification limits the function change for each cancellation to the augmentation of the Merge Tree edge that the cancelled extremum belongs to. Additionally, all cancellations of 1-saddle-minimum pairs and 2-saddle-maximum pairs happen in separated phases. This allows for the cancellation of a persistence pair without knowledge about persistence pairs that are not nested within the first. Additionally, it allows for one phase to be safely executed without knowledge about the persistence pairs of the other. With taking some care about avoiding f' to become non-Morse, these aspects allow for an efficient parallelization of the process [Luk+21] and will also be beneficial to our efforts of task-parallel and distributed simplification.

However, the isolated phases weaken the formal guarantees of general simplification, if based on persistence. From the original function f to the fully simplified function f' an intermediate function $\hat{f}$ is produced after the first phase. Due to the changes in persistence of persistence pairs inherent to all simplification techniques, some but not all persistence pairs with a persistence greater or equal ε in f might have a persistence smaller than ε in $\hat{f}$ and thus be cancelled in f'. The formal guarantees given by general simplification (based on ε persistence and one iteration) are therefore as follows:

1. all persistence pairs with a persistence of up to ε will be cancelled.
2. only persistence pairs with a persistence of up to 2ε might be cancelled.
3. the maximal distance in function value before and after the simplification of a point p is ε: $|f(p) - f'(p)| \leq \varepsilon, \forall p \in \sigma \in K$.

For a given ε it might therefore cancel more persistence pairs than *ε-simplification*, of course up to the upper limit of optimal simplification. It is also important to note, that even if the exact same set of persistence pairs is cancelled by all three methods (e.g. if no pairs with a persistence p with $\varepsilon < p \leq 2\varepsilon$ exist), the resulting functions f' will generally not be identical.

All of the above mentioned simplification methods do not compute the Merge Trees or Contour Tree completely. However, they do contain work that partially overlaps with Merge Tree construction. One might only be interested in the Merge Trees or Contour Tree of the simplified function f' and try to reach it by simplifying the trees of f accordingly, without having to actually compute f' completely. This can be achieved by *symbolic* simplification of the Merge Tree of f, that is the desired Merge Tree graph edges and nodes are simply discarded to obtain the Merge Tree of f'.

One method for symbolic Merge Tree simplification is to compute its Branch Decomposition and discard all branches that correspond to persistence pairs with a persistence less than ε. This results in the same Merge Tree as computing the Merge Tree on a simplified function f' obtained by the *ε-simplification* with ε or a different function f'' obtained by the optimal simplification with $\varepsilon/2$.

Another method for symbolic simplification can be achieved by identifying Y-shapes that include leafs in the tree and pruning the leaf edges [TTF04]. This approach can be adapted to degenerate multi-saddle scenarios [CSP10]. Since branches with minimal persistence in the tree always coincide with leaf edges, both these methods produce the same simplified Merge Tree as above.

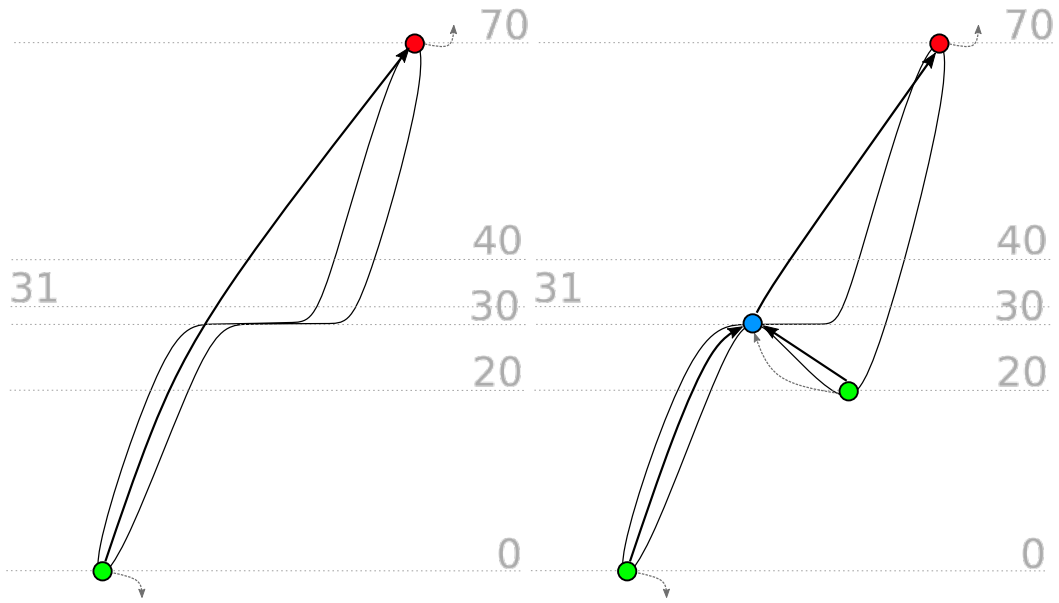


Fig. 3.2: (a) Optimal simplification for $\varepsilon = 10$ of the domain given in Figure 3.1. (b) Result after the first phase (2-saddle-maximum) of generalized simplification of the same domain for $\varepsilon = 10$. Note that the persistence of the remaining pair dropped to 10. After the second phase the result will be similar to the result of optimal simplification.

It has to be noted however, that the Contour Tree created by symbolic simplification based on the Branch Decomposition may differ from the tree created by symbolic simplification based on Y-shapes and both of them differ again from the actual Contour Tree of f' (which is also the Contour Tree of f'') see Figure 3.3.

Additionally, the function f''' obtained by one iteration of generalized simplification based on persistence is different from f' and f'', has Merge Trees that are subtrees of the above and thus generally has yet another different Contour Tree. The Merge Trees and Contour Tree of f''' can not be obtained by symbolic simplification of a non-augmented Merge Tree or Contour Tree of f given only ε, since the effect phase one has on phase two cannot be reproduced from this data.

In contrast to symbolic simplification, the simplification of augmented Merge Trees or Contour Trees can also be *applied back* onto the underlying scalar function. This can speed up topological scalar function simplification if an augmented tree structure is already available, as persistence pairs and corresponding regions do not have to be computed again. In general, there are two different methods for applying the pruning of a tree edge to the scalar function. One is to "carve" a bridge between the pruned extremum and the other parent node of its saddle. This involves modifying

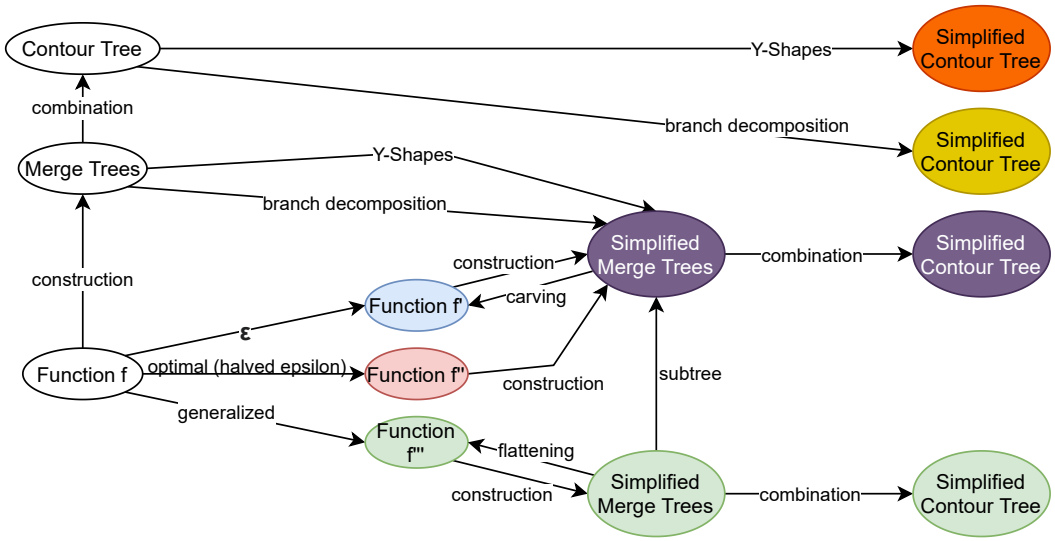


Fig. 3.3: An overview of different simplification methods for scalar functions, Merge and Contour Trees. Note, that differently colored functions and trees are not identical in general. Only the green Merge and Contour Trees could be made identical to the purple variants, if the generalized simplification is taken with care and effects of the first phase on the second phase are actively avoided.

function values in regions around the saddle just like in the $\varepsilon - simplification$. Applying this method in conjunction with symbolic simplification that would result in the Merge Tree of f' actually modifies the scalar function to become f'. Again, no such consistency can be found in Contour Tree simplification, see Figure 3.4.

The other method is to "flatten" regions around the pruned extremum towards its saddle [Luk+21]. This performs the same function value modifications as done in a single phase of the generalized simplification. In fact, when first simplifying the augmented Join Tree and applying the prunings by flattening the resulting scalar function is the same intermediate result as produced by generalized simplification when starting with the 1-saddle-minimum phase. Simplifying the augmented Split Tree of this intermediate result and applying the prunings by flattening will result in f'''. As discussed above, generalized simplification has some beneficial properties for our application. Therefore, for the purpose of topological simplification in this thesis, we chose to implement the two-step process of flattening Merge Tree simplifications described above.

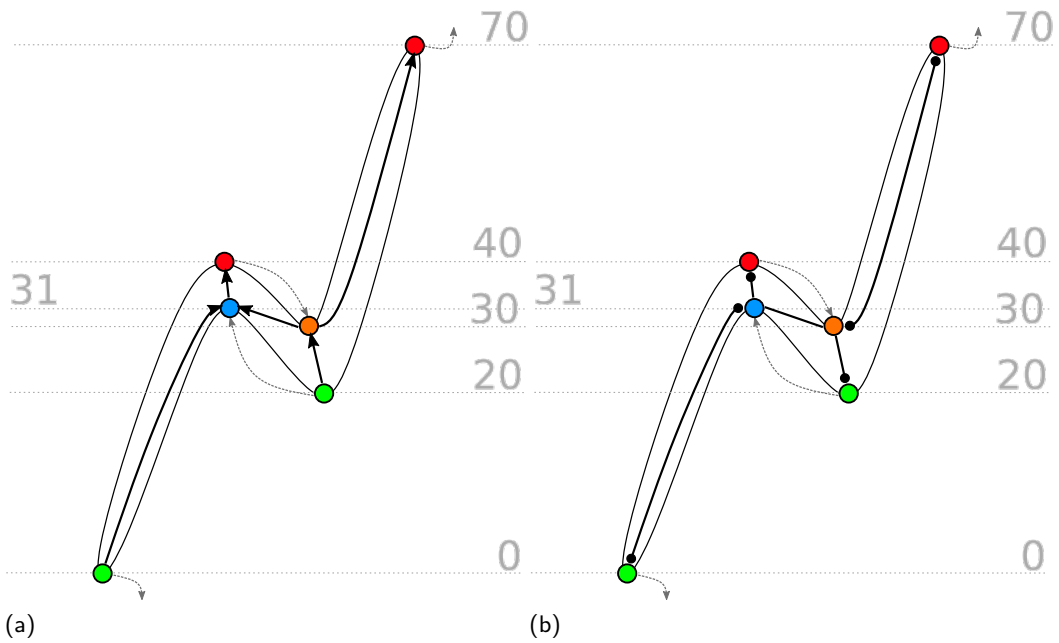


Fig. 3.4: (a) Contour Tree of the domain given in Figure 3.1. (b) Branch Decomposition of that Contour Tree. Note that the branches do not coincide with persistence pairs. Branch based tree simplification will leave the tree unchanged for $\varepsilon < 20$. Y-shape based tree simplification will leave the tree unchanged for $\varepsilon < 30$.

4 Contour Tree Construction

In recent years, multiple Contour Tree construction methods emerged in the literature. Techniques vary in whether they compute augmentations, target distributed memory settings and are task-parallel or data-parallel or expose massive parallelism suited for SMP or GPU architectures. Our work is unique in this field, as it demonstrates the best performance and scalability for augmented Merge Tree construction on distributed systems at the time of writing.

In this section, a survey of 11 such techniques that emerged over the last 2 decades (primarily over the last 5 years) presents a representative overview of the field of contour tree construction and its development to highlight innate attributes of the underlying problem. We chose a chronological approach, as reoccurring concepts, data structures, observations and process steps will make it easier to describe contemporary work, that has typically become rather complex. This will also help with introducing the algorithm of this thesis.

It is hard to compare the capabilities and performance of all methods. Hardware capabilities increased over the years, targeted hardware systems range from multi core CPU to GPU to distributed, hybrid systems. Additionally, contour tree construction is always output sensitive (for some algorithms this has a larger impact then for others) so the complexity of the data sets can drastically impact performance. Although some data sets became popular for direct comparisons, not all methods benchmarked on the same data sets. We therefore use a rough approximative metric to give an overview of the order of magnitude of performance: processed vertices per second. That is the number of vertices in the data set divided by the reported ideal total runtime in seconds for that data set. This ideal runtime may come from any number of parallel workers and any hardware configuration. Since some papers only reported times for Contour Tree construction, we doubled the calculated amount of vertices per second to compare against Merge Tree construction. This is an unfair comparison to some degree, as in a sequential setting the overhead of the actual combination step is neglected and in a perfectly parallel setting the construction of both Merge Trees does not take any longer than the construction of one. However,a factor of 2 in this data-set agnostic metric should not be interpreted as significant prove for superior performance anyway.

Let us first explore the similarities and common ground of all techniques, to be able to focus on the characterizing differences. All presented Contour Tree construction methods compute the Contour Tree by first constructing both Merge Trees and combining them. The combination of the Merge Trees is unaltered since its first description [CSA03], except for recently developed partial parallelizations [AN15; Gue+17; Car+16b]. It is typically not the bottleneck for runtime or memory consumption and augmentation treatment is trivial. Additionally, the computation of one Merge Tree is perfectly analogous and embarrassingly parallel to the other. Going forward, we will therefore only be concerned with Join Tree construction, refering to minima and 1-saddles. Split Tree construction follows symmetrically in an identical manner and Contour Tree construction from both Merge Trees is well understood. No contributions to this combination step are made in this thesis.

The identification of leafs in the tree is also common ground for all presented methods. All Morse critical minima can be identified by their immediate local neighborhood and have a 1 to 1 correspondence to Join Tree leafs. What remains is the identification of inner nodes and edges of the Join Tree. So far, no parallelization scheme was able to identify inner node edges, before all ancestors in the tree have been identified. In other words Join Trees are always constructed bottom up to some degree. It is possible, that this is an innate sequential requirement of the problem and no concurrency can ever be exposed here.

Considering this common ground, all presented methods are therefore primarily characterized by the process (and the nature of its parallelization) of finding the Merge Tree adjacent saddle for an extremum. The development of algorithms in this field therefore also was a development of observations about the properties of the saddle for a given extremum. These observations were approached from a multitude of strongly different perspectives. To help with understanding the different methods -foremost the one presented in this thesis-, related work will be viewed from one consistent, formal perspective for the saddle identification: the perspective of monotone paths. This is neither a limitation nor an extension to observations made in the literature, but rather a unification of underlying principles, although it is in no way canonical and other valid formulations of the introduced relationships exist.

4.1 Saddle Identification with Monotone Paths

Formally, the problem of saddle identification is to find for each local minimum m a vertex s_m, such that there exists an edge $(\tilde{m}, \tilde{s}_m)$ in the Join Tree. Let $V_{\min}$ denote the set of all local minima in M. We call a path p monotone ascending w.r.t. the scalar function f if $f(p(a)) \leq f(p(b))$ for all $0 \leq a < b \leq 1$. We denote the set of such paths by P^+. Symmetrically, we define P^- as the set of monotone descending paths. We call a path p monotone if $p \in P^+ \cup P^-$.

With this, we want to define the set of all vertices reachable from a local minimum $m \in V_{\min}$ through a monotone ascending path as:

$$\mathrm{Up}(m) \coloneqq \left\{ v \in V(M) : \exists p \in P^+ \text{ s.t. } p(0) = m, p(1) = v \right\}$$

Lemma 1. For a join node $\tilde{v}$ that is a descendant in the Join Tree to a leaf $\tilde{m}$ one has $v \in \mathrm{Up}(m)$.

Proof. For the proof, it is sufficient to show that for each child $\tilde{u}$ and parent $\tilde{v}$ in the join tree there exists a path $p \in P^+$ from u to v. This is already shown in the literature (e.g. Lemma 9 in [Chi+05]). The claim then follows through a transitivity argument. □

This criterion is combined with the criterion, that Join Tree nodes are always Morse critical points to directly motivate the algorithmic structure of [Chi+05].

Now consider the set of *saddle candidates* of a local minimum $m \in V_{\min}$ as the set:

$$\mathrm{Sc}(m) \coloneqq \bigcup_{\substack{n \in V_{\min} \\ n \neq m}} (\mathrm{Up}(m) \cap \mathrm{Up}(n))$$

In other words, a saddle candidate for a local minimum m is a vertex that is reachable by a monotone path from m and another local minimum in M. The motivation behind the chosen name *saddle candidates* becomes clear when considering the following lemma:

Lemma 2. s_m is a saddle candidate for m, i.e. $s_m \in \mathrm{Sc}(m)$.

Proof. From the definition of s_m follows $s_m \in \mathrm{Up}(m)$ due to Lemma 1. Furthermore, $\tilde{s}_m$ has to be connected to at least one other leaf $\tilde{n}$. Since a leaf in the join tree corresponds to a local minimum $n \in V_{\min}$ one has $s_m \in \mathrm{Up}(n)$ again due to Lemma 1 and therefore $s_m \in \mathrm{Sc}(m)$. □

Lemma 3. The vertex s_m is the smallest valued saddle candidate of m. That means $s_m = \arg\min_{v \in \mathrm{Sc}(m)} f(v)$.

Proof. We need to show that every saddle candidate's function value provides an upper bound for $f(s_m)$. Then, the claim follows from Lemma 2. Let $v \in \mathrm{Sc}(m)$ be an arbitrary saddle candidate of m. Then, there exist monotone ascending paths to v from m and at least one additional local minimum n. The entirety of these paths is in $f^{-1}_{-\infty}(f(v))_v$. This especially means that m and n are connected in $f^{-1}_{-\infty}(f(v))_v$ and therefore $f^{-1}_{-\infty}(f(v))_m = f^{-1}_{-\infty}(f(v))_n$. However, the initial connected components of m and n were disjoint i.e. $f^{-1}_{-\infty}(F)_m \cap f^{-1}_{-\infty}(F)_n = \emptyset$, where $F \coloneqq \max\{f(m), f(n)\}$. Since both of these components are contained in $f^{-1}_{-\infty}(f(v))_v$, they must have joined by then; leaving the estimate $F < f(s_m) \leq f(v)$. □

In other words, the smallest valued vertex that is reachable through monotone paths from m and at least one local minimum in M other than m, corresponds to the adjacent inner node for m in the join tree. This criterion is behind most massively parallel merge tree construction algorithms [RS14; RTP18; Car+19] that trace the entirety of $\mathrm{Sc}(m)$ and find its minimum. Other approaches avoid the tracing of the entire set $\mathrm{Sc}(m)$ by ordering progression based on function value. The first saddle candidate encountered by strictly ascending progression from minima is their actual saddle [Gue+17]. The conventional, fully sequential solution [CSA03] and its divide & conquer based parallelizations [V P03; Lan+14; Gue+16] also rely on this property.

From here, an additional observation allows us to search for saddle candidates without the need for ordered progression and without tracing the entirety of $\mathrm{Sc}(m)$ for all minima.

We define

$$\mathrm{Ex}(m) \coloneqq \mathrm{Up}(m) \setminus \mathrm{Sc}(m)$$

for $m \in V_{\min}$ as the set of vertices that are exclusively reachable through a monotone path from m. We write $v_1 \leftrightarrow v_2$ for vertices $v_1, v_2 \in M$, if there exists a 1-simplex

in K being the convex hull of these vertices. That is, v_1 and v_2 are *adjacent* to each other in the skeleton graph of K. With this, let furthermore

$$\mathrm{Bd}(m) \coloneqq \{v \in M \setminus \mathrm{Ex}(m) : \exists v' \in \mathrm{Ex}(m) : v' \leftrightarrow v \text{ in } M\}$$

be the set of vertices forming a boundary around $\mathrm{Ex}(m)$. Regarding this set, consider the following properties.

Lemma 4. The set $\mathrm{Bd}(m)$ is a subset of $\mathrm{Sc}(m)$.

Proof. Let $v \in \mathrm{Bd}(m)$. By definition $v \notin \mathrm{Ex}(m)$ which is only possible if either $v \notin \mathrm{Up}(m)$ or if $v \in \mathrm{Sc}(m)$. Thus, we have to rule out the first case by proving $v \in \mathrm{Up}(m)$. Again due to the definition of $\mathrm{Bd}(m)$, there exists $v' \in \mathrm{Ex}(m)$ such that $v \leftrightarrow v'$. If $f(v') < f(v)$, the vertex v is reachable from m by a monotone ascending path through v'; therefore, $v \in \mathrm{Up}(m)$. The remaining case can be ruled out by contradiction. Assume that $v \notin \mathrm{Up}(m)$ and $f(v') \geq f(v)$. Note that then, there has to exist at least one local minimum $n \in V_{\min}, n \neq m$, such that $v \in \mathrm{Up}(n)$. This becomes clear when considering the following construction: by successively choosing adjacent vertices in M with decreasing function values, one eventually ends up in such a local minimum n. By traversing the involved 1-simplices in reverse order, one thus obtains a monotone ascending path from n to v, proving $v \in \mathrm{Up}(n)$. Since v' was chosen adjacent to v and has a larger function value by assumption, v' is reachable by a monotone path through v from n, thus $v' \in \mathrm{Up}(n)$. This however contradicts $v' \in \mathrm{Ex}(m)$. With this contradiction we prove $f(v') < f(v)$, thus $v \in \mathrm{Up}(m)$ and finally $v \in \mathrm{Sc}(m)$. □

Lemma 5. s_m is in the boundary set of m, i.e. $s_m \in \mathrm{Bd}(m)$.

