Scalable Extraction and Visualization of Scientific Features with Load-Balanced Parallelism

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Jiayi Xu, B.E.

Graduate Program in Computer Science and Engineering

The Ohio State University

2021

Dissertation Committee:

Han-Wei Shen, Advisor

Rephael Wenger

Jian Chen

Abstract

Extracting and visualizing features from scientific data can help scientists derive valuable insights. An extraction and visualization pipeline usually includes three steps: (1) scientific feature detection, (2) union-find for features' connected component labeling, and (3) visualization and analysis. As the scale of scientific data generated by experiments and simulations grows, it becomes a common practice to use distributed computing to handle large-scale data with data-parallelism, where data is partitioned and distributed over parallel processors.

Three challenges arise for feature extraction and visualization on scientific applications. First, traditional feature detectors may not be effective and robust enough to capture features of interest across different scientific settings, because scientific features usually are highly nonlinear and recognized by domain scientists' soft knowledge. Second, existing union-find algorithms are either serial or not scalable enough to deal with extreme-scale datasets generated in the modern era. Third, existing parallel feature extraction and visualization algorithms fail to automatically reduce communication costs when optimizing the performance of processing units. This dissertation studies scalable scientific feature extraction and visualization to tackle the three challenges.

First, we design human-centric interactive visual analytics based on scientists' requirements to address domain-specific feature detection and tracking. We focus on an essential problem in earth sciences: spatiotemporal analysis of viscous and

gravitational fingers. Viscous and gravitational flow instabilities cause a displacement front to break up into finger-like fluids. Previously, scientists mainly detected the finger features using density thresholding, where scientists specify certain density thresholds and extract super-level sets from input density scalar fields. However, the results of density thresholding are sensitive to the selected threshold values, and a few single threshold values are usually not sufficient to extract and track satisfied time-varying finger features. In our study, scientists can detect and visualize spatiotemporal fingers interactively to elucidate the dynamics of the flow instabilities. Our study has two main contributions. (1) We propose a ridge-guided detection to extract curvilinear geometry and branching topology of fingers, which provides richer geometric structures than the density thresholding. (2) We devise an interactive visual-analytics system with geometric-glyph augmented tracking graphs to allow scientists to navigate how the fingers and their branches grow, merge, and split over both space and time. Feedback from earth scientists demonstrates the efficacy of our approach for spatiotemporal geometry-driven analyses of fingers.

Second, we improve the scalability of union-find algorithms using asynchronous and load-balanced parallelism. Union-find is widely used in scientific feature extraction and visualization techniques, such as tracking critical points and extracting level sets. However, distributed and parallel union-find can suffer from high synchronization costs and imbalanced workloads of participating processors. In our study, we present a novel distributed union-find algorithm that features asynchronous parallelism and k-d tree based load balancing for scalable scientific feature extraction and visualization. We prove that global synchronizations in existing distributed union-find can be eliminated without changing final results, allowing overlapped communications and computations

for scalable processing. We also use a k-d tree decomposition to redistribute inputs in order to improve workload balancing. We benchmark the scalability of our algorithm with up to 1,024 processors using both synthetic and application data. We demonstrate the use of our algorithm in critical point tracking and super-level set extraction with high-speed imaging experiments and fusion plasma simulations, respectively.

Third, we take communication costs into account of parallel algorithm design. We explore an online reinforcement learning (RL) paradigm to optimize parallel particle tracing performance dynamically in distributed-memory systems with the reduction of I/O and communication costs. Our method combines three novel components: (1) a workload donation model, (2) a high-order workload estimation model, and (3) a communication cost model. First, our RL-based workload donation model monitors the workloads of processors and creates RL agents to donate particles and data blocks from high-workload processors to low-workload processors to minimize the execution time. The RL agents learn the donation strategy on-the-fly based on reward and cost functions. The reward and cost functions are designed to consider processors' workload changes and data transfer costs for every donation action. Second, we propose an online workload estimation model to help our RL model estimate the workload distribution of processors in future computations. Third, we use the communication cost model that considers both block and particle data exchange costs to help the agents make effective decisions with minimized communication costs. We demonstrate that our algorithm adapts to different flow behaviors in large-scale fluid dynamics, ocean, and weather simulation data. Our algorithm improves parallel particle tracing performance in terms of parallel efficiency, load balance, and costs of I/O and communication for evaluations up to 16,384 processors.