Proof. Because of Lemma 2, s_m is reachable from m through at least one monotone ascending path. Let p denote one such path. Since $m \in \mathrm{Ex}(m)$ and $s_m \notin \mathrm{Ex}(m)$, there has to exist a vertex $v \notin \mathrm{Ex}(m)$ on p that is adjacent to a vertex in $\mathrm{Ex}(m)$. Thus, $v \in \mathrm{Bd}(m)$ and therefore due to Lemma 4 $v \in \mathrm{Sc}(m)$. Because p is monotone ascending, the estimate $f(v) <= f(s_m)$ holds true and due to Lemma 3 we obtain $s_m = v \in \mathrm{Bd}(m)$. □

Lemma 4 and Lemma 5 together directly allow for a stronger variant of Lemma 3:

Lemma 6. The vertex s_m is the smallest valued vertex in the boundary set of m. That means $s_m = \underset{v \in \mathrm{Bd}(m)}{\arg\min} f(v)$.

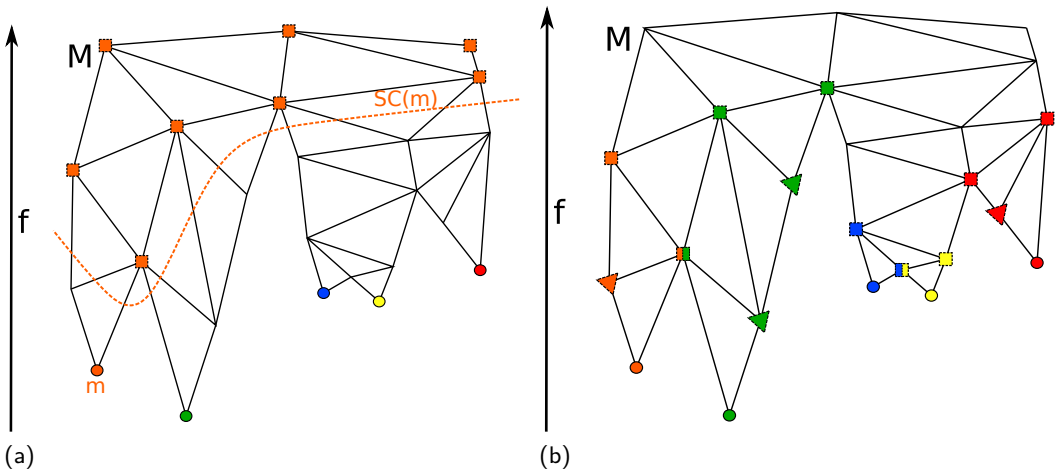


Fig. 4.1: (a) illustrates the saddle candidate set Sc for the leftmost local minimum with rectangles. Note that saddle candidate sets of local minima overlap and can span large portions of the domain. (b) illustrates the exclusively monotone reachable region set Ex for all local minima with triangles according to color. Note that those sets are mutually disjoint, connected and leave out large portions of the domain. Additionally, it illustrates the boundary sets Bd for all local minima with rectangles according to color. The smallest valued vertex in each such set is a saddle node in the Join Tree.

In other words, the set $\mathrm{Bd}(m)$ restricts the search for saddles to special saddle candidates that form a kind of hill ridge around the valley $\mathrm{Ex}(m)$ a minimum m lives in. As these valleys are mutually disjoint, the amount of double work and size of involved data regions is drastically reduced by this restriction, see Figure 4.1. To our knowledge, no other construction method has utilized, let alone formally introduced this tightening of saddle candidates.

4.2 Related Work Survey

4.2.1 Totally Ordered Construction

A general algorithm for computing the Contour Tree in all dimensions was introduced in 2003 [CSA03]. We will refer to this algorithm as the sequential, or totally ordered approach. This algorithm introduced the method to construct Contour Trees from both Merge Trees, which all relevant Contour Tree construction work known to the authors at the time of writing follow to date. Understanding this algorithm in detail is a fundamental basis for understanding other Contour Tree construction methods, including the one presented in this thesis.

Method The major data structure used in this algorithm is the union-find data structure, often also called disjoint-set. The structure has a close relation to Merge Tree construction and is found in most construction algorithms in some form. It represents a partition of a fixed set of elements into disjoint subsets. It is typically initialized with each element forming a one-element subset and allows for the two operations *union* and *find*. The union operation performed on two set elements merges the two subsets they belong to into one. The find operation performed on one set element returns the subset it belongs to. Some definitions of the data structure allow for a third operation that adds elements to the (then not fixed) underlying set.

The most common implementation of the disjoint-set data structure is an array based disjoint-set forest: An array of integer values, with each set element being associated with an index in the array. The stored integer values are themselves indices of the array and therefore act as pointers. Initially, each element points to itself, as in the array holds the first n natural numbers. A find operation for an element x follows the linked list of pointers starting at x in the array until an element points to itself. This element is returned as a representative for the subset x is in. Union operations make one of the representatives of the arguments point to the other. This represents subsets as trees in the array and makes representatives the roots of each tree. If the union operations carefully make the root of the smaller subset point to the root of the larger subset regardless of argument order and find operations compress the linked list of pointers leading to the root, so that each element along the way points to the root after the operation, the amortized runtime of both union and find operations is in $O(\alpha(n))$ with n the number of elements in the set and α the inverse ackermann function. The memory consumption of the structure amounts to one pointer per element (and one integer value for subset size management) and is thus linear in n if pointer sizes are fixed.

The totally ordered construction method originally computes the fully augmented Merge Trees as the non-canonical representation with one graph node per vertex mentioned in Section 3.3. We will describe a variant here, that computes the augmented Merge Tree as a tuple, like described in the same section. This algorithm maintains a disjoint-set data structure over the vertices of the input. All input vertices are ordered by function value and this ordered list is traversed in sequence. For each traversed vertex v, a union operation is performed with all smaller-valued neighbors. If all involved unions were trivial (as in the subsets were either already identical or one of the subsets only consisted of v) then v is added to the augmentation for the arc starting at the current representative of its subset. If however one (or more in a degenerate multi-saddle setting) union is non-trivial (as in the merged subsets

are not identical and both contain vertices that are not v), then Join-Tree edges are formed between v and both former representatives of the involved subsets. In this case, v becomes the new representative of the resulting subset and is augmented to itself (starting a new arc).

The algorithm closely follows the definition of the Join-Tree. After any given traversed vertex v the subsets managed by the disjoint-set structure correspond precisely to the sub-level set connected components for the level $h = f(v)$. The water level rising "over time" is associated with actual compute time by this algorithm, bodies of water are directly tracked by the disjoint-set structure and whenever to formerly disjoint bodies of water touch for the first time, the Join-Tree is updated.

It is this relation to intermediate events during traversal, that differentiates Merge Tree construction from the more simple counting of total connected components, often called union-find algorithm. The union-find algorithm only depends on the final state of the union-find data structure after all unions and can be parallelized disregarding intermediate states. This is why distributed union-find algorithms can not be applied in Merge Tree construction, where we need to know "when and where" non-trivial unions happen.

Although very close to the definition of Join-Trees, this sequential algorithm can already be viewed from the perspective of monotone paths. By the total ordered progression, minima are traversed before their neighbors. Every one-element trivial union can be seen as extending ascending paths from such a minimum. It is always the ascending path with the globally smallest valued reachable vertex that is extended next. These trivial union path extensions trace portions of $\mathrm{Ex}(m)$ for their respective minima m. At some point two such paths will meet, producing a non-trivial union at a vertex v. Since v is adjacent to vertices in $\mathrm{Ex}(m)$ but reachable by another ascending path it lies on a hill ridge and belongs to $\mathrm{Bd}(m)$. Due to the progression ordered by function value it is the smallest valued vertex with these properties and thus the saddle for m.

Here we can also observe how to find saddles for saddles. Vertices that are exclusively reachable by monotone paths from ancestors of v cannot lead to non-trivial unions and cannot be nodes in the Join-Tree, as all minima that can reach these vertices have already merged to the connected component of v. Imagine contracting the subtree of ancestors of v onto v. This can also be visualized by contracting the entire domain that is augmented to edges leading towards v, that is the entire sub-level set component (imagine the body of water) of v at the level $h = f(v)$. This contraction would make v a minimum, but leave the entire rest of the domain and Join Tree unchanged. v itself can therefore now be treated like a minimum and search for the

smallest valued vertex that is reachable by monotone paths from v and at least one local minimum that is not already merged with v. This recurring concept will later be called virtual contraction of saddles.

Profile The totally ordered construction of the Merge Trees allows for the construction of the canonical augmented Contour Tree in all dimensions and simple treatment of boundaries and degenerate multi-saddles. It is a general and fundamental solution to the problem and guided all later algorithm designs. The totally ordered construction of Merge Trees achieves a performance of 23 to 32 thousand vertices per second. The algorithm does not expose any concurrency and it is clear that the algorithm has been advanced in all aspects by more recent solutions.

4.2.2 Divide & Conquer

In the same year, building on the above algorithm, the authors of [V P03] adjusted the amount of tree nodes with degree 2 (reaching another non-canonical form) and the augmentation of the tree to explicitly represent Betti numbers. Additionally, they introduced a generalization of the formal Contour Tree construction on triangulations. Treating cell interiors as externally solvable black boxes, allows to abstract from function behaviour inside the cells and basically extended Contour Tree Construction to CW-complices. However, the most important contribution with respect to our survey is the first parallel Contour Tree construction.

Method The basic algorithm to compute the Merge Trees is basically identical to the totally ordered construction. The parallelization is a classic data-parallel divide & conquer approach: The data set is divided into segments of preferably equal size. For each segment, the totally ordered construction method (modified like mentioned above) is performed. The resulting Merge Trees of the segments can be combined to form the overall Merge Tree in a "conquer" stage. For this the totally ordered construction method is performed again, but using the segment Merge Trees as its domain (connectivity between trees following from the original domain).

Profile This first Divide & Conquer technique introduced some powerful formal extensions to the original algorithm. The parallel approach is methodically simple and while the paper assumed shared memory, a strong basis for distributed data handling is inert to the algorithm. The authors demonstrated a speedup per processor

of at least 62,5% at a maximum of 32 parallel workers. The largest utilized data set contained around 7 million vertices. However, combining segment Merge Trees introduces extra work over the sequential solution that is linear in their size. This combination work does not expose the same concurrency as the computation of the segment Merge Trees, which leaves available resources idle after the initial steps. Especially for very complex or noisy data, the size of the segment Merge Trees can become similar to the size of the underlying domain, which would make the final combination almost as expensive as the entire sequential construction. No results on absolute runtime were reported, but given the speedups they might go up to 900 thousand vertices per second. It is clear that the algorithm has been advanced in all aspects by more recent divide & conquer solutions.

4.2.3 Domain Restriction

The relation of monotone paths to contours was already known in 2003 [CS03]. We also refer to [CS03] for an overview of sequential and early work on contour trees of that time, which will not be analysed in detail here. The first algorithm to directly address this relation and use monotone path traversal for Merge Tree construction [Chi+05] was introduced in 2005.

Method Morse critical points can be identified by their local neighborhood. The observations about PL criteria for Morse critical points stated in 3.3 were actually introduced in [Chi+05]. These points can therefore be identified in a strictly data-parallel way (which would allow to process each vertex in parallel). After identification of all 1-saddles and minima, the totally ordered construction method is applied to a new domain, consisting only of these critical vertices.

Connectivity of the new domain is established on demand: When a critical point is traversed by the algorithm and tries to perform union operations with its smaller "neighbors", monotonely descending paths in the original domain are followed until reaching other vertices that exist in the new domain. The Join Tree of the new domain is the same as the Join Tree of the complete domain. This allows to skip most trivial union operations by restricting the totally ordered construction to be performed only on critical points and monotone paths between them.

Profile While the algorithm is a modified application of the purely sequential original, the input is reduced by an embarrassingly data-parallel pre-processing.

Real world data may be reduced by a factor less than 1% according to the authors. Noisy and complex data sets that were examined in this thesis however rather conform with a factor of one fourth to one third also encountered in some data sets by the authors. This resulted in a performance of 30 to 41 thousand vertices per second. The computations performed on this restricted domain may be parallelized by a divide & conquer approach just like the original algorithm. Given the speedups of [V P03] this may result in a performance of up to around 1 million vertices per second. Of course, this algorithm cannot compute the full augmentation. It depends on shared memory and sequential progression. However, it is worst case optimal w.r.t output size and depending on the data set may be the fastest construction method for single-thread systems if no augmentation is required, to date.

4.2.4 Minimum Lists

So far, all parallelization attempts were based on modified applications of the totally ordered construction method. The relation of monotone paths to saddles allow for parallel per-minimum approaches, like introduced in 2012 [MDN12]. When reading the paper, please be advised that it defines the Join Tree with sur-level sets, so that the terms Join Tree and Split Tree are swapped compared to this thesis.

Method Like with the domain restriction above, a fully data-parallel scan identifies Morse critical 1-saddles by their local neighborhood. In this paper, authors did utilize a GPU for this process. The algorithm uses the property we state in Lemma 1 above. From each 1-saddle s, arbitrary monotone descending paths are followed until reaching minima. This allows to find all minima that are ancestors of s in the Join Tree, if at least one such path is followed for each connected component of the lower link of s.

Once this step is performed, *minimum lists* containing all ancestor minima for each saddle and *path lists* containing all descendant 1-saddles for each minimum are available. The node $\hat{s}$ for the smallest valued path list entry s for a minimum m is connected to $\hat{m}$ in the Join Tree (more specifically the (non-canonical) Join Topology Tree, as all Morse critical 1-saddles are still contained in the lists). The edge $(\hat{s}, \hat{m})$ is added to the algorithm output. m is removed from the minimum list of s and replaced by s in all other minimum lists. Additionally, it adds its remaining path list to that of s (possibly creating it). Once a minimum list for a saddle becomes empty, we can perform virtual contraction on the saddle and it is treated like a minimum above.

This paper also introduces the recurring concept of *trunk skipping*. Each minimum processes remaining saddles in growing order of function value. If only one (virtually contracted) minimum is still performing work, the remaining path list can just be sorted by function value. The *trunk* of the Join Tree is a sequence of edges, basically forming a linked list of this path list.

Profile The procedure of minimum and path list discovery can be done embarrassingly parallel for each 1-saddle and each path which exposes a fine-grained parallelism. Considering the distributed memory adaptability of the algorithm, localities would have to send messages when following monotone paths outside of their data region. Since monotone paths can be extended to arbitrary smaller valued neighbors, a rapid back and forth of messages between localities following a zig-zag path could be avoided.

The procedure of minimum and path list processing that actually constructs the Join Tree is far more restricted in terms of parallelism. Each edge creation modifies multiple minimum and path lists with unpredictable access patterns, which requires shared memory. While edge creation can be parallelized over all (virtually contracted) minima, after each such creation round a global barrier is necessary to make minimum and path list changes securely visible. Since noisy and complex data sets can easily

Additionally, the saddle down-to minimum approach follows monotone paths along their entire length. The high valued 1-saddles will parse multiple paths through the entire function height, creating double work. For example, for a simple 1D data set with the height function forming just a large inverted V structure, the entire domain will be parsed similar to the totally ordered construction method.

The reported performance resulted in around 14,5 to 20,5 million vertices per second in parallel for 8 threads and the use of a GPU. The largest processed data set contained 16,7 million vertices. The algorithm requires shared memory and cannot produce an augmentation. Depending on the data set, it be the fastest construction method for single-thread systems if no augmentation is required, along with [Chi+05].

4.2.5 Local-Global Merge Trees

In 2013, the first algorithm specifically targeting distributed memory [MW14; MW13] revisited the conquering stages of the divide & conquer approach. The

paper also concerns itself with topological simplification based on the branch decomposition of Merge Trees and introduces the possibility to interleave simplification and construction. They also immediately identify a "chicken-and-egg" problem with choosing ε before having access to the topological insights of the Merge Trees. This recurring concept will be called on-the-fly simplification in this thesis.

Method The actual merge tree construction on each distributed data region is treated as a black box by this algorithm. The algorithm concerns itself with the combination scheme of segment Join Trees and introduces multiple improvements. The local-global Merge Tree is introduced. It is a sub-tree of the actual Join Tree, that contains only relevant nodes and edges for a given data segment. Relevant here means, that all and only those branches with an augmentation containing local vertices are represented. Branches not represented in the local-global sub-tree are called *ghost*.

Two observation about the local impact of sub-level set components are made: Firstly, a body of water that is completely surrounded by land within the local data segment cannot merge with any remote bodies of water anymore and thus its saddle is final. Secondly, only connectivity along monotone ascending paths coming from other data segments can have impacts on segment Join Trees while merging. Monotone paths that enter the local data segment through vertices that have smaller valued neighbors on the boundary cannot introduce any connectivity, that could not also be introduced by paths through those smaller valued neighbors.

The first observation allows to prune any branches that are completely contained in the data regions already represented by a tree, before merging the tree with the ones from other data regions. The second observation allows to ease the handling of domain connectivity for segment Join Tree merge processes, by tracking only vertices with no smaller valued neighbors on data boundaries and "stitching" segment Merge Trees together at those vertices.

Lastly, the algorithm does not gather the complete Join Tree on a single locality in a fan-in reduction manner like in [V P03]. Instead, all localities merge their trees with a partner locality and prune internal branches along with degree 2 nodes and vertices that are not relevant for stitching. Holding ghost branches they now have information about the united domain between them and their partner. In an iterative all-to-all communication new partners are found until all localities hold information about the entire domain and thus computed the complete local-global join tree for their original segment.

Profile The algorithm improved on the divide & conquer based distributed parallelization method. It allowed for the processing of 33 to 57 million vertices per second on up to 2048 (distributed) parallel workers. The maximal data set used contained 8.5 billion vertices, which is more than there are function values representable by 32-bit precision. It introduced a possibility to represent the Merge Tree allowing certain operations without gathering the entire tree on a single node, which is a unique capability to date. However, it does not provide an augmentation and for storage or certain operations, a gathered, complete representation of the Merge Tree may be necessary. Even for large, noisy and complex data used in the work presented in this survey, merge tree edge counts stay within a few millions. Even when using several hundred bytes of meta information per edge, contemporary memory limits offer multiple orders of magnitude more memory than is needed for tree storage. Nonetheless, the algorithm constitutes a benchmark against which new distributed solutions must compare.

4.2.6 Pruned Divide & Conquer

Shortly after, in 2014, another distributed Merge Tree construction method was introduced [Lan+14]. It is able to produce the augmented Merge Tree as described in this thesis: A tuple of the canonical Merge Tree and a mapping of vertices to edges. Again, Merge Tree computation on single data segments is a black box for the actual algorithm and the variant of the totally ordered construction described above is used in the paper.

Method Like in the local-global approach above, the conquering stage is modified to prune irrelevant information from segment Join Trees to optimize the merging process. This involves pruning branches that reside entirely in the already represented data region. The additional information that the augmentation provides allows to skip domain connectivity handling altogether. Instead, the data segment boundary vertices relevant for stitching are already augmented to the Join Tree branches affected by merge processes, which allows for a per-branch stitching process. In contrast to the local-global representation, the full merge tree is gathered by a k-way reduction fan-in among all localities. Additionally, they suggest, that stopping the process during this reduction produces Join Trees of larger and larger data regions with increasing fan-in steps. For some operations, these trees may suffice and further fan-in processes might be skipped.

Profile The comparison of this algorithm to [MW14] comes naturally. The authors demonstrated comparable runtime on a data set also benchmarked in [MW14] with a performance of 33 million vertices per second on the same number of parallel workers. However, they match this runtime while additionally providing the augmentation (called segmentation in the paper) and local gathering of the complete, canonical tree. This is made possible by using the additional information of the augmentation to their advantage. With this, the algorithm is the fastest and most scalable distributed construction method for augmented Merge Trees. However, the authors acknowledge that the algorithm runtime suffers from load imbalance typical for fan-in processes. The merging of segment Join Trees has a linear runtime in the size of the trees. Despite all pruning efforts, these sizes may be in the same order of magnitude as the entire data set for worst case noisy data.

4.2.7 Unordered Construction

In 2014, a rather theoretical paper [RS14] introduced an algorithm that emphasizes the semi-local nature of Merge Tree construction. The only sequentially ordered dependencies exist between ancestors and descendants in the tree (consider virtual saddle contraction). When reading the paper, please note that Join Trees again refer to sur-level sets and are called Split Trees in this thesis.

Method Arbitrary ascending monotone paths are followed from minima and each visited edge is "colored" by the color of that minimum. If an edge is already colored it is not visited and colored by different minima again. While some write-after-read synchronization efforts become necessary, this could potentially be done for all minima in parallel.

Once done, the colors partition the domain edges with each colored region for a color of m being a subset of (edges between vertices of) $\mathrm{Up}(m)$. Vertices that are adjacent to edges of different colors form boundaries between these regions and are per definition in $Sc(m)$ and $Sc(n)$. The arbitrary handling of precedence when coloring regions results in an arbitrary shape of this boundary. However, there is no "way around" the actual saddle of a minium: The smallest valued "color boundary vertex" is also the smallest valued saddle candidate and thus saddle of a minimum.

Now, all Morse critical points are identified, sorted by increasing function value and added to heaps of minima based on the colors of adjacent edges. These heaps basically correspond to the Path Lists of [MDN12] and allow for a bottom-up construction of the Join Tree.

Profile The method as described in the paper ultimately is similar to the Minimum Lists. Monotone paths are traced along their entire length (although in different direction compared to [MDN12]) to identify minimum-saddle relations. These relations are used to build the tree in a partially parallelizeable manner. Within the paper parallelization is not addressed. No implementation or performance benchmarks are presented.

However, the approach explores the relation between minima and their saddles in a way, that exposes high concurrency. The method of using monotone paths to partition the domain edges forming arbitrarily placed (color) region boundaries "catching" the saddle on this boundary as the smallest valued vertex will reoccur in later work. In fact, the approach of our thesis is very similar to this, but restricts the colored regions to $\mathrm{Ex}(m)$ to avoid a complete and unnecessary partitioning of the data.

4.2.8 Distributed Domain-Restriction

Both [MW14] and [Lan+14] present divide & conquer policies, with a black box performing the actual local Merge Tree construction for each data segment. In their respective papers, this black box was a realization of the totally ordered construction [Car+16b]. In 2015, a combination of the divide & conquer policy of [Lan+14] and the Join Tree construction kernel of [Chi+05] created a memory-efficient Join Tree construction [AN15].

Method The data is segmented based on an octree structure. One parallel worker constructs the Join Tree for each data segment based on the domain restriction solution explained above. Iterating between pruning and merging, all data segments merge their trees in a reduction based on the distributed augmented divide & conquer approach explained above. The overall method is very memory efficient, as both combined methods prune unnecessary information as early as possible.

Additionally, the paper introduces a partially parallelized method for combining the Merge Trees to form the Contour Tree. Instead of iterating over all Merge Tree edges in sequence, the algorithm allows for parallel treatment of all leafs. After a global barrier and virtual contraction of saddles, the resulting virtual leafs are treated in parallel.

Profile Since the domain restriction approach is faster than the totally ordered construction, this combination with the divide & conquer policy of Landge et al. is an expected improvement in terms of performance. The volvis.org vertebra data set used to compare [Lan+14] and [MW14] is used again and showed a speedup of around 3. Performance was between 21 and 50 million vertices per second. This is impressive, considering the experiments were limited to a shared memory machine with 64 parallel workers, as the focus of the data segmentation lied in memory efficiency and not actual distributed computation.

Since most other publications did not mention memory consumption, it is an interesting benchmark to see this arguably very memory efficient solution to consume e.g. 11 GB of memory during construction for an 8GB data set (compared to 60GB consumption using the method of [MDN12]). This demonstrates a large memory footprint of Merge Tree construction that for example limits the maximal size of processed data for GPU based approaches.

Of course the above approach cannot produce an augmentation (as it is based on [Chi+05]) and exposes little concurrency within data segments, limiting its use in hybrid distributed settings. When an augmentation is needed (which introduces additional memory consumption and disqualifies the memory efficient solution above) this memory footprint can even bring shared memory systems to their limits. This highlights the importance of a distributed, augmented Merge Tree construction like in [Lan+14].

4.2.9 Contour Forests

The latest work on divide & conquer based solutions presented here was introduced in 2016 [Gue+16]. Again, the actual Merge Tree construction is a black box and the totally ordered construction is used in the paper. This is also due to the fact, that no inherently parallel construction method for augmented Merge Trees existed at the time of writing. In this approach, the data segmentation is deliberately chosen to split in the function image space instead of domain space. This reduces work for the stitching operations.

Method To produce the necessary data segmentation all vertices are sorted by function value and this list is split into the desired number of segments. Each segment Contour Tree is computed with augmentation and in parallel. These segment Contour Trees are actually segments of the complete Contour Tree as

well. In other words, their interior edges are already final and they deviate only in artificial leaf nodes "dissecting" the tree. Instead of stitching arbitrary edges while merging the segment trees, artificial minima and maxima need to be matched by augmentation in adjacent segment trees and removed by stitching leaf edges.

Profile The initial sorting and segmentation of the data -or in other words the necessary control over the data distribution- makes this approach unfit for many distributed application scenarios where data distribution tends to be given. The actual speedup of the approach over domain space segmented divide & conquer approaches is very dependant on the data set, as the number of the artificial leaf nodes may rise to the same order of magnitude as the number of total vertices for a worst case. The authors demonstrate performance of 6 to 16 million vertices per second on up to 8 parallel workers (shared memory CPU system) with the largest (upsampled) data set containing 82 million vertices. While this was the fastest shared memory solution producing augmented trees at the time, it has since then been deprecated in favour of faster solutions.

4.2.10 Massively Parallel Peak Pruning

In 2017 [Car+16b] (and in greater detail in 2019 [Car+19]) a strictly data-parallel algorithm for Merge Tree construction was introduced. It fully realizes the concurrency that was glimpsed at in the unordered construction description above [RS14] and is the first parallel Merge Tree construction kernel (instead of divide & conquer policies) since the introduction of the (rather limited) Minimum-Lists.

The paper introduces some terms with similar names to the ones used here. For example the *governing saddle* for a minimum m corresponds to "the" saddle s_m, to differentiate it from Morse critical saddles adjacent to m in non-canonical Merge Trees like Topology Trees. The *saddle candidates* in the paper form subsets of the saddle candidates Sc introduced here.

Method Like described in the unordered construction above, the edges of the input are partitioned based on which minimum m they can form a monotone path to (they are adjacent to vertices in $\mathrm{Up}(m)$). Instead of following the monotone paths upwards from minima however, they are followed downwards from each vertex of the data set in parallel. This is done by pointing to an arbitrary smaller valued neighbor for each vertex. Then minima are found by iterative "pointer-jumping"

which basically corresponds to the path compression of Union-Find approaches. In fact the whole data structure corresponds to a disjoint set forest.

The regions $\mathrm{Ex}(m)$ are guaranteed to be incident to the edges in the partition of m, while the Sc of multiple minima are arbitrarily divided between them. A boundary of vertices separates these partitions. These partition boundary vertices are called saddle candidates in the paper and in fact are guaranteed to be saddle candidates (like defined here) for the minima of all involved partitions. They can be identified in parallel, by performing the find operation for each incident edge for each vertex.

Although partition boundaries are arbitrarily chosen from the saddle candidates by path precedence, it is guaranteed that the governing saddle is among them (and thus the smallest valued one). This is where the algorithm deviates from the ideas of [RS14] and actually realizes concurrency. For each minimum the governing saddle is found with a parallel sorting process. Edges that are incident to a saddle candidate are sorted by partition and then function value of that saddle candidate, identifying the smallest valued saddle candidate for each partition and thus minimum.

Next, the algorithm utilizes an interesting observation about Merge Tree edge augmentations. The augmentation for an edge from a minimum m to its governing saddle s_m is exactly the set of all vertices that are reachable through monotone paths from m with a smaller function value than s_m. With that it becomes a subset of $\mathrm{Ex}(m)$ which is guaranteed to be incident to the edges in the partition for m. The augmentation can therefore be collected, by adding all vertices that are incident to edges in the partition of m and have a function value smaller than s_m. This observation and "cutting" of considered regions at the saddle function value will reoccur in the algorithm presented in this paper.

This allows for the virtual saddle contraction, in fact it is not that virtual in this case. All vertices in the augmentation are deleted from the domain going further and all edges that are incident to exactly one deleted vertex are made incident with s_m instead. This process is called saddle pruning in the paper, giving the procedure its name.

Profile The method is a realization of the idea of treating each minimum-saddle edge in the Merge Tree individually in parallel. The concurrency of this approach is fully exploited by formulating region growth and minimum saddle candidate search as operations over the complete domain with one up to one parallel worker per edge of the input. This of course introduces extra work, as not all edges have to be sorted or even visited to find the saddle. Monotone paths are again traced along

their entire length creating a complete partitioning of the domain. The approach therefore lends itself well for massively parallel hardware like GPUs. Additionally, operations like pointer jumping and edge sorting heavily rely on fast communication between parallel workers and thus shared memory. Performance is reported only including an optimization that is based on domain restriction like explained above [Chi+05], which makes it impossible to construct the full augmentation of the tree. For augmented Merge Tree construction the algorithm did not terminate within 24 hours of runtime for data sets that otherwise took less than a minute to compute. The reported Performance is between 3 and 47 million vertices per second with a maximum of 64 parallel workers and the largest data set containing over 1 billion vertices. Performing this on a GPU instead resulted in an additional speedup, which makes this and similar approaches [RTP18] the fastest method for Merge Tree construction, if the limited memory of GPU is sufficient.

4.2.11 Locally ordered Task-Parallelism

Simultaneously, a task-parallel augmented construction method was introduced, that also highlights the individual treatment of each minimum-saddle edge [Gue+17]. Instead of exposing massive parallelism over all edges, one parallel task performs the saddle search for each minimum.

Method A parallel scan over the data identifies all minima by their local neighborhood. Then, one parallel task for each minimum starts to grow regions around the minima. The region growth is similar to breadth first or depth first searches, but instead of using a FIFO-queue or LIFO-stack the growth is governed by a smallest valued vertex first out priority queue (realized by a fibonacci heap to allow for faster merging later on).

In other words, starting at the minimum the region always grows to the smallest valued vertex adjacent to any region-vertex. Region membership of vertices is tracked by a disjoint set forest with minima representing connected components. In fact, this process performs the exact same union find operations as the totally ordered construction would. The only difference being, that the trivial, one-element union operations of different connected components are performed in parallel. From the perspective of monotone paths, the totally ordered construction always only follows the ascending monotone path with the globally smallest valued next-to-visit vertex, while the locally ordered construction follows for each minimum in parallel

one ascending monotone path with the locally smallest valued next-to-visit vertex respectively.

Just like in the totally ordered construction, this process guarantees that the regions around minima (or connected components of sub-level sets) reach the governing saddle s_m at some point. The problem is, that for totally ordered progression, this saddle can be identified by performing a non-trivial union. This is the case, because it is guaranteed to be adjacent to two vertices that have already been processed and belong to different regions. At first glance, this criterion requires some synchronization between region growth tasks to "meet" at saddles.

However, as discussed multiple times above, finding saddles ultimately is a local operation. The region growth for a minimum m is terminated, once a vertex v is visited, that has smaller-valued, un-visited neighbors. These smaller valued neighbors cannot belong to $\mathrm{Ex}(m)$, because all monotone paths leading to them from m would have already be considered by the strictly ordered region growth before v and thus they could not be un-visited at this point. Since v is reachable by a monotone path from m, but also adjacent to a vertex not in $\mathrm{Ex}(m)$ and additionally the smallest valued vertex with these attributes, it is the saddle s_m.

In other words, the region growths perform the same trivial unions as the totally ordered progression and once a vertex is reached that could potentially perform a non-trivial union, it is guaranteed to do so and is the searched saddle. Additionally, the vertices that form the region when encountering the saddle corresponds exactly to the augmentation of the edge between the minimum and the saddle. This is because these vertices are reachable by monotone paths from m and are smaller valued than s_m.

Virtual saddle contraction however does require some synchronization. The terminated minimum finalizes its augmentation and stores its fibonacci heap tied to the saddle. For each task terminating this way, the saddle is tested for contractability, by performing find operations on all smaller valued neighbors. If all neighbors belong to a finalized augmentation, then all involved minima have arrived and the saddle can be contracted. This is done by merging all stored fibonacci heaps and updating the union find data set accordingly.

With a global counting of active tasks, trunk skipping can be performed once only one task is active. This is because the one region that is still growing will sweep through the entire unvisited domain in sequence and collect dangling saddles in order of their function values. Trunk skipping can typically speed up construction by an order of magnitude.

Profile This elegant dissection of the totally ordered construction allows a fine-grained task-parallel construction of augmented Merge Trees. Presented performance was between 10 and 93 million vertices per second for (resampled) data sets of 16 million vertices on up to 32 OS-threads. This makes it the fastest augmented Merge Tree construction when limited to shared memory to date. As the default implementation in the TTK [Tie+17], it is also arguably the most widely used solution at the time of writing.

Of course, concurrency is limited by the number of Merge Tree edges. If a small number of regions (but more than one) contain large portions of the data in their augmentation the region growth becomes rather sequential and expensive. Additionally, adaption to distributed settings is difficult, as each region can span multiple data segments but can only perform work on one locality at a time, back and forth communication may dramatically hinder performance.

4.2.12 Other works

A quantized approximation of the Contour Tree can be constructed by a data-parallel algorithm [Car+16a]. They define interval level sets as the union of level-sets for levels between $n * q$ and $(n + 1) * q$ with q as the quantization fidelity parameter. Interval contours are connected components of interval level sets and can be used as the equivalence relation basis like contours for Contour Trees. These "Interval Contour" Trees are a quantized approximation of Contour Trees and can be constructed by taking fragments with an image space sampling rate of q. These fragments are subjected to a data-parallel union-find algorithm to construct the Contour Tree directly, without the use of Merge Trees.

The algorithm produces an augmentation with respect to fragments instead of vertices. It is not dependant on shared memory and distributed benchmarks have been performed. Performance was below 20 million vertices per second for 2D data sets of 23 million vertices on up to 256 parallel workers (16 localities). The quantization of course introduces large portions of extra work and memory consumption while lossless approximation cannot be guaranteed a priori. The gained concurrency may cancel out this extra work on massively parallel systems like GPUs, but this approach is harshly limited by memory constraints. For these reasons, the authors state they shifted their focus towards the massively parallel peak pruning method described above.

There is a reformulation of a classic divide & conquer solution [Nat+16] for the massively parallel communication model MPC. Actual Merge Tree construction per

data segment is black boxed and suggested to be done with the totally ordered construction. Data segmentation is based on the MPC model with a recursive cuboid structure. The merging of two segment trees is done like in [V P03] with some pruning ideas like in [MW14] or [Lan+14]. No implementation or performance benchmarks are demonstrated.

Lastly we want to mention an adaption of Contour Tree construction to ad-hoc sensor networks [Sar+08]. In this hardware setting each vertex in the data is a reading from a sensor that also forms a parallel worker. Edges are formed by the network topology of the sensor computers, so communication is only possible among neighbors. This requires a strictly data-parallel approach with no global data structures. As early as 2008, the paper introduced a monotone path based, decentralized and data-parallel solution. Each sensor can identify its vertex to be a local minimum and start a region growing procedure. A sensor is visited by a region growth if all smaller valued neighbors have been visited by the growth. The region growth therefore follows monotone ascending paths in arbitrary order. These regions will in turn create a partition of the data similar to those in [RS14] and [Car+16b]. Sensors that find neighbors visited by different minima are again partition boundary vertices and thus saddle candidates (called potential merge saddle in the paper). Their smallest valued member is determined by an elegant broadcasting poll and saddle contraction can continue. Of course performance in this setting is not comparable to other settings. The paper is not cited in any other work described here (likely because of the network setting being published in a different community) and similar ideas emerged here almost a decade later.

4.2.13 Conclusion and Comparison

The history of Merge and Contour Tree construction has let to a diverse field of methods. Some of these methods became obsolete, but most are still the most performant solution for at least some hardware or requirement setting. To complete our overview, we therefore refer to Table 4.1. Only methods with implementations and reported runtimes are listed. The table is sorted into two types: actual Merge Tree construction methods (kernels) and divide & conquer distribution policies (policy) that can be combined with any kernel. The parallel hardware column differentiates between data-parallel solutions for massive amounts of parallel workers (oversimplified as GPU), multi core CPU targets with shared memory and HPC clusters or other distributed settings. The columns do not account for theoretically possible adaptions of the general approaches, but rather the actually utilized settings for the reported benchmarks. Figure 4.2 shows a flow-chart to decide on the best algorithm to use,

	Year	Parallel Hardware	Augmentation	Performance
Kernels				
Totally Ordered	2003	Sequential	Yes	0.02 - 0.03
Domain Restriction	2005	Sequential + GPU	No	0.03 - 0.04
Minimum Lists	2012	Shared Memory + GPU	No	14 - 20
Contour Forests	2016	Shared Memory	Yes	6 - 16
Peak Pruning	2016	Shared Memory + GPU	No	3 - 47
Task-Parallel	2017	Shared Memory	Yes	0.1 - 20
This Thesis [WG21]	2021	Distributed	Yes	8 - 74
Policies				
Divide & Conquer	2003	Shared Memory	No	~ 0.9
Local-Global	2013	Distributed	No	33 - 57
Pruned Divide & Conquer	2014	Distributed	Yes	~ 33
Distributed Domain-Restr.	2015	Shared Memory	No	21 - 50

Tab. 4.1: Comparison of presented Merge Tree construction methods. Performance is in million vertices per second based on all available benchmarks. For the results of this thesis, two outliers (6 and 107 million) are not included in the span.

based on whether or not the augmentation is needed and a distributed memory cluster is available or shared memory can be assumed.

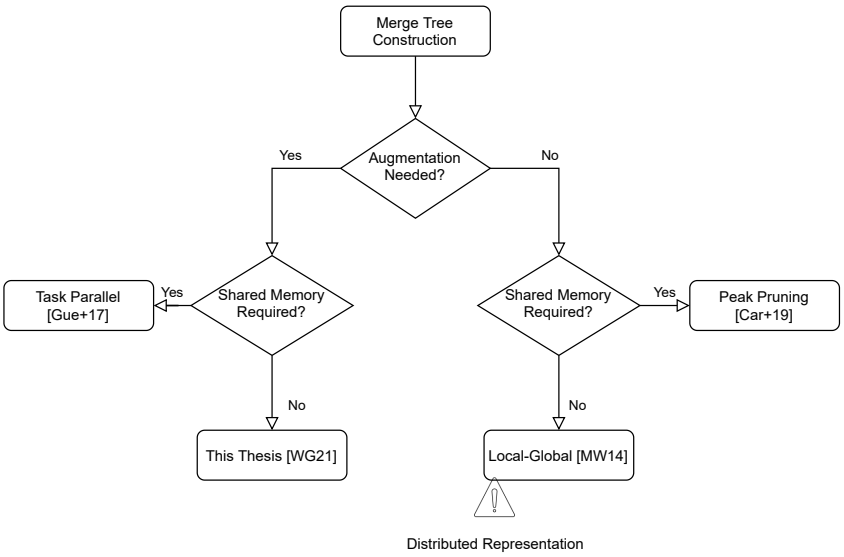


Fig. 4.2: Flow chart for deciding the Merge Tree construction type with the best expected performance.

5 Unordered Task-Parallel Distributed Augmented Merge Tree Construction

Within the surveyed related work, the Merge Tree construction method developed in this thesis can be characterized with a few keywords. The algorithm follows the *unordered* method similar to [RS14] and [Car+16b] but avoids following monotone paths through the entire domain. The resulting colorings do not form a complete partition but correspond precisely to $\mathrm{Ex}(m)$. The region boundary vertices are still subsets of $\mathrm{Bd}(m)$ and are still guaranteed to contain s_m as their smallest valued member. This avoids double work and shrinks the size of regions that have relevance for a minimum, which is beneficial in a distributed setting.

The algorithm is also *task-parallel*. Regions are not created by data-parallel operations on all edges, but are grown around minima by tasks like in [Gue+17]. From this follows that no global data structures or barriers are needed. A minimum can identify its saddle and register its region there independently of all other minima. Once a saddle is identified by all its ancestors in the Join Tree, it can be (virtually) contracted by merging involved regions. This fits well with the goal of using the algorithm in a task-parallel pipeline.

This combination allows the algorithm to be *distributed*. In a distributed memory setting, regions can grow beyond data boundaries and on multiple localities simultaneously. With a (rather complex) polling strategy, the overall smallest valued saddle candidate is found and shared with all involved localities. While the tree is ultimately collected on a master locality, the ongoing representation built on each locality is very similar to the local-global Merge tree of [MW14]. The algorithm works the same locally as on multiple localities (a few poll steps would be skipped if only a single locality is involved) and thus exposes parallelism both within a data segment and across. It is therefore the first inherently hybrid kernel in contrast to divide & conquer distribution policies of shared memory kernels. This avoids the diminishing parallelism of fan-in stages and spreads communication costs across the entire computation.

The algorithm allows to construct the *Augmentation* as well. Like in [Car+16b] we utilize the observation, that the Augmentation for an edge starting at m is exactly the subset of $\mathrm{Up}(m)$ containing all vertices with smaller values than s_m. As this is also a subset of the region colored by m -namely $\mathrm{Ex}(m)$-, we can "cut" these regions at the value of the saddle. In contrast to [Car+16b] we do not perform domain restriction like in [Chi+05] and thus obtain the complete augmentation for the edge of m after this cut.

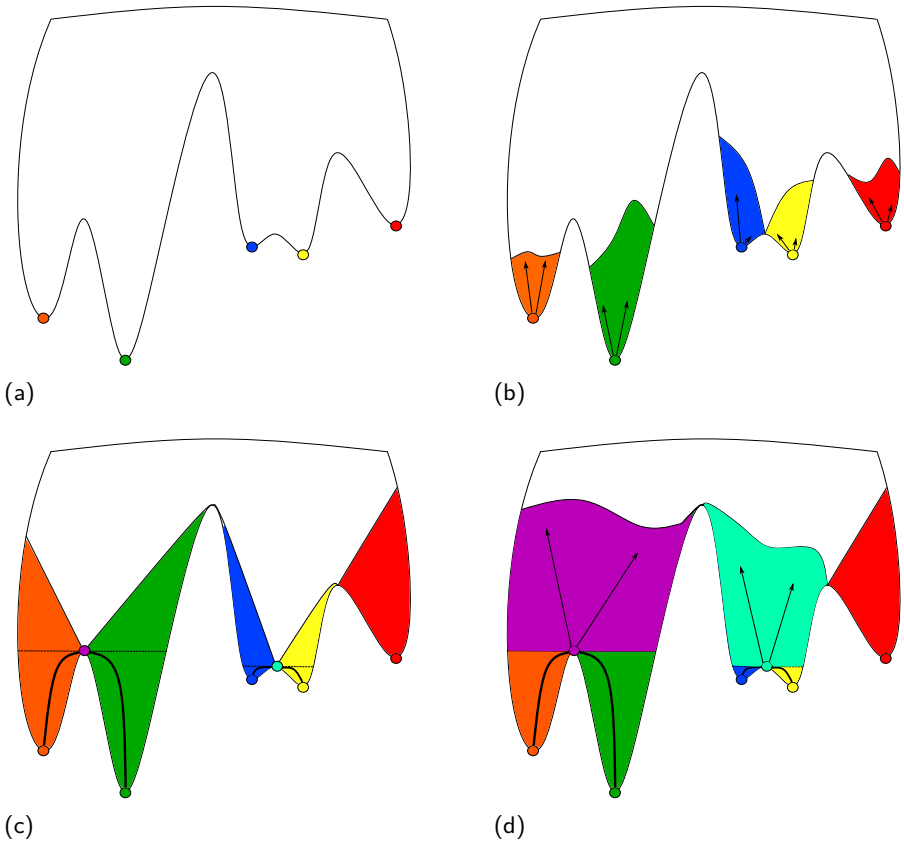


Fig. 5.1: Exemplary Join Tree computation on the height function of a manifold, deliberately made comparable to an example in [Gue+17]. In (a) local minima and thus Join Tree leaves are found according to 5.1.1. In (b) independent sweeps grow a region around each local minimum following arbitrary monotone paths according to 5.1.2. In (c) these growths terminated at non-exclusively monotone reachable vertices, namely boundary sets. The smallest valued boundary vertices are identified and prepared for their own sweep according to 5.1.3. In (d) prepared saddles continue their own sweeps in the same manner, constructing the entire Join Tree.