Dedicated to my parents, Junsheng Xu and Guifen Luan

Acknowledgments

My doctoral study at The Ohio State University totally changes my philosophy of life. I would like to give my deepest appreciation to many people who gave me guidance, help, and support in this special doctoral journey.

Above all, I would like to express my sincere gratitude to my advisor Prof. Han-Wei Shen for his constant support throughout my graduate studies and for helping me shape my research career. His vision, suggestion, and guidance were invaluable in helping me formulate the research methodology and have continuously inspired me. Besides, I would like to thank my dissertation committee members, Prof. Rephael Wenger and Prof. Jian Chen, for providing insightful comments and suggestions to help advance my dissertation. I also thank Prof. Stéphane Lavertu for serving as the graduate faculty representative for my dissertation defense.

I must also acknowledge and thank my mentors during my summer internships. The internship experience at Argonne National Laboratory (ANL) contributed significantly to my doctoral study. I would particularly like to single out my mentors, Dr. Hanqi Guo and Dr. Tom Peterka, who exemplified how to be a good researcher and mentored me in distilling research ideas and writing clear and concise scientific papers. Besides, I would like to thank my mentor, Dr. Jonathan L. Woodring, at Los Alamos National Laboratory (LANL), and my mentor, Dr. Yen-Jung Chang, at Facebook AI for the

opportunities they have given me. I am grateful for their valuable suggestions and patient support, which have helped to improve my research.

I have been privileged to join GRAVITY (GRAphics & VIsualization sTudY) research group, led by Prof. Shen. I would like to thank my coauthors for stimulating and thorough discussions and for the days and nights we were working hard together before paper submissions, Soumya Dutta, Ko-Chih Wang, Guan Li, Neng Shi, Yamei Tu, Ziwei Li, Wenbin He, and Skylar W. Wurster. I also would like to thank my fellow lab-mates for encouraging discussions on exciting research topics and for celebrations together, Junpeng Wang, Subhashis Hazarika, Xiaonan Ji, Xiaotong Liu, Ayan Biswas, Chun-Ming Chen, Tzu-Hsuan Wei, Kewei Lu, Xin Tong, Cheng Li, Rui Li, Jingyi Shen, Runqi Wang, Haoyu Li, Yifei An, Ming-Yi Su, Hyunjean Choi, Piyush Chawla, Xiaoqi Wang, Yi-Tang Chen, Tianyu Xiong, and Rui Qiu.

I have been fortunate to meet wonderful people and make long-lasting friendships during my doctoral study. I would like to sincerely thank my collaborators who gave me help and suggestion on my research, Yujia Wang, Prof. Joachim Moortgat, Amin Amooie, Mukund Raj, Xueyun Wang, Xueqiao Xu, Eric Brugger, and Zhehui Wang. Special thanks go to my friend Yanyunwen He for taking the time to help me review and prepare my dissertation.

I would like to extend my sincere gratitude to advisors during my undergraduate year. I would like to thank Prof. Wei Chen and Prof. Hujun Bao for giving me the opportunity and motivation to work on research. I would like to thank Prof. Huamin Qu for inspiring me to pursue research in computer science and sharpening my critical thinking. I would like to thank my lab-mates whom I have worked and interacted with, Yuxin Ma, Haidong Chen, Dichao Peng, Zhiyu Ding, Wenchao Wu, Haipeng

Zeng, Bing Ni, Yixian Zheng, Yong Wang, Qiaomu Shen, and Yanhong Wu. I also would like to thank all my teachers and mentors from my undergraduate university and high schools who have helped me throughout my student life.

Last but not least, I would like to thank the unconditional love, care, and encouragement from my father, Junsheng Xu, my mother, Guifen Luan, my aunt, Haifeng Luan, and all my family members. They always provided the strongest support to me and respected my decisions in my life. I am truly lucky and blessed to have them in my life. This dissertation is as much a realization of my dreams as theirs.

I conclude this by acknowledging all who gave me help and support but not mentioning within this short space.