5.1 Algorithmic Structure

The input is treated as a 1-skeleton (skeleton graph), the dimension of the cells and embedding domain are therefore not relevant for the algorithm. Cell interiors of meshes that are not a triangulation could be treated by triangulating or employing an oracle but are neglected in our implementation. This enforces saddles to be at vertices and might introduce errors in the sub-resolution scale, see Figure 5.2. Since most data sets do not represent Morse functions natively, a simulation of simplicity is employed. In our implementation this is done by breaking up ties in function value by vertex index. This might introduce artificial persistence pairs with zero persistence (another important reason for simplification). Multi-saddles can be implicitly handled by the algorithm and will just create nodes with degree larger 3.

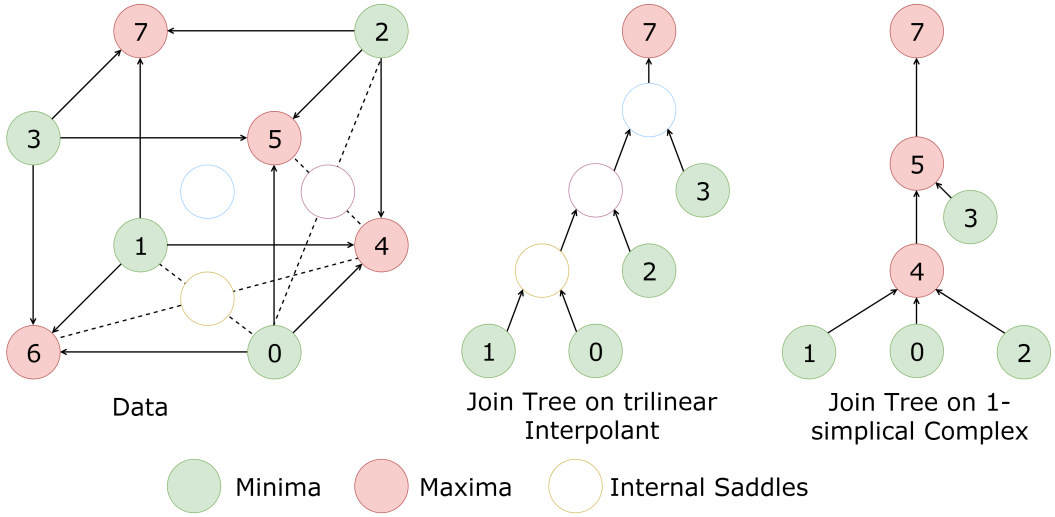


Fig. 5.2: Exemplary cuboid cell data with trilinear interpolated function. The actual Join Tree might deviate from the Join Tree of the 1-simplicial skeleton and might contain saddles that are not located at vertices.

The algorithmic structure follows the related work described above in the most general points, see Figure 5.1. Minima are identified by comparing their function values to all direct neighbors. Regions are grown around minima along monotone ascending paths to identify their saddle node adjacent in the canonical Join Tree. Augmentations can be obtained from these regions by cutting them at the saddle function value. The saddles are eventually virtually contracted and can be treated like the minima before. The Contour Tree could then be constructed from Join and Split Tree as usual. Since this thesis makes no contribution to this process it was not implemented for benchmarks. All runtimes (and those in the literature we directly compare to) were obtained for Join Tree construction only.

These individual steps will be described in all necessary detail below. We will first explain the algorithm for a shared memory system and detail the necessary communication to handle data boundaries after that. This is fitting for our algorithm, as the general process for shared and distributed memory is the same. The distributed algorithm is not an explicit combination of independent local processes but rather an adaption of region growth, saddle identification and saddle contraction to include more than one involved locality.

5.1.1 Minimum Search

Identifying minima is embarrassingly parallel and very simple. A vertex is a minimum if its function value is smaller than the function value of all its neighbors. It might be intuitive to expose as much of this concurrency as possible, creating one task for each vertex to be tested. However, these tasks would have runtimes below a few milliseconds. This may be suitable for SMP or GPU hardware, but even with the low cost context switches of HPX, the task management overhead would exceed the actual compute time. Depending on the data size and hardware availability it could be beneficial to perform the scan for minima on a GPU and pass back a list of minima to CPU.

In our implementation, the search for minima is a sequential iteration over all vertices within a single task. This allows other parallel workers to immediately start working on the region growth of discovered minima and keeps the number of pending tasks low. This also evenly distributes the trivial, low latency work on small leaf regions across the computation time, allowing it to be a reliable backup to fill the growing latencies of inner nodes with their larger regions that span more and more localities. This is a great example for the sometimes unintuitive design principles of task-parallel latency hiding.

Algorithm 1 SCAN_MINIMA()

```
for all v : DOMAIN_VERTICES do
  if smallerValuedNeighbors(v).empty() then
    HPX.async<GROW_REGION>(v);
  end if
end for
```

5.1.2 Region Growth

The heart of the algorithm is the identification of saddles for minima. This is achieved by growing a region around a minimum m that traces the set $\mathrm{Ex}(m)$. During this growth the sweep front is explicitly maintained, so that it corresponds exactly to $\mathrm{Bd}(m)$ once the growth terminates. It is then easy to extract the smallest valued vertex as the saddle.

The growth of the region is tracked by a union find data structure, more precisely a disjoint set forest. Like in other related work each minimum becomes the representative root of its connected component. Trivial one-element unions are performed for each vertex that is in $\mathrm{Ex}(m)$ to track their membership in the region around m. This allows for fast tracking (union) and lookup (find) of region memberships.

A vertex v can be added to the region of m if it is guaranteed to be in $\mathrm{Ex}(m)$. This is the case if all smaller valued direct neighbors of v are guaranteed to be in $\mathrm{Ex}(m)$. As m is in $\mathrm{Ex}(m)$ it can be added right away, initializing its region and becoming the root of its disjoint set tree. After that, all neighbors of m that have no smaller valued neighbors other than m can be added to the region and united in the disjoint set forest. This way the region grows to all vertices that only have smaller valued neighbors that are already in the region.

In theory, all those vertices could be added in parallel and we will use this property for distributed data settings. Within a locality however, only a single task is assigned to a region growth. This avoids any need to synchronize access to the involved data structures. For noisy data sets, regions will be small and additional parallelism will typically not be worth its overhead. For example, the average size of the regions $\mathrm{Ex}(m)$ of leafs was around 7 for the data sets utilized in the benchmarks in this thesis. To achieve production quality performance the region growth could autonomously create helper tasks to further parallelize the region growth, once a certain region size is reached.

In our implementation a single queue therefore manages the region growth. Beginning with m, every vertex that is added to the region (united in the union find structure) adds all its larger valued neighbors to the queue. Then, the next queue element v is visited. All its smaller valued neighbors are tested for region membership (find in the union find structure) and if they are all in the region, v is added to the region (and adds its larger valued neighbors to the queue). Once the queue is empty, all and exactly those vertices in $\mathrm{Ex}(m)$ have been added to the region.

This modified breadth first search can result in duplicate entries in the queue and in fact, vertices will be visited once for each smaller valued neighbor. To alleviate the performance impact of this, the queue could be replaced by a priority queue (based on a min heap) to visit smaller valued elements first. Similar to the locally ordered progression of [Gue+17], this would allow to actually add a vertex as soon as possible and quickly discard it every time it is visited after that, see Figure 5.3. Our experiments showed however, that the additional insertion cost of min heaps actually exceeded the benefits of that and our implementation features a regular queue.

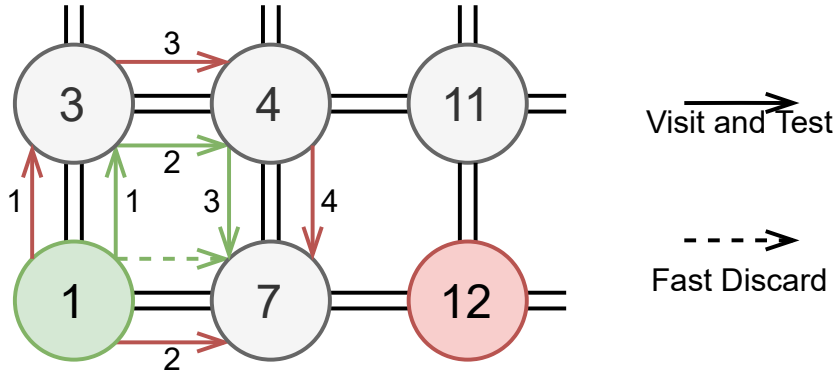


Fig. 5.3: Visiting pattern for a queue (outside red arrows) and a priority queue (inside green arrows). If the vertex 7 is visited before 4 it is fully tested but not added, even though that will change on the second visit. If 7 is visited after 4 it can be added immediately and a second visit can be skipped without full testing.

In addition to the union find structure (that is shared with all other growth tasks) and the growth queue, a third data structure allows us to track the sweep front and to capture the set $\mathrm{Bd}(m)$. Whenever a vertex is visited but not added to the region it is instead added to this boundary data structure. If the same vertex is later added to the region, it is removed from the boundary. In our implementation this data structure is a simple set (based on a red black tree), as it allows for fast insertion, removal and constant time minimum search.

In other words, a breadth first search adds all vertices that belong to $\mathrm{Ex}(m)$ to the union find component of m. It also tracks in a boundary set all vertices that are adjacent to this region but have smaller valued neighbors outside of the region. Once the growth terminates the boundary set corresponds to $\mathrm{Bd}(m)$ and a minimum search identifies the saddle s_m.

All vertices that have been added to the region and have a smaller function value than s_m belong to the augmentation of the edge $(\hat{m}, \hat{s_m})$. All of these vertices currently point to m in the union find data structure. A modified path compression

Algorithm 2 GROW_REGION(m)

```
edge = JoinTree.edgeFrom(m);
(queue, boundary, augmentation) = edge.getStructures();
queue.push(m);

while !queue.empty() do
  v = queue.pop()
  if unionFind.find(v) == m then
    continue;
  end if

  bool canAdd = true;
  for all n : smallerValuedNeighbors(v) do
    if unionFind.find(n) != m then
      canAdd = false;
    end if
  end for

  if !canAdd then
    boundary.add(v);
    continue;
  end if

  unionFind.union(m, v);
  augmentation.add(v);
  boundary.remove(v);
  for all n : largerValuedNeighbors(v) do
    queue.push(n);
  end for
end while

s_m = boundary.min();
edge.saddle = s_m;
edge.augmentation = augmentation.lowerValuedThan(s_m);
ASSIGN_SADDLE(m, s_m, boundary, augmentation.higherValuedThan(s_m);
```

may allow to implicitly represent the augmentation within the union find data structure. However in our implementation, we explicitly tracked augmentations with a forth data structure. Every time a vertex is added to the region of m it is also added to the augmentation of its edge. Edges are therefore represented by their lower end vertex, which is unique. Once s_m is found, the augmentation is split at its function value, meaning all vertices with a function value smaller than s_m remain in the structure and form the augmentation of the edge. The rest forms

an *inherited augmentation* that will be assigned to descendants in the tree during saddle contraction (see below).

To allow for a constant time cut of the augmentation we realized the structure by means of a skip list set. However, our experiments showed, that the additional overhead of its maintenance is larger than its benefits and a simple flat set is used in our implementation instead.

The same union find data structure is shared between all growth tasks. The use of this data structure is thread safe without any use of synchronization. No regions $\mathrm{Ex}(m)$ intersect and boundary vertices are not used in union operations so no two tasks will write to the same positions. Find operations only read and are thus safe to use. The growth queue, augmentation set and boundary set are only used by one task, thus no synchronization is necessary for the region growth and saddle identification.

5.1.3 Saddle Contraction

The region growth above allows a single task to find a single leaf edge and its augmentation without any synchronization or interaction with other tasks. To be able to virtually contract saddles and treat them like minima in another region growth, the tasks now have to coordinate and share information about saddles.

For this, we maintain a map of pointers to edges with the lower valued incident vertex as key. Edges store both incident vertices and also aggregate the three data structures used for their computation: growth queue, boundary set and augmentation set. This allows to modify these structures for a saddle before that saddle starts its own region growth to perform virtual saddle contraction.

Once a growth tasks terminated and identified its saddle it will call a method (ASSIGN_SADDLE) for the saddle. This method has two purposes. First, it prepares the saddle for contraction based on the information of the finished edge. Second, it tests if this is the last incoming edge to perform this preparation. If so, the virtual contraction is complete and a growth task can be scheduled for the saddle.

Let us revisit what virtual saddle contraction is. The sub-tree of a Join Tree that includes all ancestors of a given saddle s augments only vertices with lower function value than s. We will call it the sub-tree *below* s. These contained vertices are connected and could be contracted onto the saddle. For that, all contained vertices could be deleted. All edges between two deleted vertices could also be deleted. All

edges between one deleted and one remaining vertex could replace the deleted vertex with s. This would make the saddle a local minimum and the Join Tree of the resulting domain would be exactly the Join Tree minus the sub-tree below s.

This allows us to treat a saddle s like a minimum for our computation, if the sub-tree below s is already computed. Luckily, we do not need to perform the actual saddle contraction like above. It suffices to update the growth queue, boundary and augmentation for s before its growth starts as if s itself was a minimum and had already performed the region growth that was in reality done by its ancestors.

The region $\mathrm{Ex}(s)$ after contraction contains the regions $\mathrm{Ex}(m)$ for all minima m that are ancestors of s. The union find structure can simply unite these sets by performing a union operation of m and s, with s as the representative. If this is done for all minima assigned to s, the union find data structure represents a virtual contraction of s.

The initial boundary of s is also simple to compute. It is the current state of the sweep front and thus simply a union of all boundaries of minima reaching the saddle. The initial augmentation of s is again just a union of all inherited augmentations of minima reaching s. Performing the unions on the set data structures has linear runtime in their size and becomes increasingly expensive for larger and larger regions. This is one of the reasons, why the assign saddle procedure is the largest overhead introduced by our method over related work. However, this mainly becomes a problem if few large regions contain large portions of the data. For noisy and complex data sets individual regions are small and the trunk contains the majority of the domain. If data sets are known a priori to contain large, smooth features it could be advisable to replace the sets that are fast for insertion, removal, min search and split with data structures that perform faster unions (e.g. heaps or disjoint set forests).

The initial state of the growth queue is a little more complicated. The first intuition is to only push s to the queue. However, since actual saddle contraction would drastically expand the connectivity of s the direct neighborhood of s may not suffice. Naively, we need to add the entirety of the (united) sweep front boundary of s to the queue to test again whether region growth might now continue where ancestor regions could not. However, most of these tests would be redundant, as the region membership of smaller valued neighbors could only have changed because of the union operations between s and minima reaching it. This limits the vertices that need to be tested again to the intersection of boundaries of ancestors, as only those vertices have smaller valued neighbors that are members in regions that were originally different and might have united now.

Algorithm 3 ASSIGN_SADDLE(m, s_m, m-boundary, m-inheritedAugmentation

```
edge = JoinTree.edgeFrom(s_m);
edge.lock();
(queue, boundary, augmentation) = edge.getStructures();

unionFind.union(s_m, m);
(intersection, complement) = m-boundary.dissectBy(boundary);
boundary.add(complement);
queue.push(intersection);
augmentation.add(m-inheritedAugmentation);

for all n : smallerValuedNeighbors(s_m) do
  if unionFind.find(n) != s_m then
    edge.unlock();
    return;
  end if
end for
HPX.async<GROW_REGION>(s_m);
edge.unlock();
```

The assign saddle preparation for a terminated region growth of a minimum m for a saddle s thus consists of the following: Perform a union operation for m and s in the union find data structure. Intersect the boundary of m with the preparation boundary of s and add these vertices to the preparation growth queue of s. Unite the remaining boundary of m with the preparation boundary of s. Unite the inherited augmentation from m with the preparation augmentation of s.

After that the test if m is the last minimum to call assign saddle for s is easy. If find operations for all smaller valued neighbors of s return s, then all assign saddle union operations have been performed. A new region growth task for s can now be scheduled. Whether or not this was the case, the region growth task ends after performing the assign saddle operation.

Access to the edge map is guarded by a lock, allowing only one task to receive or emplace pointers at a time. A lockfree, threadsafe map implementation may have a noticeable performance benefit, as there are many accesses to this map. If enough memory is available, the map could be realized as a fixed size array of pointers for each vertex id. However, each access simply stores or retrieves a pointer so even with high congestion latencies on map access should be low. Additionally, each edge contains a lock, so that the assign saddle procedure can only be performed by one finished growth task per saddle at a time.

5.1.4 Trunk Skipping

In conclusion, the minimum search task schedules region growth tasks. Let us call them tasks of the first generation. These identify for their minimum a corresponding saddle and prepare it for its own region growth. Each saddle that has only incoming edges from leafs in the tree will become contractible after the termination of the corresponding first generation growths. It will be virtually contracted and start its own region growth task, now of the second generation. The growth for any inner tree node $\hat{v}$ will eventually be started in the (n+1)-th generation, with n being the height of the subtree below $\hat{v}$.

Like in [Gue+17] the task-parallel paradigm includes no global barriers after each generation. Tasks of all generations can run in parallel, with the only sequential dependency being between ancestors and descendants in the Tree. The critical path (as in the longest chain of sequentially dependent tasks) therefore corresponds to the branch with the greatest height in the tree. For Merge Trees, this corresponds to the branch between global minimum and global maximum and is called the trunk. Noisy data sets, especially measurements of objects of interest that are contained in a larger open volume (like air or tissue) tend to augment the majority of vertices to edges along the trunk and have a rather large jump in height from the second highest branch to the trunk. For such data sets the procedure of trunk skipping described in [Gue+17] can increase performance by an order of magnitude.

Once only one task is still running, it is clear that the corresponding region will "collect" all saddles that are not yet contractible. This also has to happen in order of their function value, since only ascending paths are followed to discover saddles. All vertices that do not yet belong to an augmentation will be reached during this process and will be assigned to edges according to their function value. With these observations the actual spatial and neighborhood relations become irrelevant and the region growth procedures can be replaced by a fast parallel scan and sort.

First, stop the last region growth task and sort all *dangling* saddles, as in non-contractible saddles that are missing one incoming edge. In our implementation, dangling saddles are identified by iterating over the edge map and adding all edges that have been created but not assigned an endpoint to an ordered collection (e.g. set). If memory is available, the list of dangling saddles could be maintained explicitly throughout computation.

The last region growth task can assign the first saddle in the sorted list of dangling saddles. Iterating over this list, each endpoint can be assigned to the next dangling saddle, with the highest valued dangling saddles endpoint being the global maximum

of the function. These saddle assignments can skip union find, queue and boundary operations completely.

Second, for each vertex v that still points to itself in the disjoint set forest, find the edge $(s1, s2)$ with $f(s1) < f(v) < f(s2)$ via binary search in the list of dangling saddles. Add v to the augmentation of this edge. Again, it could be beneficial to move this computation to a GPU depending on the data.

With this, the augmented canonical Join Tree is constructed in the form of a set of edges, containing their augmentation as meta information.

5.2 Hybrid Distribution

The above description of the algorithmic structure assumed shared memory for simplicity. The described algorithm cannot have better performance than the one in [Gue+17], as we are performing precisely the same, but also additional computations. Especially the explicit management of sweep fronts (boundaries) introduces additional computational effort.

The only benefit of our approach is that a single region growth could be parallelized, like in [RS14] or [Car+16b]. However, as a majority of regions is very small in complex, real world data sets and task granularity has a lower limit, we did not utilize this possibility on shared memory. The real strength and scope of our algorithm is scalability on distributed hardware systems. Growing regions on all involved localities in parallel, hiding communication latencies by task-parallel context switches and avoiding fan-in stages of strict divide & conquer policies creates a unique set of synergies with these systems.

For the hybrid distributed setting, we assume a number of localities. Each locality can host multiple OS-threads, but all parallel workers on the locality have a shared memory. Different localities however do not share memory and can communicate only by scheduling tasks on other localities (and by their method parameters and return values). We do not utilize HPX capabilities to explicitly migrate objects between localities.

A static segmentation of the complete data set onto the localities is assumed given in advance. This may well represent the output of an earlier pipeline stage. Each locality can begin work once its data arrives, independently of other localities data becoming available, so no global barrier is needed between the preceding pipeline stage and the Merge Tree construction. No assumptions about data distributions

or data segment topology is made, but we do require one ghost layer (the function values and indices of vertices adjacent to those in the localities own segment need to be available). The benchmarks presented in this thesis were acquired by segmenting data into axis aligned cuboids of approximately similar size.

To scale our algorithm to distributed settings, the underlying algorithmic structure is not changed. No barriers, global data structures or fan-in stages are necessary. Instead, the four main aspects of the algorithm have to be adjusted, so that multiple involved localities can cooperate. This allows to focus communication on individual edge construction.

5.2.1 Minimum Search

Localities are responsible for the vertices within their own data segment. This excludes ghost layer vertices, that are known on a locality but have a different locality responsible for them. This way, each vertex has exactly one locality responsible for it. The minimum search procedure does not change in a distributed setting. Each locality starts one task that iterates over the vertices it is responsible for and starts a region growth for each minimum found. Using the ghost cells actual minima can be distinguished from vertices that have no smaller valued neighbors within the data segment.

5.2.2 Region Growth

Each locality schedules a region growth task for each local minimum it is responsible for. These tasks fundamentally perform the same growth as described for shared memory. The main difference, is that the region might grow beyond the data segment of the locality. In this case, each involved locality maintains its own version of the edge data. Growth queue, boundary and augmentation exist once per locality and edge and will only ever contain vertices this locality is responsible for. The union find and edge map data structures were global in the shared memory setting. Now, each locality has its own version of these two structures (still shared between all local tasks).

The union find data structure will at least contain all vertices the locality is responsible for. Additionally, it will have to track all remote vertices that represent connected components that any local vertex belongs to. These remote vertices will be saddles of edges the local vertices are augmented to and their descendants in the Join Tree.

For this purpose the data structure is represented by a tuple of a fixed size array over all local vertices and a map with remote vertex IDs as key. The "pointers" stored in the array and map are in fact vertex IDs that are either arithmetically assigned to array positions or looked up in the map. This way, local and remote vertices can perform union operations with each other (possibly introducing remote vertices to the map) and find operations always return correct IDs. Even though some remote saddles are represented in the structure, the majority of remote vertices is not. This is very similar to the local-global representation of Merge Trees in [MW14].

Similarly, the edge map will at least contain all edges with start- or endpoints the locality is responsible for. Additionally, it will contain a local version for edges between descendants of these points. Since the edge map is a map with vertex IDs as key, the implementation does not need to change.

The cooperative region growth allows to continue region growth on remote localities. These localities keep track of their own local boundary and can themselves again continue the region growth on remote localities (even coming back to already involved localities). Once all growth queues on all localities are empty the set Bd is available, but is partitioned onto all involved localities. Each involved locality reports its local boundary minimum to the locality that originally started the region growth, which in turn finds the saddle and issues the saddle assignment on all involved localities. See Figure 5.4.

This simple idea is not trivial to implement. The original region growth will at first proceed as in shared memory. The reaching of a data boundary can be identified by trying to push a ghost vertex to the queue. The growth can of course not operate on the ghost vertex and instead schedules a subordinate region growth task on the locality responsible for the vertex. We will call such recursive, remote region growth tasks a *finger* of the calling task.

If a vertex is added to the region and has larger valued neighbors that are ghost vertices it is called a *finger vertex*. For each finger vertex, the region growth "spills" to the neighboring data segments and "stretches a finger" to another locality to continue the region growth. Within the fingers, the minimum m is a remote vertex, but this can be handled by the union find data structures map portion. Vertices can still perform union operations and the local portion of $\mathrm{Ex}(m)$ is traced on each locality individually. The local portion of the sweep front can be maintained per locality and once the finger terminates, the minimal valued vertex on that sweep front (the fingers saddle nominee) can be sent back to the caller as the return value.

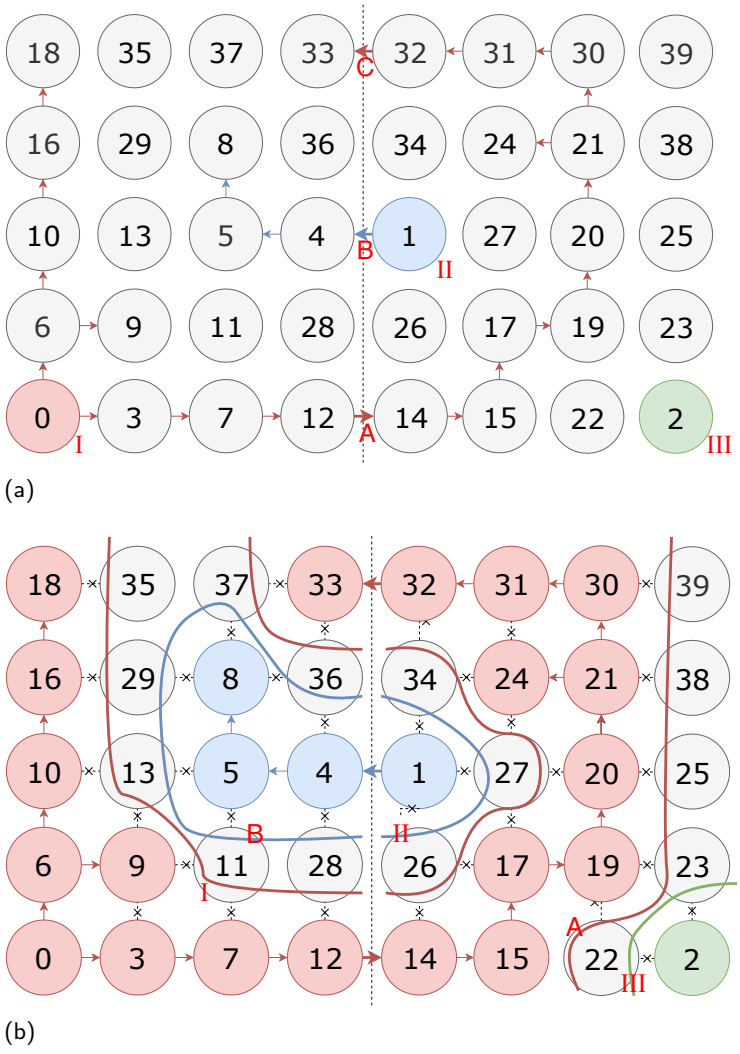


Fig. 5.4: (a) Region growth tasks (I-III) start at each minimum (colored vertices). Regions are grown like indicated by the arrows, which creates finger tasks (A,B) which again can create finger tasks (C). (b) Ex regions (colored vertices) and boundary sets (colored lines) are identified locally for each locality. The illustration shows task labels at the saddle nominees returned by them. C started while I was still running and thus just injected 32 to the queue and returned an empty resultMap.

There is one problem however. A locality might receive multiple fingers for the same minimum region growth, since one finger has to be scheduled for each ghost vertex that is put in the queue. The timings and interaction between these fingers create multiple edge cases, highlighting the complexity of task-parallel responsibility management. These additional fingers might be scheduled while another finger (or the original growth task) on that locality for the same minimum is already running. Since queue, boundary and augmentation are not thread safe, this will not work

Algorithm 4 GROW_REGION(m)

```
edge = JoinTree.edgeFrom(m);
(queue, boundary, augmentation, flag, counter) = edge.getStructures();
queue.push(m);
resultMap = new ResultMap();
GROW_LOOP(m, resultMap);
resultMap[locality(m)] = boundary.min();
s_m = resultMap.overallMin();
for all loc : resultMap.keys() do
  HPX.async<ASSIGN_SADDLE>(loc, m, s_m);
end for
```

naively. On the other hand, additional fingers might be scheduled after earlier fingers already terminated and reported back a saddle nominee.

To clear up this confusion, additional meta-data is embedded in the edge data structures. First, a flag is added, that keeps track of whether or not a finger is currently performing work on the edge region. Second, a counter is added, that counts how many saddle nominees have already been returned.

Once a finger task starts, it acquires the lock of the edge and tests whether the ongoing work flag is set. If not, then the task sets the flag, releases the lock and starts a region growth loop like in the shared memory case. It adds the starting finger vertex to the region (similar to the minimum being added in the original growth task) and grows the region until the growth queue is empty again. At this point it acquires the lock of the edge again, increases the nominee counter, un-sets the ongoing work flag and returns a tuple of the smallest valued vertex on the boundary as saddle nominee and the current nominee count.

If a starting finger task finds the working flag set, it simply adds its starting finger vertex to the queue, which is replaced by a double ended queue for this purpose, to allow for thread safe handling of front and back ends. Since ghost vertices are never pushed to the queue by the actual region growth loop, any ghost cells that are pulled from the queue are guaranteed to be finger vertices and can be added to the region without testing their smaller neighbors (which would not be possible, since not all neighbors for ghost vertices are known on a locality).

This way, an arbitrary number of fingers can be scheduled on a locality. If the target locality is already performing a growth, the new information about data segment border vertices is simply injected into the running process queue. If the target locality has already reported saddle nominees for earlier fingers, the counter allows the calling task to overwrite older results with the most current finger.

Algorithm 5 GROW_LOOP(m, resultMap)

```
edge = JoinTree.edgeFrom(m);
(queue, boundary, augmentation, workflag, counter) = edge.getStructures();
while !queue.empty() do
  v = queue.pop()
  if isGhost(v) then
    for all n : smallerValuedNeighbors(v) do
      if unionFind.find(n) != m then
        canAdd = false;
      end if
    end for
  end if
  if unionFind.find(v) == m then
    continue;
  end if

  bool canAdd = true;

  if !canAdd then
    boundary.add(v);
    continue;
  end if

  unionFind.union(m, v);
  augmentation.add(v);
  boundary.remove(v);
  for all n : largerValuedNeighbors(v) do
    if isGhost(n) then
      resultMap.UpdateWith(HPX.async<GROW_FINGER>(m, v));
    else
      queue.push(n);
    end if
  end for
end while
```

However, there is another problem. Fingers might create fingers themselves and have to account for the resulting remote saddle nominees as well as their own. Naively, a finger could simply return the best (lowest valued) saddle nominee among all its recursive sub-fingers and its own. However, consider the following scenario: Locality 0 starts a region growth and creates a finger on locality 1. The finger on locality 1 creates another finger on locality 2, performs its region growth and finds a saddle nominee of value 30. The finger on locality 2 performs its region growth and finds a saddle nominee of value 20. Naively, 20 is reported back to locality 0 as a saddle nominee. Locality 0 however might also start a finger on locality 2

Algorithm 6 GROW_FINGER(m, v)

```
edge = JoinTree.edgeFrom(m);
edge.lock();
(queue, boundary, augmentation, flag, counter) = edge.getStructures();
resultMap = new ResultMap();
if workflag then
  queue.push_back(v);
  edge.unlock();
  return resultMap;
end if

workflag = true;
edge.unlock();
unionFind.union(m, v);
for all n : largerValuedNeighbors(v) do
  if !isGhost(n) then
    queue.push(n);
  end if
end for

GROW_LOOP(m, resultMap);

edge.lock();
resultMap[locality(v)] = (boundary.min(), ++counter)
flag = false;
edge.unlock();
return resultMap;
```

afterwards. This finger might continue the region growth, adding the vertex with value 20 to the region and removing it from the boundary. If both locality 0 and locality 2 now find saddle nominees with values larger 30, the actual saddle 30 is lost.

Therefore, instead of returning only its own saddle nominee and counter, each finger returns a map with involved localities as key and saddle nominee/counter tuples as value. This map keeps track of the most recent nominee of each locality as observed by the finger task. Everytime a finger returns, the task that called it compares this returned map to its own and replaces the entry for each locality with the nominee that has the most recent counter. Once the original region growth has an empty queue AND all called fingers returned, the final map contains the saddle as the smallest valued, current saddle nominee.

The original region growth task therefore is the root of a call tree of fingers. Each finger either performs a recursive region growth or just injects the new information

into a running region growth. Every time a growth queue runs empty, the smallest valued boundary vertex is reported as a saddle nominee. All recursive calls that return, are tracked in a result map. Once all recursive calls returned, this map represents all nominees found by descendants in the call tree.

5.2.3 Saddle Contraction

With this branching, distributed region growth, the original GROW_REGION task for m collects a map of all localities the region spans and the minimal elements of their respective local boundary. The smallest valued of these final nominees is the actual saddle s_m.

Considering the objectives of the assign saddle task in shared memory, there is little we need to change for the distributed setting. The edge for s_m in the edge map needs to be created or retrieved if it already exists. A union operation with m and s_m is performed. The augmentation is split and the larger valued portion is inherited. The boundary of m is intersected with that of s_m to fill initial queue and boundary. The difference is simply, that all involved localities have to perform these operations on all their respective local portions of these data structures.

Edge maps and union find structures exist once per locality and are able to track entries for remote vertices. Edges that local vertices are involved in and all their descendants in the Join Tree are tracked on a locality. This coincides with the local-global tree representation [MW14].

Augmentations, boundaries and queues of different localities are disjoint, as all data structures only incorporate vertices that the locality is responsible for. The local augmentation can be cut and added to the edge. The inherited portion consists of vertices that will be augmented to edges that are between descendants of m in the Join Tree. All of those edges will be represented in the local edge map and will perform assign saddle operations on this locality. Each vertex will find the edge it is augmented to at some point, even if the locality is not responsible for both incident vertices. Similarly, the local boundaries can be intersected and added to local boundaries and queues, as no remote vertices can interfere here.

The only adaptions to the saddle contraction phase, is that the assign saddle task has to be called by the finished original region growth not only on its own locality, but on all localities that are responsible for any involved vertices. This is handily available as the set of keys in the final result map. Note, that finished fingers can

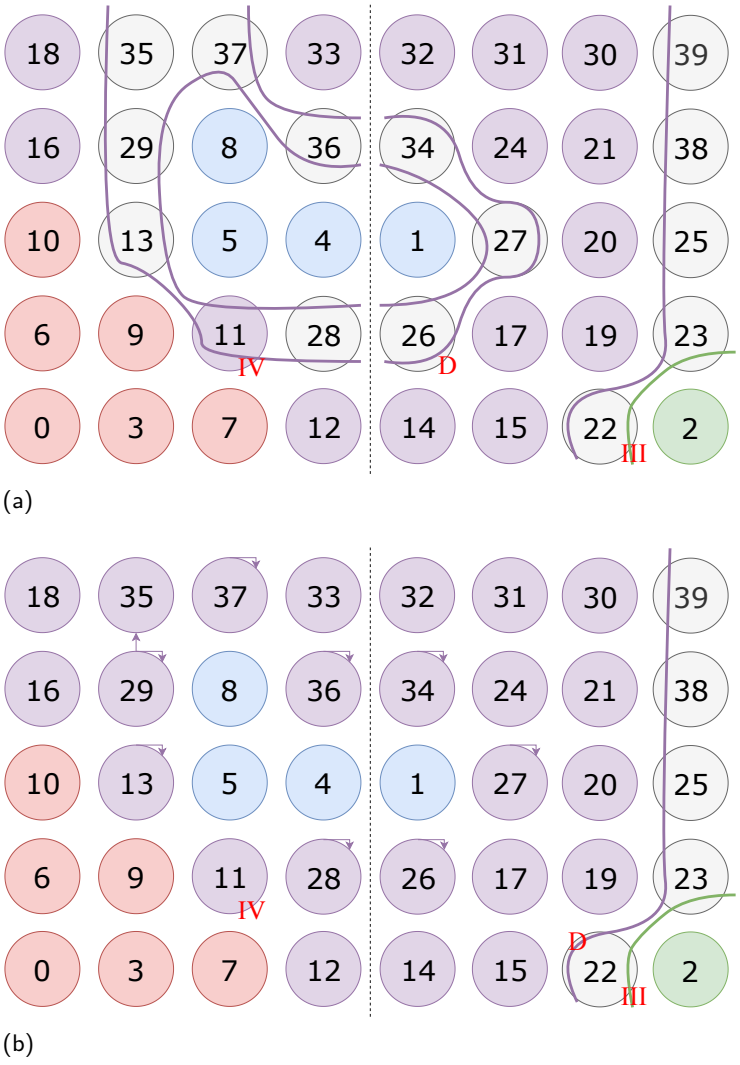


Fig. 5.5: (a) For the saddle with value 11 a new edge (purple color) is created on both localities. Augmentations have been cut and inherited (colored vertices). Boundary intersections push 13, 28, 29, 36 and 37 to the queue on the left locality and 26, 27, 34 to the queue on the right locality. Additionally, 35 is added to the initial boundary of 11 on the left locality and 22, 23, 25, 38 and 39 are added to the initial boundary on the right locality. A region growth task IV starts on the left locality, which immediately starts a peer finger D on the right locality. (b) IV grows, finds an empty boundary and has no saddle nominee. D grows and finds 22 as the smallest valued vertex on the remaining boundary. This is returned to IV, which in turn will assign the saddle 22 to 11 and issue the right locality to do the same.

not initiate assign saddle operations, as there is no way to determine locally if the overall region growth is fully terminated.

The assign saddle operation of shared memory contains a check for the saddle being contractible. In a distributed setting this is only possible and necessary on the locality that is actually responsible for s_m. If all smaller valued neighbors of s_m

return s_m from a find operation, then the saddle can start its own region growth. However, multiple localities have local representations for the edge of s_m. These representations include initial region growth queues of vertices where the growth needs to be continued.

In other words, the starting region growth on the locality responsible for s_m has multiple peers: localities that also have initial work to perform for the region growth of s_m. In order to correctly perform the region growth for s_m, all peers need to start a region growth on their own and report back to the original region growth exactly like fingers would, see Figure 5.5.

5.2.4 Tree Collection and Trunk Skipping

The procedures described above allow each locality to identify local minima and start region growths in parallel. The region growths progress independently and span across localities with recursive fingers. Saddles are identified, prepared and ultimately start their own region growths with mutual communications between involved localities only, creating local-global Join Tree representations.

In order to collect the entire Join Tree on a single locality, a master locality (e.g. with rank 0) is informed about every finished edge. Every time a saddle is checked for contractibility, the incoming edge that issued the assign saddle operation is registered with the master locality by scheduling a register task. This edge only contains both incident vertex indices and no data structures or meta information. When the Join Tree construction is done, the master locality therefore has a set of all Join Tree edges. The augmentations of the edges however are partitioned among the localities with all localities having knowledge about the augmentations of all local vertices. If memory is sufficient, these augmentations could be collected on a master locality too.

To manage trunk skipping in the distributed setting, the master locality also keeps track of the number of active region growths. For this, each locality reports the number of local minima its scan found back to the master locality, which adds them all up in a counter. This is the total number of starting points.

If a saddle is not yet contractible during the responsible assign saddle operation, then the number of active region growths effectively is reduced by one. The work issued from that starting point is completely done and the continuation is left to be discovered by work from other starting points. The register edge task therefore is passed a boolean flag, that is true only if the saddle was contractible and the

region growth can continue. If the flag is false, the master locality decreases its work counter by one.

While intuitively the counter should always be positive and only reach 0 once, the minimum scan tasks will most probably return only after the majority of region growths. The counter will therefore be negative for most of the computation time. Thus, trunk skipping can only be issued once all localities have reported their minimum counts AND the counter reaches 1. Only then is it guaranteed, that only one active region growth remains and trunk skipping can safely be performed.

The process of trunk skipping needs only little adaption to the distributed setting. The master locality has knowledge about all dangling saddles and can sort them like in a shared memory setting. This list is then broadcasted to all other localities (who are idling anyway) to allow them to sort their not yet augmented local vertices into these edges with binary sort like before.

6 On-The-Fly Simplification

We have described an effective Merge Tree construction module for use in a task-parallel, distributed pipeline. Especially in this setting, it might be interesting to represent the output of the pipeline stage not as the final tree, but as a stream of tree edges. Whenever a saddle is contractible, all information about the incoming tree edges are available. The localities responsible for the extrema have information about the saddle and all involved localities, which in turn hold complete information about the augmentations. At this point subsequent pipeline stages could start to perform work on these incoming edges.

The scientific work we want to support typically requires topological simplification of the data as an initial step in visualization pipelines. As discussed in Section 3.5, generalized simplification allows for topological simplification based on individual Merge Tree leaf edges and thus presents itself for a simplification module, that can perform online work on the tree edge output stream described above. Similar ideas of on-the-fly simplification emerged in related work [Pas+07].

This on-the-fly simplification profits from all benefits our task-parallel pipeline aims for. Idle times at late construction stages can be filled with simplification work, as a global barrier between pipeline stages is avoided. The increased occupancy allows for latency hiding and parallel scalability is improved. Additionally, since simplification can drastically reduce the size of the tree, the master locality collecting the tree is relieved of some memory consumption, as only persistent edges are reported to it.

This module is very simple in its function. For each leaf edge in the stream, test if the difference in function value between saddle and extremum is larger than a given ε. If not, let all involved localities set the function value of all augmented vertices to the value of the saddle (to obtain f'''). In this case, the edge is also removed from the stream (to obtain the simplified tree), see Figure 6.1. That way the simplification module acts as a kind of filter on the stream.

There are two formal problems to this. First, the resulting stream of edges now may represent a tree with obsolete degree 2 nodes. Discarding a leaf edge should discard the corresponding saddle as well. This can be achieved by modifying the construction

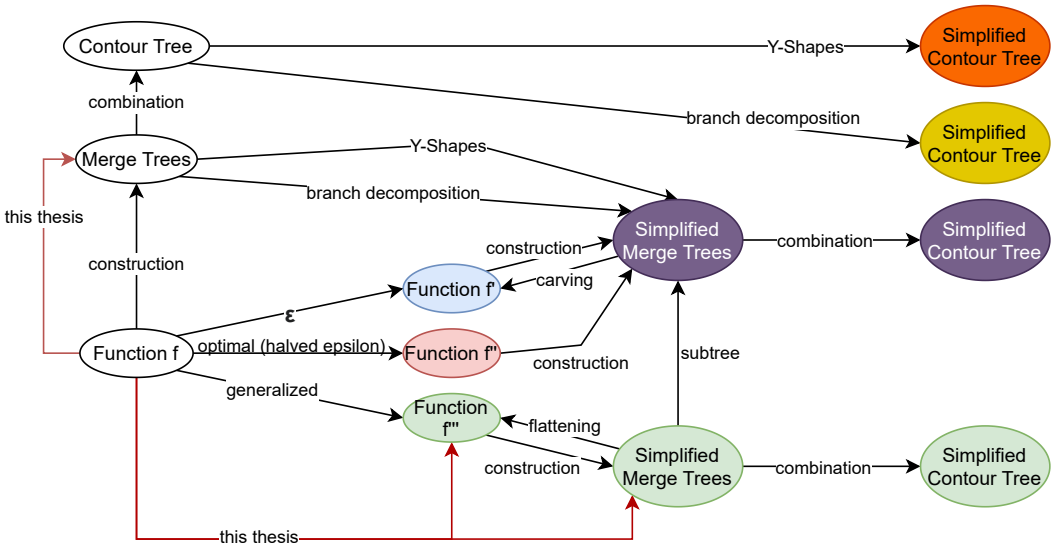


Fig. 6.1: The construction module presented in this thesis allows for unsimplified Merge Tree construction. Additionally, a connected pipeline stage can perform on-the-fly simplification of the constructed tree and apply the simplification back to the domain to obtain f'''.

module by extending the check saddle procedure. If at most one incoming edge for a saddle has a persistence larger ε, the saddle needs to be discarded and region growth is instead handled as if continuing the growth for the ancestor with the highest persistence. Instead, the degree 2 nodes could be collapsed in a post-processing step.