Vita

June 4th, 1991	Born, Taixing, Jiangsu, China.
2010 - 2014	B.E. in Computer Science and Technology, Chu Kochen Honors College, Zhejiang University, Hangzhou, Zhejiang, China.
2014 - 2015	Research Assistant, The Hong Kong University of Science and Technology, Kowloon, Hong Kong, China.
2015 - 2016, 2017 - 2018, Spring 2021	Graduate Teaching Associate, The Ohio State University, Columbus, OH, USA.
2016 - 2017, 2018 - 2020, Autumn 2021	Graduate Research Associate, The Ohio State University, Columbus, OH, USA.
May - August, 2018	Research Internship, Los Alamos National Laboratory, Los Alamos, NM, USA.
May - August, 2019	Research Internship, Argonne National Laboratory, Lemont, IL, USA.
May - August, 2021	Research Internship, Facebook AI, Menlo Park, CA, USA.

Publications

Research Publications

Jiayi Xu, Hanqi Guo, Han-Wei Shen, Mukund Raj, Xueyun Wang, Xueqiao Xu, Zhehui Wang, and Tom Peterka. "Asynchronous and Load-Balanced Union-Find for Distributed and Parallel Scientific Data Visualization and Analysis", *IEEE Transactions on Visualization and Computer Graphics*, 27(6): 2808-2820, 2021.
[Best Paper Award at the 14th IEEE Pacific Visualization Symposium].

Jiayi Xu, Hanqi Guo, Han-Wei Shen, Mukund Raj, Skylar W. Wurster, and Tom Peterka. "Reinforcement Learning for Load-balanced Parallel Particle Tracing", *arXiv preprint*, 2109.05679, 2021 (Under Review at IEEE Transactions on Visualization and Computer Graphics).

Hanqi Guo, David Lenz, Jiayi Xu, Xin Liang, Wenbin He, Iulian R Grindeanu, Han-Wei Shen, Tom Peterka, Todd Munson, and Ian Foster. "FTK: A Simplicial Spacetime Meshing Framework for Robust and Scalable Feature Tracking", *IEEE Transactions on Visualization and Computer Graphics*, 27(8): 3463-3480, 2021.

Yamei Tu, Jiayi Xu, Han-Wei Shen. "KeywordMap: Attention-based Visual Exploration for Keyword Analysis", In *Proc. IEEE Pacific Visualization Symposium*, pages 206-215, 2021.

Dmitriy Morozov, Tom Peterka, Hanqi Guo, Mukund Raj, Jiayi Xu, and Han-Wei Shen. "IExchange: Asynchronous Communication and Termination Detection for Iterative Algorithms", In *Proc. IEEE Symposium on Large Data Analysis and Visualization*, 2021 (Accepted).

Jiayi Xu, Soumya Dutta, Wenbin He, Joachim Moortgat, and Han-Wei Shen. "Geometry-Driven Detection, Tracking and Visual Analysis of Viscous and Gravitational Fingers", *IEEE Transactions on Visualization and Computer Graphics*, 2020 (Early Access).

Zhehui Wang, Jiayi Xu, Yao E. Kovach, Bradley T. Wolfe, Edward Thomas Jr, Hanqi Guo, John E. Foster, and Han-Wei Shen. "Microparticle Cloud Imaging and Tracking for Data-Driven Plasma Science", *Physics of Plasmas*, 27(3): 033703, 2020.

Guan Li, Jiayi Xu, Tianchi Zhang, Guihua Shan, Han-Wei Shen, Ko-Chih Wang, Shihong Liao, and Zhonghua Lu. "Distribution-based Particle Data Reduction for

In-situ Analysis and Visualization of Large-scale N-body Cosmological Simulations", In *Proc. IEEE Pacific Visualization Symposium*, pages 171-180, 2020.

Ko-Chih Wang, Jiayi Xu, Jonathan Woodring, and Han-Wei Shen. "Statistical Super Resolution for Data Analysis and Visualization of Large Scale Cosmological Simulations", In *Proc. IEEE Pacific Visualization Symposium*, pages 303-312, 2019.

Yuxin Ma, Jiayi Xu, Xiangyang Wu, Fei Wang, and Wei Chen. "A Visual Analytical Approach for Transfer Learning in Classification", *Information Sciences*, 390: 54-69, 2017.

Yuxin Ma, Wei Chen, Xiaohong Ma, Jiayi Xu, Xinxin Huang, Ross Maciejewski, and Anthony KH Tung. "EasySVM: A Visual Analysis Approach for Open-Box Support Vector Machines", *Computational Visual Media*, 3(2): 161-175, 2017.

Bing Ni, Qiaomu Shen, Jiayi Xu, and Huamin Qu. "Spatio-temporal Flow Maps for Visualizing Movement and Contact Patterns", *Visual Informatics*, 1(1): 57-64, 2017.