The second formal problem, is that simply setting the simplified regions function values to the function value of the saddle can introduce artificial persistence pairs, because of simulation of simplicity. This problem can be avoided by careful handling of function values like discussed in [Luk+21]. Since the introduced persistence pairs have a persistence of zero and the affected regions are guaranteed to have been processed by the construction module already, we chose to ignore these artifacts instead.

6.1 Alternative Parameters for Persistence Based Simplification

Apart from some formal caveats, the simplification module above is simple and effective. However it still depends on a user-supplied ε, which is an absolute value expressed in the function value scale. This value has to be chosen based on prior

domain knowledge. Since gaining domain knowledge is often the very goal of the visualization pipeline, this is a chicken-or-egg problem that is typically solved by multiple iterations of analysis and interaction.

For large and complex data this process can become lengthy and thus costly. Especially in the setting of high performance computing, compute time on the hardware systems typically has to be reserved and is too limited for interactive trial and error. This is especially true for in-situ scenarios. Here, a costly simulation producing the data is the first step in a visualization pipeline that is only run once. Often, the raw data is too large to store in consistent memory and is made only temporarily available during pipeline execution, so that only the resulting visualizations are stored for later human interaction.

Thus, choosing a fixed ε sometimes becomes a guessing game and rules of thumb based on a fixed percentage of the total function range are suggested in the literature. However, if the distinction between noise and features is not clear or does not fit such estimations, simplification effects can differ greatly from expected results.

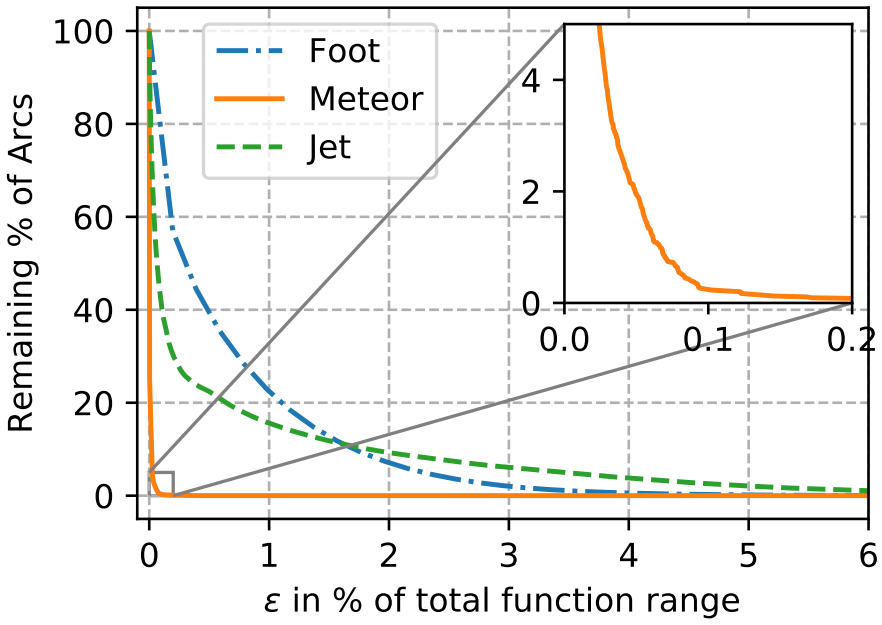


Fig. 6.2: The percentage of remaining arcs after simplification decreases with larger choices for the persistence threshold ε. The relation is highly non-linear and depends strongly on the specific data, making it difficult for the user to control simplification results by choosing ε. Other data sets lie between the shown graphs and are not shown to avoid visual cluttering.

From the perspective of the actual effect of the simplification -expressed in the percentage of pruned edges p- the fixed ε is a very unwieldy parameter. In the data sets used in this thesis, we observed a highly non-linear and greatly varying relation between ε and p, see Figure 6.2. We observed some data sets where an epsilon as small as 0.1% of the total function range would already result in the simplification of over 99% of all tree edges.

While alternatives to persistence based simplification have been proposed [CSP04] and general simplification can be applied to an arbitrary set of minima, such parameters typically suffer from the same problem of being based on prior domain knowledge and formulated in terms of absolute domain specific metrics.

We propose to perform persistence based simplification, but expose parameters to the user, that are based on the actual effect of the simplification. Setting either the percentage p or the absolute number N of tree edges that shall remain after simplification will allow users a more direct control over the simplification and increases flexibility in large-data applications.

The following presentation of online simplification based on p and N closely follows the published version of this work [WG20], for which the author is also the primary author.

6.2 Constrained Branch Count N

One setting of interest is to set ε such that a given number N of branches remain after simplification. This is for example necessary if work is performed in a memory-constrained environment.

For conventional post-processing, finding an ε so that N branches have a larger weight than ε is easily done by an inverse rank query. However for on-the-fly simplification the decision to prune or keep a branch must be done before weights for all branches are known.

To this end, we propose to use a lock-free priority queue [Pug89] Q. For each arc in the stream S, the arc is enqueued in Q, with its weight as priority. If more than N elements have been enqueued, Q is immediately dequeued from, resulting in the arc with $N+1$ largest weight so far. This arc cannot be among the N largest weighted arcs overall and thus has a weight smaller than the hypothetical ε we search. Thus it can be immediately pruned.

Substituting ε for N as a decision basis for topological simplification allows to maintain maximum detail in a memory constrained setting. Subsequent simplification may then be performed within main memory outside of the large-data application.

6.3 Percentile Size Reduction to p

Persistence is expressed in terms of the scalar function values and thus requires knowledge about the scale of that function to interpret and use. Reducing the output size by a given percentage however does not rely on that knowledge. Thus another interesting problem is to choose ε, such that a given percentage p of branches remain after simplification.

For on-the-fly simplification based on a given p the problem is the following: For each arc in the stream S, calculate the percentile rank of the arc and prune it if it is smaller than p. Of course precise ranks are only known a posteriori, thus an estimation based on the streamed arcs so far has to be made.

We next turn to the problem of estimating the percentile rank of an arc from all previously streamed arcs with minimal memory overhead.

6.3.1 Quantile Summary

To this end we propose the use of a biased quantile (bq-)summary [Cor+06]. When restricting the range of possible weights for the arcs, we can store those arcs as leafs in a binary tree over this range. The bq-summary instead stores a subset of nodes of this tree with associated counts, to approximate the distribution of stored leafs. By maintaining a set of invariants upon insertion, and running an amortized compression of the tree, sublinear memory consumption, insertion and estimation runtimes are achieved.

The data structure as proposed by the authors depends on a discretized range restriction of possible weights, containing U different weights. Insertion of a weight to the summary has an amortized cost of $\mathcal{O}(\log\log U)$. Rank estimation technically has the same cost, however as we will estimate the rank of every inserted weight (thus for every arc) we can slightly adapt the insertion method to yield the rank estimation as a byproduct. Memory consumption of the data structure is $\mathcal{O}(\frac{\log U}{\varepsilon}\log(\varepsilon N))$, with N the overall size of the stream and ε the maximal relative error of the estimation.

Since we do not want to rely on previous domain knowledge, we choose the range restriction to contain the whole range representable by floating point variables. Overall estimation accuracy achieved on real world data sets and runtime penalties paid for maintaining the data structure will be shown in Section 7.1.

6.3.2 Statistical Estimation

Online rank estimation inevitably suffers from irregular distribution of weights within the stream. With this, ranks of arcs within the history of the stream upon their arrival will deviate from the ranks of those arcs in the overall data. In other words, if a lot of short arcs are finalized first, the resulting summary data structure will rank short arcs too high.

To alleviate this problem one can try to introduce a measure of uncertainty into the summary, that represents size and variance of the observed part of the stream. If uncertainty is high, rank estimation can be adjusted to, for example, prune less arcs.

The most simple approach to statistical online rank estimation, is to assume arc weight distribution to be Gaussian. If arc weights are distributed according to a normal distribution, we can estimate this distribution by interpreting the previously observed stream as a sample. Small sample sizes will result in pessimistically estimated distributions, that will prune less arcs. Consider the following update mechanism for each arc a:

1. Filter the weight of a with Tukey's Fences [Tuk77] to reduce impact of outliers. Small outliers are pruned, large outliers are stored to the output.

2. If a is not an outlier, increase the sample size n by 1 and update overall empiric mean and empiric variance of the sample with a numerically stabilized Steiner Translation [CGL83].

3. From the sample, calculate a Students t and χ squared distribution with $n - 1$ degrees of freedom. For a given significance ε find the smallest explainable mean and variance.

4. These mean and variance correspond to the most pessimistic normal distribution that can explain the sample with significance ε. Evaluate the upper p percent quantile of this distribution and compare it to the value of a.

With this, after each arrival of an arc a, we calculate a confidence interval around the empiric mean, in which the true mean of the arc weights lies with ε percent certainty. To be most pessimistic and thus try to keep arcs instead of pruning them when in doubt, we choose the smallest mean in this range. Similarly we calculate an interval around the empiric variance, in which the true variance lies with ε percent certainty and choose the smallest variance. From this mean and variance we derive a pessimistic normal distribution. The upper p percent quantile of this distribution is a value, below which most probably at most $1 - p$ percent of the actual arc weights lie. Thus if the weight of a is below that value it is pruned.

7 Conclusion

7.1 Results

In this section, we present performance and scalability of the resulting implementation of the construction module.

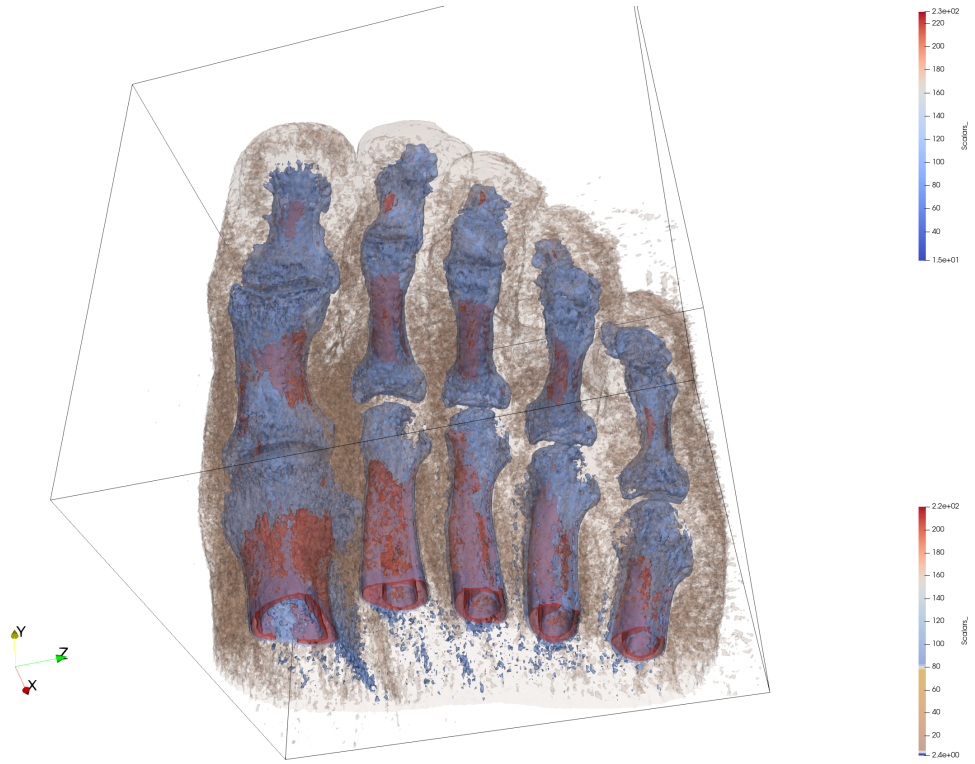


Fig. 7.1: Isosurface visualization of the Foot data set. CT scan of a human foot.

Benchmark Environment

All results emerged from experiments run on the AHRP High Performance Computer 'Elwetritsch' at TU Kaiserslautern. All involved processors were of type Intel XEON SP 6126 (19.25M Cache, 2.6 GHz, 12 CPU cores, 96GB RAM) with two processors per cluster node (locality). All times were measured for Join Tree construction, including

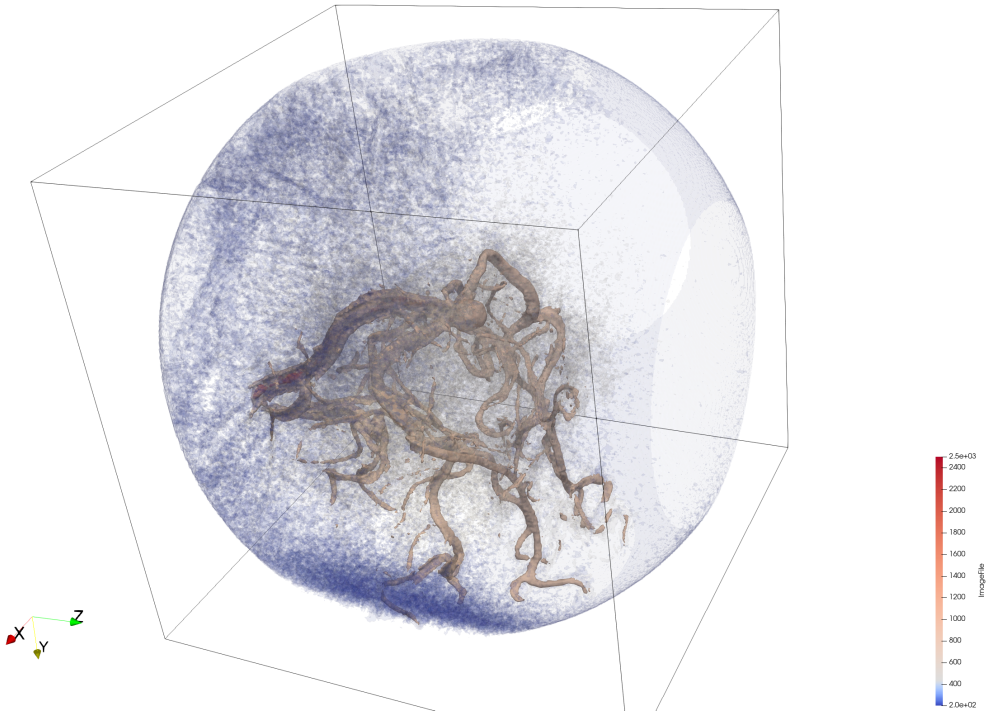


Fig. 7.2: Isosurface visualization of the Vertebra data set. Rotational angiography scan of a head with an aneurysm (contrasted vessels).

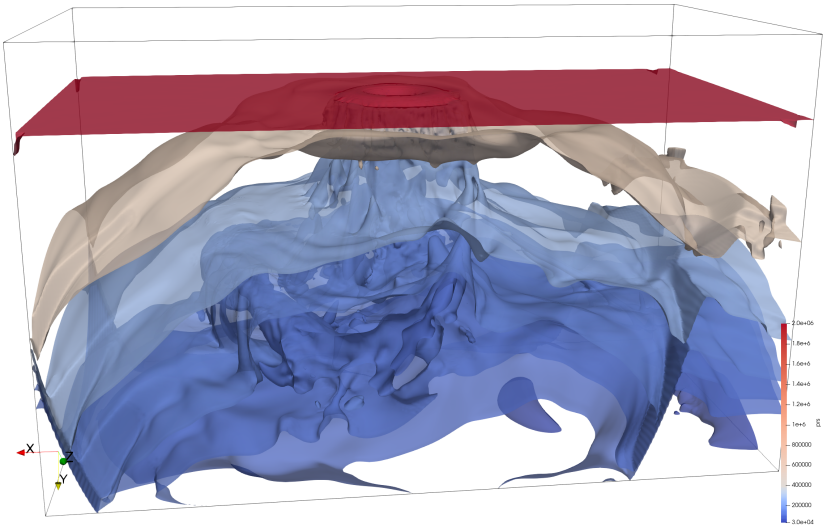


Fig. 7.3: Isosurface visualization of the Meteor data set. Simulation of a meteor impacting on deep ocean surface.

the augmentation and gathering of resulting arcs at the master node. No on-the-

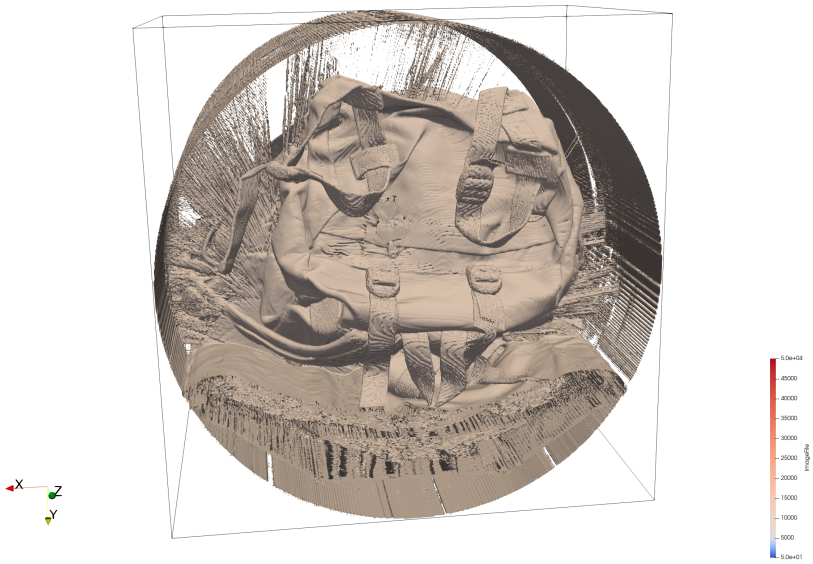


Fig. 7.4: Isosurface visualization of the Backpack data set. CT scan of a backpack.

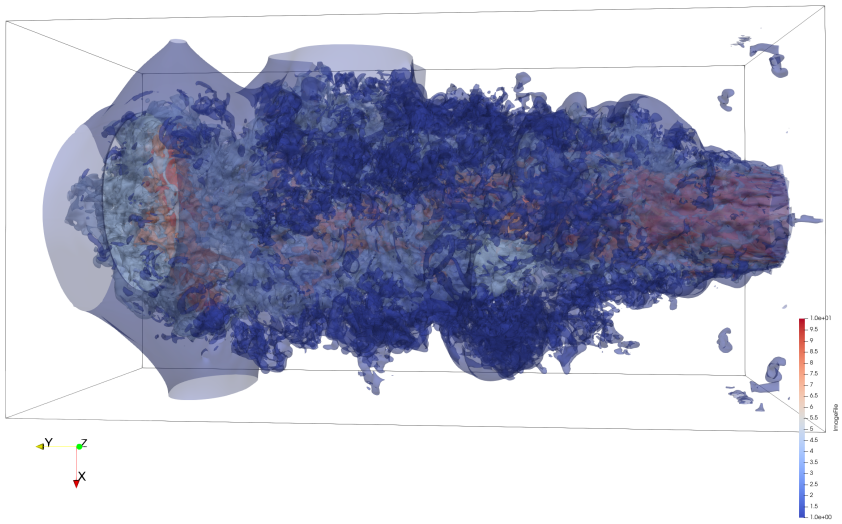


Fig. 7.5: Isosurface visualization of the Jet data set. Simulation of a water jet flow.

fly simplification was performed and the complete Join Trees were constructed. Runtime impacts and estimation accuracy of the simplification module are presented separately further below.

Our C++ and CUDA based implementations utilize the HPX framework on top of an OpenMPI parcelport and VTK for data input, using gcc version 9.1, nvcc version 9.2,

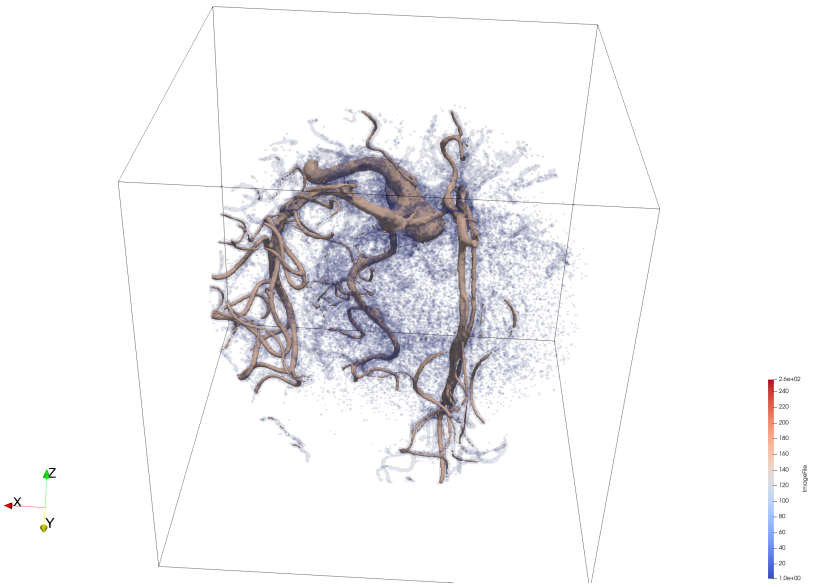


Fig. 7.6: Isosurface visualization of the Aneurism data set. Rotational C-arm x-ray scan of the arteries of the right half of a human head.

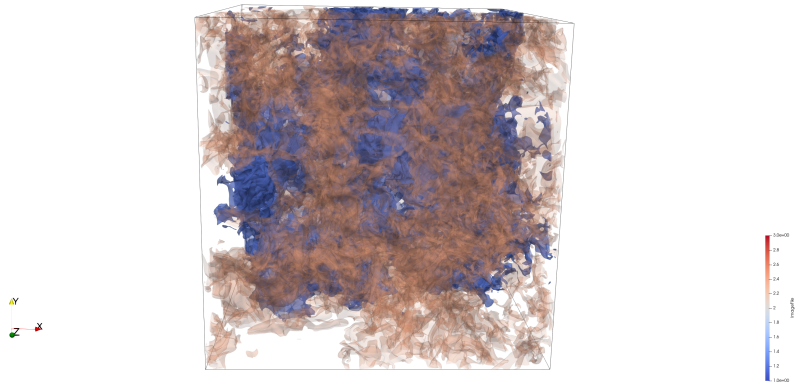


Fig. 7.7: Isosurface visualization of the Miranda data set. Density field in a simulation of the mixing transition in Rayleigh-Taylor instability.

hpx 1.3.0 and OpenMPI version 4.0. They are made publicly available via codeocean [Wer20].