Wenchao Wu, Jiayi Xu, Haipeng Zeng, Yixian Zheng, Huamin Qu, Bing Ni, Mingxuan Yuan, and Lionel M. Ni. "TelCoVis: Visual Exploration of Co-occurrence in Urban Human Mobility Based on Telco Data", *IEEE Transactions on Visualization and Computer Graphics*, 22(1): 935-944, 2016.

Hui Lei, Haidong Chen, Jiayi Xu, Xiangyang Wu, and Wei Chen. "A Survey on Uncertainty Visualization", *Journal of Computer-Aided Design and Computer Graphics*, 25(3): 294-303, 2013.

Yuxin Ma, Jiayi Xu, Dichao Peng, Ting Zhang, Chengzhe Jin, Huamin Qu, Wei Chen, and Qunsheng Peng. "A Visual Analysis Approach for Community Detection of Multi-Context Mobile Social Networks", *Journal of Computer Science and Technology*, 28(5): 797-809, 2013.

Fields of Study

Major Field: Computer Science and Engineering

Studies in:

Computer Graphics	Prof. Han-Wei Shen
High Performance Computing	Prof. Dhabaleswar K. Panda
Artificial Intelligence	Prof. DeLiang Wang

Table of Contents

Page

List of Tables

Table **Page**

List of Figures

Chapter 1: Introduction

1.1 Background and Motivation

Scientists perform experiments and run simulations to analyze real-world phenomenon in many scientific disciplines, such as computational fluid dynamics and plasma sciences. The experiments and simulations can produce high-resolution scientific datasets at the scale of terabytes, petabytes, or even approaching exabytes [144]. It becomes challenging to analyze such large-scale datasets in detail and extract useful information.

Extracting and visualizing features can help scientists derive valuable insights from large-scale scientific data. A range of feature extraction and visualization applications appears in the literature. For example, extracting and tracking microparticles generated from interactions between plasmas and materials in high temperatures can help scientists comprehend plasmas' properties [165]. Extracting and tracking blob-like features in time-varying ion density fields of fusion reactor simulations can help scientists understand heat and particle transport in fusion reactors [87, 119, 141]. Other applications include halo visualization in cosmology [154], flame extraction in combustion sciences [88], and vortex tracking in superconductivity [66, 67, 132], just to name a few. Although many feature extraction and visualization approaches

have been proposed, scientists usually demand specialized solutions to handle their problems more effectively and efficiently given specific new scientific problems.

Feature extraction and visualization pipeline usually includes three steps: (1) scientific feature detection, (2) labeling connected components of features using a union-find algorithm, (3) finalization for further visualization, analysis, and storage. Moreover, as the scale of scientific data generated by experiments and simulations grows, it becomes a common practice to use High-Performance Computing (HPC) clusters and supercomputers to analyze and visualize scientific data in parallel. In such distributed and parallel computing environments, data parallelism is a default paradigm; input data are partitioned into data blocks, which are distributed among parallel processes. With data-parallelism, intermediate results are produced from individual data blocks before they are merged into the final result. Although feature extraction and visualization is a well-studied topic in scientific visualization, most algorithms are either designed for single processors or not scalable enough to handle extreme-scale datasets in the modern era. That leads to research potential for developing new distributed and parallel algorithms for feature extraction and visualization problems.

1.2 Challenges and Problem Statement

We identify three typical problems, which usually arise with respect to different scientific applications.

First, designing robust feature extraction for domain-specific problems is non-trivial in terms of how to model features of interest, accommodate data with varying qualities, and capture spatiotemporal changes of features. Given scientific data, it is not trivial to properly model the features of interest, which are different in domain

applications. The features of interest may have explicit mathematical forms, such as critical points [57, 159] and level sets [51, 72], may be categorized by statistical distributions of data values [47], or most may only be described by soft domain knowledge such as viscous and gravitational fingers [98, 170]. Moreover, a feature detector may be effective in high-quality data but fail in a low-quality counterpart with noise or low resolution in space or time. Furthermore, an existing detection approach may not be effective and robust enough to capture features of interest across all available settings and timesteps because the characteristics of features, such as geometric/topological structures and data statistics, may vary non-linearly.