Experiments were performed on openly accessible, well known data sets to allow for better comparability, see Table 7.1. Most data sets are from the Open SciVis Dataset page (https://klacansky.com/open-scivis-datasets/). Additionally, we use time step 15422 from simulation yA31 of the SciVis contest asteroid data set [PG17], the foot ct scan from the TTK example data (https://topology-toolkit.github.io/downloads.html) and a simulation of a jet fluid stream [Gar20]. These

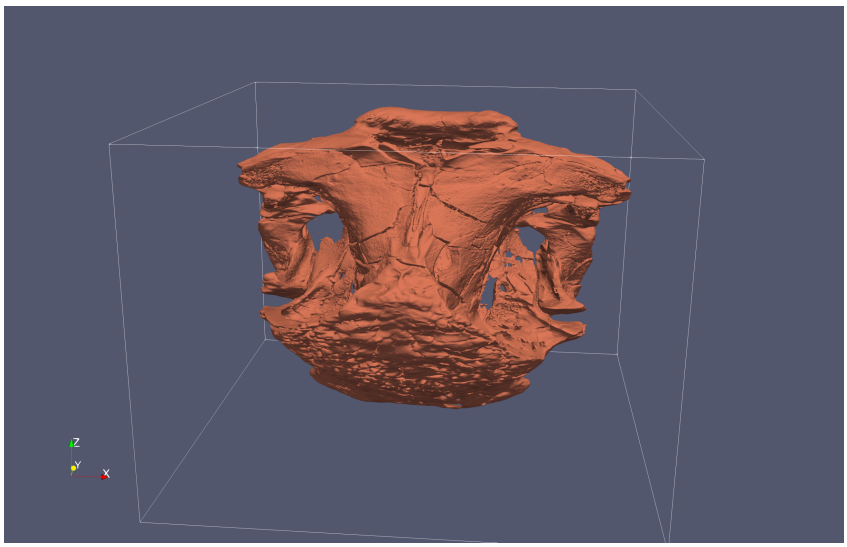


Fig. 7.8: Isosurface visualization of the Spathorhynchus data set. Density field of a scan of a Spathorhynchus fossil. This specimen, the holotype, was collected from the Middle Eocene Green River Formation of Sweetwater County, Wyoming on 27 July 1967 by Frank L. Pearce.

data sets are also the data sets referred to in earlier chapters, where observations and attributes of contemporary data sets are mentioned.

Tab. 7.1: Data set overview including runtimes on an ideal number of nodes and dimensionality for all involved data sets.

Data set	Size	Edge Count [million]	Runtime [seconds]	Acknowledgment
Foot	256^3	0.54	1.19	[Tie+17]
Vertebra	512^3	1.5	2.48	[Kla19]
Meteor	300^3	0.038	3.31	[PG17]
Backpack	512^2x373	4.8	6.23	Kevin Kreeger, Viatronix Inc., USA
Jet	256^2x512	0.24	4.49	[Gar20]
Aneurism	256^3	0.007	0.23	Philips Research, Hamburg, Germany
Vertebra 1024^3	1024^3	1.7	10.02	see above
Foot 1024^3	1024^3	2.2	14.64	see above
Miranda	1024^3	3.4	29	[CCM04]
Spathorhynchus	1024^2x750	30	117.33	Matthew Colbert

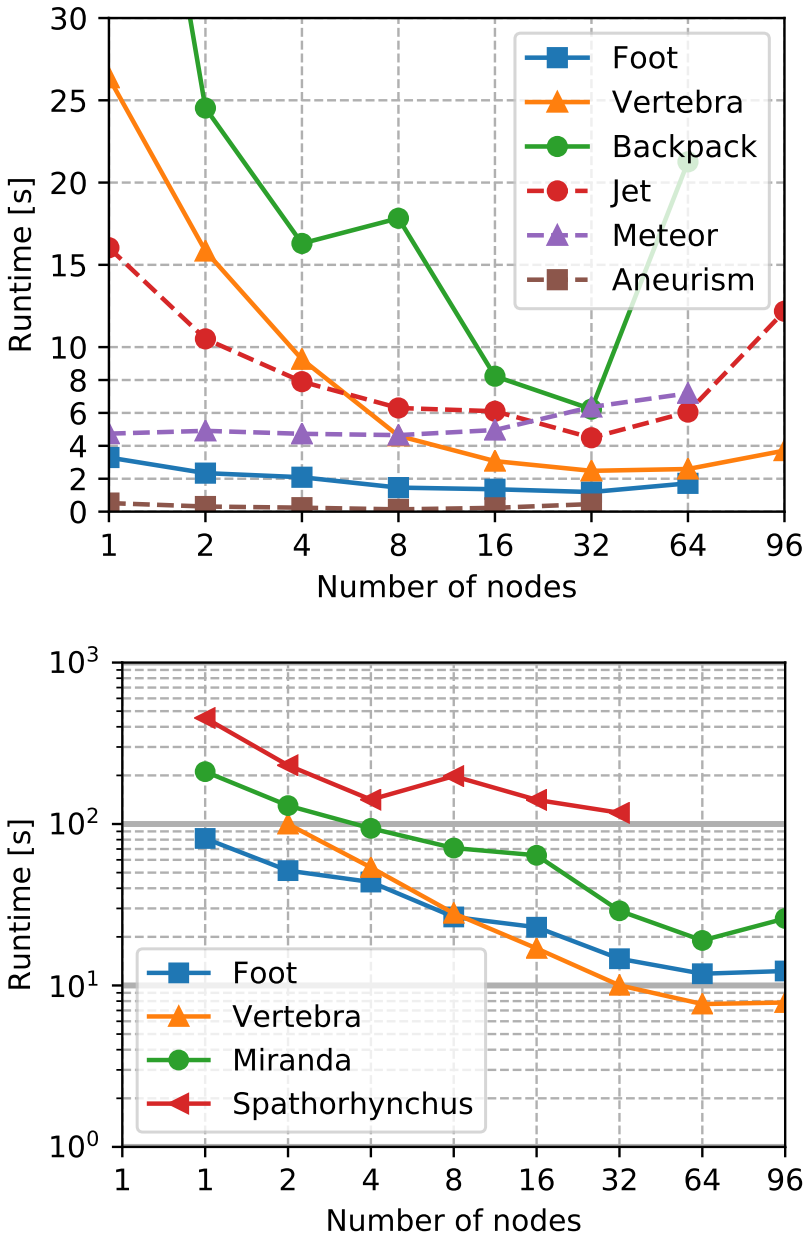


Fig. 7.9: Strong scaling for different data sets. Runtimes are illustrated for a growing number of localities showing feasible scalability on up to 96 nodes depending on data size. On the bottom, data sets Vertebra and Foot are resampled to a 1024^3 grid size.

Considering Amdahl's law, a deciding factor for the usefulness of a distributed algorithm is the turnaround point at which the total runtime is no longer reduced by additional resources, as communication and synchronization overhead exceed

parallelization benefits. The term *strong scalability* measures the impact of additional resources on runtime for a given problem size and makes this turnaround point visible. Our results show sufficient strong scalability of up to 96 nodes (2304 cores), see upper Figure 7.9, although the turnaround point is typically reached around 32 or 64 localities. As will be shown in a direct comparison to other distributed solutions below, this is an improved level of scalability, which allows us to utilize the capabilities of modern HPC hardware. This allows us to reduce the necessary runtime for augmented Merge Tree construction on large data sets by an order of magnitude over a single shared-memory node, as can bee seen in lower Figure 7.9. However, Merge Tree construction -like other topological problems- retains some global attributes and sequential dependencies which still makes distributed construction methods communication heavy. As can be seen for the Spathorhynchus data set, this limits scalability for very complex data sets (with large amounts of tree edges) and improvements to communication patterns present itself as interesting future work. For example, it might be beneficial to buffer finger-start-tasks and send them in batches between neighboring localities to reduce the number of messages.

Note, that missing entries in the diagram represent configurations, that timed out consistently. This may be due to a sharp increase in parallel overhead, due to network congestion, or similar hardware related thresholds. Such sharp communication overhead increases can for example be seen for the Backpack and Spathorhynchus data sets when reaching 8 localities. This is probably due to the network topology of the Elwetritsch cluster, where some (e.g. up to 4) localities might share a faster sub-net (e.g. on the same board).

Weak Scaling

In *weak scaling* experiments, problem sizes are grown proportionally to the number of utilized cores to determine the maximal problem sizes the algorithm can feasibly solve. Runtimes of our algorithm stayed within the same order of magnitude when scaling problem sizes up to 2 billion data points, see Figure 7.10. Again, some sudden and steep increases in runtime can be observed. This is no statistical artifact, as all data points represent the average of 20 iterations. As mentioned before, the increasing worst case round trip time in network communication will increase sharply with a growing number of involved localities. Additionally, the number of tree edges does not increase or decrease linearly when resampling data sets to different vertex counts. Weak scalability on over three thousand involved CPU cores allowed to compute the augmented Merge tree on a data set with over two billion vertices in around 20 seconds. As the problem requires/enforces unique

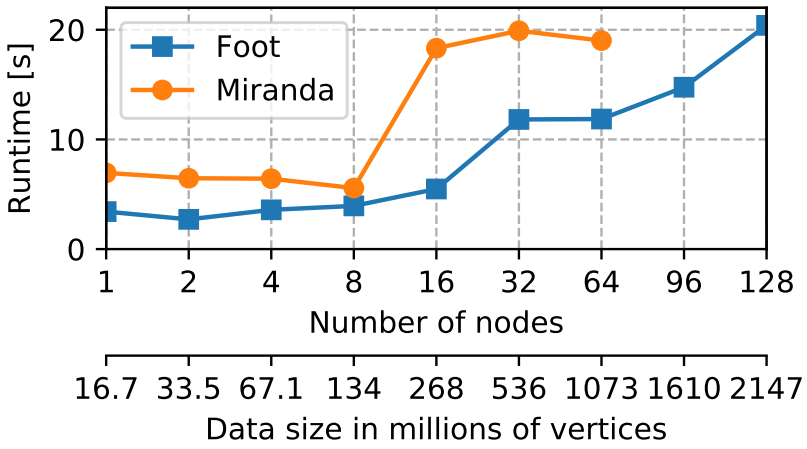


Fig. 7.10: Weak Scaling demonstrated on the Foot and Miranda data sets. To achieve adjustable data size, the Foot data set has been upsampled and the Miranda data set has been downsampled accordingly.

vertex values, the number of values a float can represent (4 billion) will become a scalability issue before the runtime of our algorithm exceeds a few minutes.

Shared Memory Performance

The sequential runtime of our novel algorithm on a single, shared memory system is almost on par with the current state-of-the-art TTK implementation [Gue+17], see Figure 7.11. Additionally, a GPU-hybrid solution as mentioned in Section 5.1 demonstrates speed ups between x5 and x20. While the algorithm is targeted towards distributed systems and faster and more readily available solutions for shared-memory systems exist, this comparison shows, that speedups of parallelization do not have to compensate for a subpar sequential runtime.

Distributed Performance

Comparison to related work in a distributed setting shows, that our novel algorithm outperforms both [Lan+14] and [MW14] significantly (compared to their reported runtimes on the volvis.org vertebra data set, see Figure 7.12). Thus, we conclude that the algorithm described here is at least competitive with the state of the art with respect to performance and scalability.

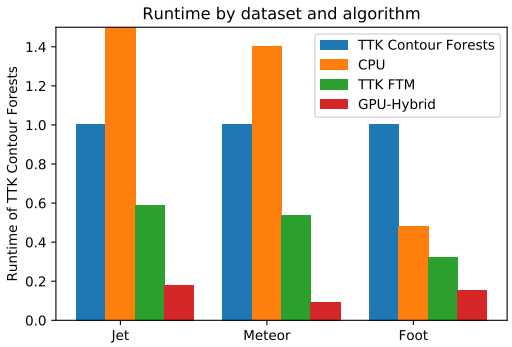


Fig. 7.11: Runtime comparison between the state of the art task parallel TTK solution (FTM) [Gue+17] and the previous TTK solution (Contour Forests) [Gue+16] with our novel solution (CPU), along with a GPU-hybrid version (GPU-hybrid). All algorithms constructed the augmented Join Tree running on a single cluster node.

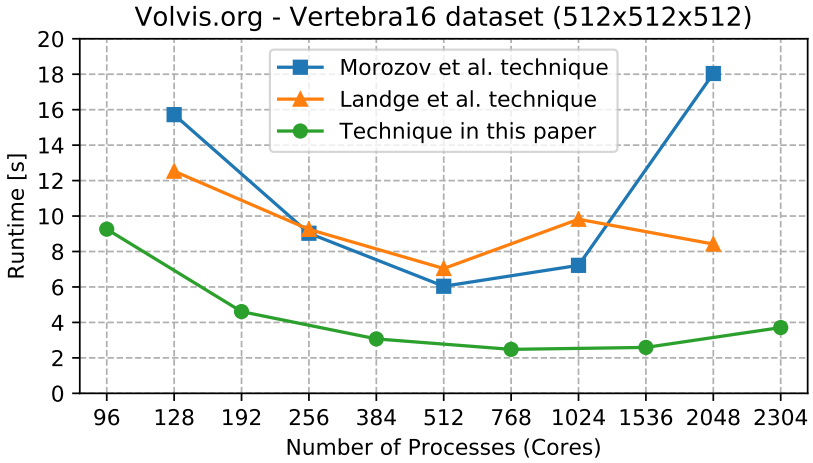


Fig. 7.12: Runtime comparison between our algorithm and reported runtimes of [Lan+14] and [MW14] on the volvis.org vertebra data set.

Alternative Simplification Parameter Overhead

Note that the results reported above used no simplification and the complete tree was constructed. With respect to simplification the fastest solution is to employ a fixed ε like is conventionally used for simplification. To allow the user to guide simplification based on p (percentage of tree edges to keep) either the BQ-Summary or the Gaussian kernel density estimation can be used. Finally, a priority queue based solution can limit the number of edges to a budget of N. Since using either p or N as simplification parameter introduces additional work, we measured runtime differences to the use of a pre-defined ε, see Table 7.2. The simplification was

performed purely symbolic, so that the measured runtime differences represent the overheads for using p or N over ε most closely.

Tab. 7.2: Total runtimes of simplification based on classical fixed threshold, BQ-Summary or kernel density estimation based percentile threshold and fixed memory budget. The overhead of simplification methods based on p or N over ε are also shown in %.

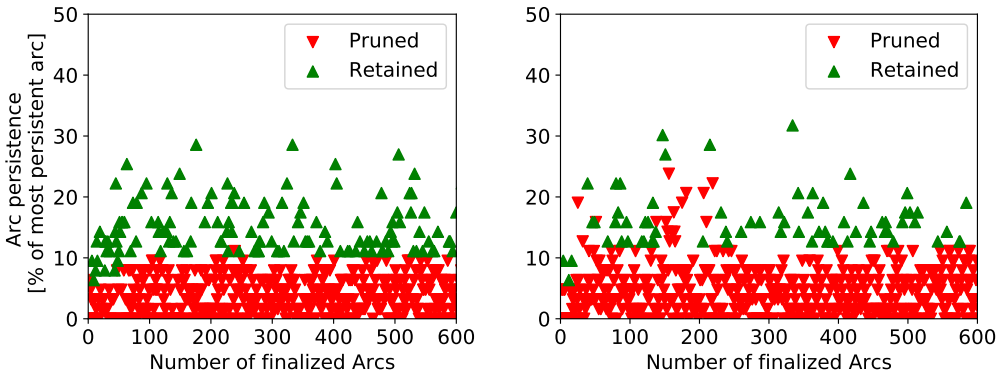
Data set	Fixed ε	BQ-Summary		Gaussian KDE		Budget N	
	[s]	[s]	[%]	[s]	[%]	[s]	[%]
Foot	6.24	7.01	12,3	7.17	14,9	6.48	3,8
Meteor	0.42	0.48	14,2	0.44	4,7	0.44	4,7
Jet	46.81	48.03	2,6	48.98	4,6	47.2	0,8

Percentile Rank Estimation Precision

Both statistical estimators need some initial information to base their estimations on. This means, that estimation precision will be low at first. Over time however, a stable estimation of a fixed value ε that corresponds to the required p in the given data is derived. In the observed data sets this stabilization is achieved very fast. Figure 7.13 shows only the first 600 edges in the stream for an application of the simplification module on a single cluster node. The used data is the foot data set and p was specified to prune all but 20% of the edges. This is only one visualized example, but the behaviour was similar for all other data sets and values of p. Note, that the kernel density estimation barely contains false negatives. The method always uses the most pessimistic estimation and thus avoids pruning edges that in fact are among the p percent most persistent ones. Similarly, this process could be inverted to focus on the most optimistic estimation and avoid to keep any edges that in fact are not among the most persistent. In contrast, the BQ-summary error is more or less symmetrical by nature creating both false positives and false negatives equally.

The overall estimation accuracy is demonstrated in Figure 7.14. The data sets Foot, Meteor and Jet have been benchmarked with a pipeline of both Join Tree construction and simplification modules. In addition, an artificial "normal" data set was created, so that the persistence of edges in the data set follows a Gaussian normal distribution. This was done to evaluate the general feasibility of the kernel density estimation approach, if applied to data that fits its normal distribution assumption perfectly.

The x-axis represents the parameter p that was used to guide the simplification. Both estimators tried to perform simplification so that p percent of edges remain after



(a) (b)

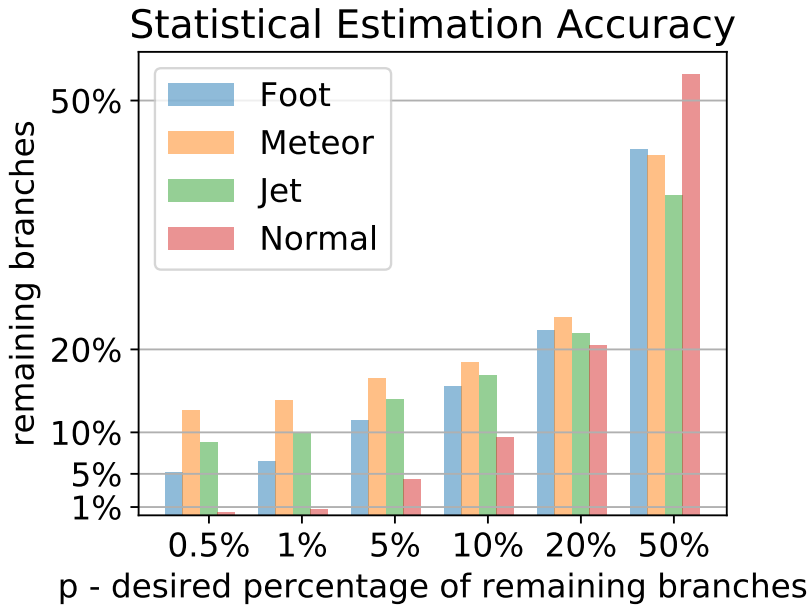
Fig. 7.13: (a) and (b) show the first 600 (of ca. 400.000) decisions/edges for the Gaussian estimation and bq-summary on the Foot data set. Each finalized arc is represented by a triangle in sequence of their arrival in the stream on the x-axis and their (relative) persistence/weight w on the y-axis. One can see some initial fluctuation, that stabilizes towards a mostly constant threshold (for the rest of the 400.000 decisions).

simplification. The x-axis spans only p lower than 50%, because the precision is symmetrical: for p larger 50% just take $p = 1 - p$ and invert the decision of keeping vs. pruning. The y-axis represents the percentage of edges that actually remained after simplification.

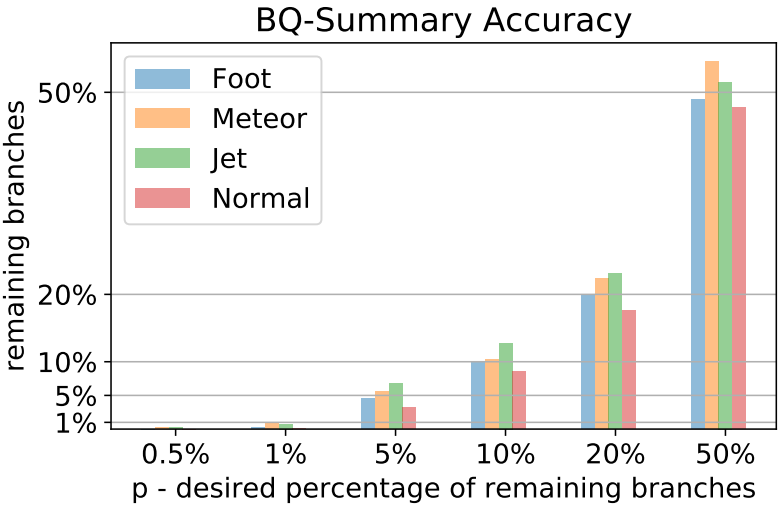
As can be seen, the Gaussian kernel density estimation method performs well for un-biased p, that is for p between 20% and 70%. Beyond that, the skewness of real-world persistence distributions hurts estimation accuracy. The pessimistic "keep rather than prune" approach makes the method keep between 5% and 10% on real world data, even if p is as small as 0.5%. For very noisy data, where over 90% of edges are considered noise this approach is not feasible. However, please note that the estimation works almost perfectly for the artifical data where persistence is actually distributed normally among edges and skewness is low. This suggests, that the kernel density estimation approach could perform well for real world data sets, if the Gaussian kernel is replaced with a distribution that fits real world persistence distribution better.

The BQ-Summary approach on the other hand performs rather well, even for biased p. The percentage of edges remaining after simplification was always close to the target p. This is the case for some data sets even when considering 99% of the edges as noise. A reduction in tree size by 95% was successful in all tests for all data sets. However, for extremely biased p the risk grows, that the estimator calculates an ε too high and prunes every single edge from the tree. Data simplification of this

degree maybe needs to be addressed by further user interaction, after the heavy burden of data complexity was reduced by a less biased p. This improved precision over the kernel density estimation comes at the price of a larger runtime overhead for maintaining the BQ-Summary.



(a)



(b)

Fig. 7.14: (a) and (b) show achieved estimation accuracy for the Gaussian estimation and the BQ-Summary with different p on four data sets.

7.2 Summary, Future and Ongoing Work

We laid the groundwork for a framework for task-parallel and distributed visualization and data analysis pipeline management. We evaluated HPX as a task-parallel runtime to base the framework on and identified topological simplification as a prerequisite for most continuative pipeline stages. As task-parallel, distributed Merge Tree construction was still an open problem, we had to completely revisit Merge Tree construction and derive a novel insight and algorithm, to fulfill this requirement. We supplied efficient and scalable modules for Merge Tree construction and topological simplification and introduced alternative parameters for topological simplification, to increase flexibility in our targeted use case of large data analysis.

Going beyond availability in our framework, the resulting Merge Tree construction brought the benefits of contemporary task-parallel approaches to a distributed setting and achieved improved performance and scalability over existing distributed techniques, contributing to the state of the art in Merge Tree construction.

There are multiple opportunities to further improve performance and scalability of our implementation. From buffered communication to the distributed use of GPUs (per locality) these opportunities have been discussed in this thesis in the appropriate sections. The most important prerequisite to efficiently go forward with this work is a detailed profiling toolchain that can work with the challenging setting of task-parallel and distributed programs. Conventional profilers and Gantt charts are not applicable to or insufficient for the analysis of task parallel programs, as tasks can for example suspend and continue on different OS-threads and have no clear hierarchical nesting. The profiler APEX that is shipped with HPX is not usable out of the box and is still subject to regular mailing list issues. The absence of such a toolchain also limited the level of detail in benchmark and result acquisition.

The author collaborated on a Bachelors thesis that produced a prototype for a novel profiling tool that specifically focuses on interactive exploration of task-parallel dependency and execution graphs. At the time of writing, the author is collaborating on two additional Bachelor theses continuing this work.

Additionally, the author collaborated on work (primarily Bachelor and Master theses) with the goal of producing other modules for task-parallel, distributed visualization pipelines. These works include modules for iso-surface extraction, volume rendering [BG15], integral lines in vector fields and a rendering component [Ebe20]. An actual framework -that is runtime, API and GUI- to allow for user-defined pipelines built

from these modules is currently worked on in collaboration with the author in the scope of an other PhD thesis.

There are also some future work opportunities that are primarily based on the contributions in this thesis rather than the pipeline framework efforts. The task-parallel partially ordered Merge Tree construction [Gue+17] has been adjusted to Reeb Graph construction [Gue+19] and it poses an interesting question, whether this adjustment can be made in a similar way to our work, to bring Reeb Graph construction to distributed systems. During the work on this thesis, many weeks have been invested into improving the Merge Tree combination step, where the Contour Tree is constructed. Ultimately, we were unable to identify a method with improved concurrency or better suitability for distributed systems. This is mainly due to the problems posed by W-structures in Contour Trees [HC20]. These structures are also behind the divergence of the different simplification methods (see Section 3.5).

Lastly, a statistical kernel function that better represents the distribution of persistence among Merge Tree edges would greatly benefit the estimation accuracy of the kernel density estimation approach presented above. This would allow for a faster and more accurate use of the simplification parameter p instead of ε.

7.3 Acknowledgements

Work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 398122172.

Thank you for reading.

Bibliography

[AN15] Aditya Acharya and Vijay Natarajan. “A parallel and memory efficient algorithm for constructing the contour tree”. In: *2015 IEEE Pacific Visualization Symposium (PacificVis)* (2015), pp. 271–278 (cit. on pp. 32, 46).

[Amd07] Gene M. Amdahl. “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, N.J., Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California”. In: *IEEE Solid-State Circuits Society Newsletter* 12.3 (2007), pp. 19–20 (cit. on p. 5).

[BLW12] Ulrich Bauer, Carsten Lange, and Max Wardetzky. “Optimal Topological Simplification of Discrete Functions on Surfaces”. In: *Discrete & Computational Geometry* 47.2 (Mar. 2012), pp. 347–377 (cit. on p. 24).

[BG15] Tim Biedert and Christoph Garth. “Contour Tree Depth Images For Large Data Visualization”. In: *Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by C. Dachsbacher and P. Navrátil. The Eurographics Association, 2015 (cit. on pp. 23, 97).

[Bre+11] Peer-Timo Bremer, Gunther Weber, Julien Tierny, et al. “Interactive Exploration and Analysis of Large-Scale Simulations Using Topology-Based Data Segmentation”. In: *IEEE Transactions on Visualization and Computer Graphics* 17.9 (2011), pp. 1307–1324 (cit. on p. 23).

[Car+16a] H. Carr, C. Sewell, L-T. Lo, and J. Ahrens. “Hybrid Data-parallel Contour Tree Computation”. In: *Proceedings of the Conferece on Computer Graphics & Visual Computing*. CGVC ’16. Bournemouth, United Kingdom: Eurographics Association, 2016, pp. 73–80 (cit. on p. 52).

[Car+16b] H. A. Carr, G. H. Weber, C. M. Sewell, and J. P. Ahrens. “Parallel peak pruning for scalable SMP contour tree computation”. In: *IEEE Symposium on Large Data Analysis and Visualization 2016, LDAV 2016*. IEEE, 2016 (cit. on pp. 32, 46, 48, 53, 55, 56, 66).

[CS03] Hamish Carr and Jack Snoeyink. “Path Seeds and Flexible Isosurfaces Using Topology for Exploratory Visualization”. In: *Eurographics / IEEE VGTC Symposium on Visualization*. Ed. by G.-P. Bonneau, S. Hahmann, and C. D. Hansen. The Eurographics Association, 2003 (cit. on p. 40).

[CSA03] Hamish Carr, Jack Snoeyink, and Ulrike Axen. “Computing contour trees in all dimensions”. In: *Computational Geometry* 24.2 (2003). Special Issue on the Fourth CGC Workshop on Computational Geometry, pp. 75–94 (cit. on pp. 19, 22, 32, 34, 36).

[CSP10] Hamish Carr, Jack Snoeyink, and Michiel van de Panne. “Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree”. In: *Computational Geometry* 43.1 (2010). Special Issue on the 14th Annual Fall Workshop, pp. 42–58 (cit. on p. 26).

[CSP04] Hamish Carr, Jack Snoeyink, and Michiel van de Panne. “Simplifying Flexible Isosurfaces Using Local Geometric Measures”. In: *Proceedings of the Conference on Visualization ’04*. VIS ’04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 497–504 (cit. on p. 80).

[Car+19] Hamish Carr, Gunther Weber, Christopher Sewell, et al. “Scalable Contour Tree Computation by Data Parallel Peak Pruning”. In: *IEEE transactions on visualization and computer graphics* (Nov. 2019) (cit. on pp. 34, 48).

[CGL83] Tony F. Chan, Gene H. Golub, and Randall J. Leveque. “Algorithms for Computing the Sample Variance: Analysis and Recommendations”. In: *The American Statistician* 37.3 (1983), pp. 242–247. eprint: `https://amstat.tandfonline.com/doi/pdf/10.1080/00031305.1983.10483115` (cit. on p. 82).

[Chi+05] Yi-Jen Chiang, Tobias Lenz, Xiang Lu, and Günter Rote. “Simple and optimal output-sensitive construction of contour trees using monotone paths”. In: *Computational Geometry* 30 (Feb. 2005), pp. 165–195 (cit. on pp. 22, 33, 40, 42, 46, 47, 50, 56).

[CEH07] David Cohen-Steiner, Herbert Edelsbrunner, and John Harer. “Stability of Persistence Diagrams”. In: *Discrete & Computational Geometry* 37.1 (Jan. 2007), pp. 103–120 (cit. on p. 24).

[CCM04] Andrew W. Cook, William Cabot, and Paul L. Miller. “The mixing transition in Rayleigh-Taylor instability”. In: *Journal of Fluid Mechanics* 511 (2004), pp. 333–362 (cit. on p. 89).

[Cor+06] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. “Space- and Time-efficient Deterministic Algorithms for Biased Quantiles over Data Streams”. In: *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS ’06. Chicago, IL, USA: ACM, 2006, pp. 263–272 (cit. on p. 81).

[Ebe20] Kevin Eberle. “A Task-Parallel Distributed Rendering and Compositing Module”. MA thesis. Germany: TU Kaiserslautern, 2020 (cit. on p. 97).

[EJ09] H. Edelsbrunner and J.Harer. *Computational Topology: An Introduction*. American Mathematical Society, 2009 (cit. on p. 2).

[Ede+03] Herbert Edelsbrunner, John Harer, Vijay Natarajan, and Valerio Pascucci. “Morse-smale Complexes for Piecewise Linear 3-manifolds”. In: *Proceedings of the Nineteenth Annual Symposium on Computational Geometry*. SCG ’03. San Diego, California, USA: ACM, 2003, pp. 361–370 (cit. on p. 18).

[EMP06] Herbert Edelsbrunner, Dmitriy Morozov, and Valerio Pascucci. “Persistence-sensitive Simplification Functions on 2-manifolds”. In: *Proceedings of the Twenty-second Annual Symposium on Computational Geometry*. SCG ’06. Sedona, Arizona, USA: ACM, 2006, pp. 127–134 (cit. on p. 24).

[ELZ02] Edelsbrunner, Letscher, and Zomorodian. “Topological Persistence and Simplification”. In: *Discrete & Computational Geometry* 28.4 (Sept. 2002), pp. 511–533 (cit. on p. 23).

[FR96] Michael J Flynn and Kevin W Rudd. “Parallel architectures”. In: *ACM computing surveys (CSUR)* 28.1 (1996), pp. 67–70 (cit. on p. 8).

[Gar20] Christoph Garth. *Simulation of a jet flow*. 2020 (cit. on pp. 88, 89).

[Gue+17] C. Gueunet, P. Fortin, J. Jomier, and J. Tierny. “Task-based augmented merge trees with Fibonacci heaps”. In: *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*. Oct. 2017, pp. 6–15 (cit. on pp. 16, 32, 34, 50, 55, 56, 60, 65, 66, 92, 93, 98).

[Gue+16] C. Gueunet, P. Fortin, J. Jomier, and Vijay. “Contour forests: Fast multi-threaded augmented contour trees”. In: *IEEE Symposium on Large Data Analysis and Visualization 2016, LDAV 2016*. IEEE, 2016 (cit. on pp. 34, 47, 93).

[Gue+19] Charles Gueunet, Pierre Fortin, Julien Jomier, and Julien Tierny. “Task-based Augmented Reeb Graphs with Dynamic ST-Trees”. In: *Eurographics Symposium on Parallel Graphics and Visualization*. Porto, Portugal, June 2019 (cit. on pp. 16, 98).

[Hei+16] C. Heine, H. Leitte, M. Hlawitschka, et al. “A Survey of Topology-based Methods in Visualization”. In: *Comput. Graph. Forum* 35.3 (June 2016), pp. 643–667 (cit. on p. 2).

[Hil+01] Masaki Hilaga, Yoshihisa Shinagawa, Taku Kohmura, and Tosiyasu L. Kunii. “Topology Matching for Fully Automatic Similarity Estimation of 3D Shapes”. In: SIGGRAPH ’01. New York, NY, USA: Association for Computing Machinery, 2001, pp. 203–212 (cit. on p. 23).

[HC20] P. Hristov and H. Carr. “W-Structures in Contour Trees”. In: *Topological Methods in Data Analysis and Visualization VI*. Mathematics and Visualization. Cham, Switzerland: Springer, Aug. 2020 (cit. on p. 98).

[KBS09] Hartmut Kaiser, Maciej Brodowicz, and Thomas Sterling. “ParalleX An Advanced Parallel Execution Model for Scaling-Impaired Applications”. In: Oct. 2009, pp. 394–401 (cit. on p. 6).

[Kla19] Pavol Klacansky. *Open SciVis Datasets*. `https://klacansky.com/open-scivis-datasets/`. Apr. 2019 (cit. on p. 89).

[Kre+97] Marc van Kreveld, René van Oostrum, Chandrajit Bajaj, Valerio Pascucci, and Dan Schikore. “Contour Trees and Small Seed Sets for Isosurface Traversal”. In: *Proceedings of the Thirteenth Annual Symposium on Computational Geometry*. SCG ’97. Nice, France: Association for Computing Machinery, 1997, pp. 212–220 (cit. on p. 23).

[Lan+14] Aaditya G. Landge, Valerio Pascucci, Attila Gyulassy, et al. “In-Situ Feature Extraction of Large Scale Combustion Simulations Using Segmented Merge Trees”. In: *SC*. IEEE Computer Society, 2014, pp. 1020–1031 (cit. on pp. 34, 44, 46, 47, 53, 92, 93).

[Luk+21] Jonas Lukasczyk, Christoph Garth, Ross Maciejewski, and Julien Tierny. “Localized Topological Simplification of Scalar Data”. In: *IEEE Transactions on Visualization and Computer Graphics* 27.2 (2021), pp. 572–582 (cit. on pp. 25, 28, 78).

[MDN12] S. Maadasamy, H. Doraiswamy, and V. Natarajan. “A hybrid parallel algorithm for computing and tracking level set topology”. In: *2012 19th International Conference on High Performance Computing*. 2012, pp. 1–10 (cit. on pp. 41, 45–47).

[MW14] D. Morozov and G. Weber. “Distributed Contour Trees”. In: *Topological Methods in Data Analysis and Visualization III*. 2014, pp. 89–102 (cit. on pp. 42, 45–47, 53, 55, 68, 73, 92, 93).

[MW13] Dmitriy Morozov and Gunther Weber. “Distributed Merge Trees”. In: *SIGPLAN Not.* 48.8 (Feb. 2013), pp. 93–102 (cit. on p. 42).

[93] “MPI: A message passing interface”. In: *Supercomputing ’93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. 1993, pp. 878–883 (cit. on p. 7).

[Nat+16] Abhinandan Nath, Kyle Fox, Pankaj K. Agarwal, and Kamesh Munagala. “Massively Parallel Algorithms for Computing TIN DEMs and Contour Trees for Large Terrains”. In: *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPACIAL ’16. Burlingame, California: Association for Computing Machinery, 2016 (cit. on p. 52).

[PCS05] Valerio Pascucci, Kree Cole-McLaughlin, and Giorgio Scorzelli. “Multi-Resolution computation and presentation of Contour Trees”. In: 2005 (cit. on p. 23).

[Pas+07] Valerio Pascucci, Giorgio Scorzelli, Peer-Timo Bremer, and Ajith Mascarenhas. “Robust On-line Computation of Reeb Graphs: Simplicity and Speed”. In: *ACM Trans. Graph.* 26.3 (July 2007) (cit. on p. 77).

[Pas+11] Valerio Pascucci, Xavier Tricoche, Hans Hagen, and Julien Tierny. *Topological Methods in Data Analysis and Visualization: Theory, Algorithms, and Applications*. 1st. Springer Publishing Company, Incorporated, 2011 (cit. on p. 2).

[PG17] John Patchett and Galen Gisler. *Deep Water Impact Ensemble Data Set*. Tech. rep. LA-UR-17-21595. Feb. 14, 2017. published (cit. on pp. 88, 89).

[Pug89] W. Pugh. “Concurrent Maintenance of Skip Lists”. In: *Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, CS-TR-2222.1* (1989) (cit. on p. 80).

[RS14] Benjamin Raichel and C. Seshadhri. “A Mountaintop View Requires Minimal Sorting: A Faster Contour Tree Algorithm”. In: *CoRR* abs/1411.2689 (2014) (cit. on pp. 34, 45, 48, 49, 53, 55, 66).

[RTP18] Paul Rosen, Junyi Tu, and Les Piegl. “A Hybrid Solution to Parallel Calculation of Augmented Join Trees of Scalar Fields in Any Dimension”. In: *Computer-Aided Design and Applications* 15.1 (2018), pp. 610–618 (cit. on pp. 34, 50).

[Ros+17] Paul Rosen, Bei Wang, Anil Seth, et al. “Using Contour Trees in the Analysis and Visualization of Radio Astronomy Data Cubes”. In: (Apr. 2017) (cit. on p. 23).

[Rou72] Brian Rourke Colin amd Sanderson. *Introduction to Piecewise-Linear Topology*. Springer-Verlag, 1972 (cit. on p. 18).

[SSW14] Himangshu Saikia, Hans-Peter Seidel, and Tino Weinkauf. "Extended Branch Decomposition Graphs: Structural Comparison of Scalar Data". In: *Computer Graphics Forum* 33.3 (2014), pp. 41–50 (cit. on p. 23).

[Sar+08] R. Sarkar, X. Zhu, J. Gao, L. J. Guibas, and J.S.B. Mitchell. "Iso-Contour Queries and Gradient Descent with Guaranteed Delivery in Sensor Networks". In: *INFOCOM 2008. The 27th Conference on Computer Communications*. IEEE, 2008 (cit. on p. 53).

[Sch+06] William J. Schroeder, Ken Martin, William E. Lorensen, Lisa Sobierajski Avila, and Kenneth W. Martin. *The visualization toolkit. an object-oriented approach to 3D graphics ; [visualize data in 3D - medical, engineering or scientific ; build your own applications with C++, Tcl, Java or Python ; includes source code for VTK (supports UNIX, Windows and Mac)]*. eng. 4. ed. Literaturangaben. [Clifton Park, NY]: Kitware, 2006, XVI, 512 S. (Cit. on p. 2).

[Sol+18] Maxime Soler, Mélanie Plainchault, Bruno Conche, and Julien Tierny. "Topologically Controlled Lossy Compression". In: *2018 IEEE Pacific Visualization Symposium (PacificVis)*. 2018, pp. 46–55 (cit. on p. 23).

[TTF04] Shigeo Takahashi, Yuriko Takeshima, and Issei Fujishiro. "Topological Volume Skeletonization and Its Application to Transfer Function Design". In: *Graph. Models* 66.1 (Jan. 2004), pp. 24–49 (cit. on p. 26).

[TN14] Dilip Mathew Thomas and Vijay Natarajan. "Multiscale Symmetry Detection in Scalar Fields by Clustering Contours". In: *IEEE Transactions on Visualization and Computer Graphics* 20.12 (2014), pp. 2427–2436 (cit. on p. 23).

[Tie18] J. Tierny. *Topological Data Analysis for Scientific Visualization*. Springer, 2018 (cit. on p. 2).

[Tie+17] Julien Tierny, Guillaume Favelier, Joshua A Levine, Charles Gueunet, and Michael Michaux. "The Topology ToolKit". In: *IEEE Transactions on Visualization and Computer Graphics (`https://topology-tool-kit.github.io/`)* (2017) (cit. on pp. 52, 89).

[TP12] Julien Tierny and Valerio Pascucci. "Generalized Topological Simplification of Scalar Fields on Surfaces". In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (Dec. 2012), pp. 2005–2013 (cit. on p. 25).

[TVD06] Julien Tierny, Jean-Philippe Vandeborre, and Mohamed Daoudi. "3D Mesh Skeleton Extraction Using Topological and Geometrical Analyses". In: (Oct. 2006), pp. 85–94 (cit. on p. 23).

[Tuk77] John W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977 (cit. on p. 82).

[TS05] Tony Tung and Francis Schmitt. "The Augmented Multiresolution Reeb Graph Approach for Content-based Retrieval of 3d Shapes". In: *International Journal of Shape Modeling* 11.1 (2005), pp. 91–120 (cit. on p. 23).

[V P03] K. Cole-McLaughlin V. Pascucci. “Parallel Computation of the Topology of Level Sets”. In: *Algorithmica* 38(1) (2003), pp. 249–268 (cit. on pp. 22, 34, 39, 41, 43, 53).

[WBP07] Gunther Weber, Peer-Timo Bremer, and Valerio Pascucci. “Topological Landscapes: A Terrain Metaphor for Scientific Data”. In: *IEEE Transactions on Visualization and Computer Graphics* 13.6 (Sept. 2007), pp. 1416–1423 (cit. on p. 23).

[Web+06] Gunther Weber, Scott E. Dillard, Hamish A. Carr, Valerio Pascucci, and Bernd Hamann. “Topology-Controlled Volume Rendering”. In: *IEEE Transactions on Visualization and Computer Graphics* 13 (2006), pp. 330–341 (cit. on p. 23).

[Wer20] Kilian Werner. *Reproducible Source Code for Unordered Task-Parallel Augmented Merge Tree Construction*. Available at `https://codeocean.com/capsule/0498480/tree/v1`. 2020 (cit. on p. 88).

[WG20] Kilian Werner and Christoph Garth. “Alternative Parameters for On-The-Fly Simplification of MergeTrees”. In: *Eurographics Symposium on Parallel Graphics and Visualization*. Ed. by Steffen Frey, Jian Huang, and Filip Sadlo. The Eurographics Association, 2020 (cit. on p. 80).

[WG21] Kilian Werner and Christoph Garth. “Unordered Task-Parallel Augmented Merge Tree Construction”. In: *IEEE Transactions on Visualization and Computer Graphics* (2021), pp. 1–1 (cit. on p. 54).

[Wid+12] Wathsala Widanagamaachchi, Cameron Christensen, Valerio Pascucci, and Peer-Timo Bremer. “Interactive exploration of large-scale time-varying data using dynamic tracking graphs”. In: *Large Data Analysis and Visualization (LDAV), 2012 IEEE Symposium on*. Oct. 2012, pp. 9–17 (cit. on p. 23).

[ZMT05] Eugene Zhang, Konstantin Mischaikow, and Greg Turk. “Feature-based Surface Parameterization and Texture Mapping”. In: *ACM Trans. Graph.* 24.1 (Jan. 2005), pp. 1–27 (cit. on p. 23).

List of Figures

List of Tables

Kilian Werner received the bachelor's and master's degrees in computer science from Technische Universität Kaiserslautern, in 2016 and 2018, respectively. He is currently a PhD student there. His research interests include topology-based methods in visualization, large-scale data analysis and scientific visualization.

Colophon

This thesis was typeset with LaTeX 2$_\varepsilon$. It uses the *Clean Thesis* style developed by Ricardo Langner. The design of the *Clean Thesis* style is inspired by user guide documents from Apple Inc.

Download the *Clean Thesis* style at `http://cleanthesis.der-ric.de/`.