Second, existing union-find algorithms usually are either serial or not scalable enough to deal with large-scale scientific datasets. In scientific applications, if the large-size data do not fit in a single core, it is common to split the data into blocks and process the data blocks in parallel using distributed-memory machines. Two scalability bottlenecks usually appear for distributed and parallel union-find algorithms: (1) high synchronization costs and (2) imbalanced workloads. (1) Distributed algorithms usually are implemented with the bulk-synchronous parallel programming model [160], which manages parallel processes to alternate local computations and global synchronizations iteratively until distributed algorithms converge. Because each feature is usually distributed in a subgroup of processes, the use of global synchronizations for feature-related operations may block the rest of the processes, causing busy waits in program execution. (2) The imbalanced workloads between parallel processes lead to additional busy waits. The workload imbalance across processes is caused by processes that hold imbalanced features. For example, in super-level set extraction, data blocks in some

processes may find more voxels above the given threshold than others. Likewise, in critical point tracking, critical points may be non-uniformly distributed in a domain.

Third, according to existing studies [131, 134], the scalability and performance of parallel feature extraction and visualization in vector fields are highly dependent on two aspects: (1) the workload balance of parallel processes and (2) the cost of communications. (1) The workload of processes can be imbalanced. Uneven distributions of complex features (e.g., critical points and vortices) in space usually lead to uneven distributions of feature extraction workloads. (2) The cost of parallel interprocess communications can be high due to the exchange of data. For example, the circular flow patterns in input vector field data usually lead to frequent data transfer, causing a high communication overhead. The simultaneous optimization of both workload balance and communication efficiency requires re-distributing data among processes, which can be categorized into the integer programming problem and is NP-complete [56, 58]. Additionally, for flow feature extraction in vector fields, scientists usually seed particles and track those particles in input vector fields. The particle positions continue to change during the parallel execution, leading to volatile information for optimization decisions; hence classic methods such as dynamic programming are ineffective.

1.3 Solutions and Contributions

We present our contributions to answering the abovementioned challenges and problems on scalable extraction and visualization of scientific features.

First, we propose a human-centric feature detection and exploration framework, which leverages a voxel-based ridge detection and an interactive visual-analytics system to guide track and visualize time-varying viscous and gravitational fingers in 3D scalar

fields. Our feature detection offers an additional parameter that scientists can control to fit data with different qualities, making the detection more robust. We provide an interactive visual analytics system that allows effective exploration of fingers over space and time with minimized occlusion. Our system incorporates a novel geometric-glyph augmented tracking graph that reveals the temporal evolution of the fingers and branches. After using our system, the earth scientists recognized the value of our work that provides a new perspective on the analysis of viscous and gravitational fingering, and acknowledged that this could reduce their workload for the analysis of fingers in both space and time significantly. Specifically, the extraction of fingers by our method was effective, and that the minimal occlusion in visualizations led to clear representations of fingers. The visual-analytics system helped scientists acquire branches in space and branching behaviors over time efficiently and effectively, which relieved cumbersome work for the geometric analyses of fingers manually, since the branch tracking offered by our system is new and not directly available by using previous methods [5, 49, 98, 99].

Second, we designed a novel distributed union-find algorithm to make feature extraction and visualization algorithms more scalable. We prove global synchronizations between processes in existing distributed union-find can be eliminated, and present a method that allows distributed union-find to overlap communications and local computations, which reduces processes' busy-waiting time. We redistribute the feature elements across processes evenly for load balancing of distributed union-find using a k-d tree decomposition scheme, which improves algorithm scalability in scientific applications. We demonstrate the scalability of our distributed union-find algorithm by measuring the performance with up to $1,024$ processors for both critical point tracking

and super-level set extraction. Benchmark datasets include synthetic data, high-speed imaging data, and fusion plasm simulation data. We show that our algorithm achieves a shorter execution time and better scalability than the existing distributed union-find methods in the scientific datasets.

Third, we develop reinforcement learning based approaches to adapt decision-making with learning from the dynamic environments in distributed systems and maximize reward functions, where the reward functions are designed to incorporate both workload balance and communication costs. Specifically, we introduce three models that work hand in hand to enable online performance optimization for parallel particle tracing in distributed-memory systems: (1) dynamic work donation model, (2) workload estimation model, and (3) communication cost model. First, we propose an RL-based workload donation model to allow processes to balance their workloads periodically. We associate an RL agent with each process. An agent is trained and used to move work from processes with more workload to those with less workload. Rewards guide the agents' behaviors, and are designed according to the distributions of workloads and cost of data transfer in order to create balanced workloads among processes with minimized costs. Second, we design an online and high-order workload estimate model to estimate blockwise workload in the units of advection integration steps given the incoming particles based on historical data that is recorded during the run time. The model learns the historical data from high-order data access patterns of particles, and dynamically adapts to different flow behaviors. Third, we construct a communication cost model to estimate the data transfer time of both blocks and particles based on the historical data transfer since the beginning of the run, allowing the model to adapt to the available network bandwidth. The cost model

is constructed based on a linear transmission model [30, 85, 158] that models the cost to be a constant latency plus time proportional to the data size. We evaluate our method with applications from fluid dynamics, ocean, and weather. We run our prototype implementation on a supercomputer with up to 16,384 processors. Our method outperforms the state-of-the-art in terms of parallel efficiency, load balance, and the cost of I/O and communications.

1.4 Overview and Outline

The rest of the dissertation is organized as follows.

Chapter 2: We describe the background and present a brief survey of related works for (1) the viscous and gravitational finger in Section 2.1, (2) distributed union-find algorithms in Section 2.2, and (3) parallel particle tracing in Section 2.3.

Chapter 3: We propose a geometry-driven solution to satisfy scientists' requirements and analyze geometric structures of viscous and gravitational fingers. We present preliminaries of our solution in Section 3.1. Following the solution overview in Section 3.2, we detect fingers and branches in Section 3.3 guided by a ridge voxel detection method. In Section 3.4, we visualize fingers and branches as geometric glyphs, and nest the glyphs into a tracking graph so that we can track and compare fingers and branches efficiently and effectively. Case studies are presented in Section 3.5 to demonstrate the efficacy of our solution. We discuss the limitations of the geometry-driven solution in Section 3.6.

Chapter 4: We elucidate our distributed union-find algorithm. We describe algorithm preliminaries in Section 4.1 and an overview in Section 4.2. We detail our distributed asynchronous union-find algorithm in Section 4.3 and our load balancing

scheme in Section 4.4. In Section 4.5, we present results of our experiments with scientific datasets. We discuss algorithm limitations in Section 4.6. An appendix in Section 4.7 presents additional evaluations for our algorithm.

Chapter 5: We provide a reinforcement-learning based approach to optimize the performance of parallel particle tracing. We give an overview in Section 5.1. We detail our RL-based dynamic work donation model in Section 5.2, high-order workload estimation model in Section 5.3, and communication cost model in Section 5.4. In Section 5.5, we present results with studies on vector fields outputted from scientific simulations. Algorithm limitations are discussed in Section 5.6.

Chapter 6: We draw conclusions for our proposed solutions in Section 6.1 and present future plans in Section 6.2.

Chapter 2: Background and Related Work

2.1 Background and Requirements for Tracking and Visualizing Viscous and Gravitational Fingers

In this section, we elaborate on the concepts for the formation of fingers, discuss the domain-specific requirements, and summarize the limitations of previous detection and visualizations of viscous and gravitational fingers.

2.1.1 Viscous and Gravitational Fingers

Viscous and gravitational flow instabilities in porous media result in finger-like features. The *fingering* phenomenon refers to the formation and evolution of such fingers. The fingering instabilities are triggered by adverse mobility or density ratios between displacing and displaced fluids [110]. *Viscous fingering* is caused by viscosity contrasts between fluids: when a less viscous fluid is injected into a more viscous medium, the less viscous fluid tends to penetrate through the more viscous fluid to form elongated finger-like structures [77]. *Gravitational fingering* is caused by contrasts in density between fluids: when a denser fluid resides on top of a less dense fluid, the interface may become unstable. The denser fluid vertically penetrates the lighter fluid to form fingers, while the lighter fluid rises buoyantly.

Data description: The fingering datasets are generated from simulations of injecting carbon dioxide (CO_2) from the top of a water-saturated reservoir. The gravitational fingering helps to mix injected CO_2 throughout the aquifer. When CO_2 dissolves in water at the top, it locally increases the water density, which is prone to gravitational instabilities. The CO_2-waterfront becomes unstable and leads to fingering of CO_2-enriched, denser, water downwards with the upwelling of fresh lighter water. The fingering data at every timestep is volumetric and is constructed using a 3D rectilinear grid ($90 \times 90 \times 100$). The density is defined at the center of every grid cell; in other words, the density data generated by the simulation are cell-centered. Every simulation run produces more than one hundred timesteps.

2.1.2 Scientist Requirements

We discuss the application-specific requirements that were identified by a domain expert who specializes in Earth Sciences. This earth scientist has ten years of experience in researching fluid injection processes and the associated viscous and gravitational fingering instabilities. We generalize three requirements in the following.

2.1.2.1 Requirement One (R1): Identification of Geometric Features

Visualizing and quantifying the geometric features of fingers provide insights into the analyses of bimolecular reaction [42] and physical flow regimes of, e.g., enhanced or suppressed mixing rates [6, 8]. Moreover, the geometrical features, including widths [149] of fingers and locations [149] of fingertips, have been used to analyze scaling behaviors. However, the vast majority of studies [42,149] for geometric analysis of fingers are in 2D and often that has been done essentially by hand and visual inspection.

From our discussion with the earth scientist, two important geometric finger characteristics were identified: branching (**R1.1**) and height (**R1.2**). The earth scientist also explained that these two characteristics have the potential for identifying new scaling laws. Specifically, the branching is critical to scientists in understanding flow instabilities [21, 22, 75, 93]. Due to the gravity, fingers stretch vertically; based on the height, we can estimate the vertical speed of growth of fingers to analyze the stretching process. In this paper, we define *height (persistence) of a finger* as the height difference between the finger root and the fingertip. The *finger root* is the highest part of the finger and, generally, is where the CO_2 is injected. The *fingertip* is the deepest point that the finger can reach in the reservoir, and usually has a low-density value. The location of the finger root and fingertip is illustrated in Fig. 3.4b. Similarly, *height (persistence) of a finger branch* is the height difference between the highest point and deepest point on the branch.

2.1.2.2 Requirement Two (R2): Spatial Exploration

During our interactions, the scientist further mentioned that he was interested in how to visualize the 4D (3D in space plus time) simulations of the fluid injection. In his day to day studies, the expert often visually analyzes the fingers using visualization tools such as ParaView [10], VisIt [35], and Tecplot [1]. However, the expert informed us that the current visualization techniques that he used were not ideal. Tracking and quantifying the 3D geometry of these ramified structures in both space and time is virtually impossible with those methods. For example, some domain tools visualize fingers as hollow sheets rather than dense columns; also, the three-dimensional fingers which were visualized by standard visualization methods such as volume rendering and isosurfaces suffered from the occlusion problem. Thus, the expert usually had

to cut multiple cross-sections of those fingers to analyze the formation and internal structures of fingers rigorously.

The scientist motivated us to develop visualizations to explore fingers in space effectively and efficiently. Specifically, the visualizations were required to address how hundreds of distributed fingers grow vertically (**R2.1**) and spread horizontally (**R2.2**) with minimal occlusion (**R2.3**).

2.1.2.3 Requirement Three (R3): Temporal Exploration

The earth scientist was specifically interested to know how the fingers shield (e.g., one finger shields the other finger from growing further [77]) and merge at certain timesteps and then predominantly split into new smaller fingers at other timesteps geometrically (**R3.1**). The expert thought that interactive space-time diagrams of the fingers, which can effectively present the finger-specific evolution events such as merging, splitting, and branching of fingers, would be extremely valuable.

2.1.3 Limitations of Existing Works for Fingers

In this section, we review the methods used previously to detect and visualize viscous and gravitational fingers, and highlight the limitations of the existing methods.

2.1.3.1 Limitations of Current Detection Methods for Fingers

Since the formation of fingers is extremely non-linear, accurate detection of fingers is a non-trivial task. To extract fingers from the density (or concentration) scalar fields, previous researchers typically used thresholding on the density value to extract high-density regions. Then, they interpreted the connected high-density regions as fingers. In the previous study, Skauge et al. [148] and Fu et al. [52] modeled fingers as

high-density vertical lines, which were detected by a peak detection method. Aldrich et al. [5] and Lukasczyk et al. [99] considered each finger to be a connected component of high-density 3D regions, and detected fingers through the identification of connected components. Favelier et al. [49] detected fingertips first, and then segmented the input volume from the fingertips to isolate the complete fingers by using a watershed traversal method [164]. Luciani et al. [98] studied particle data of fingering, and grouped particles with close locations and concentrations to be fingers.

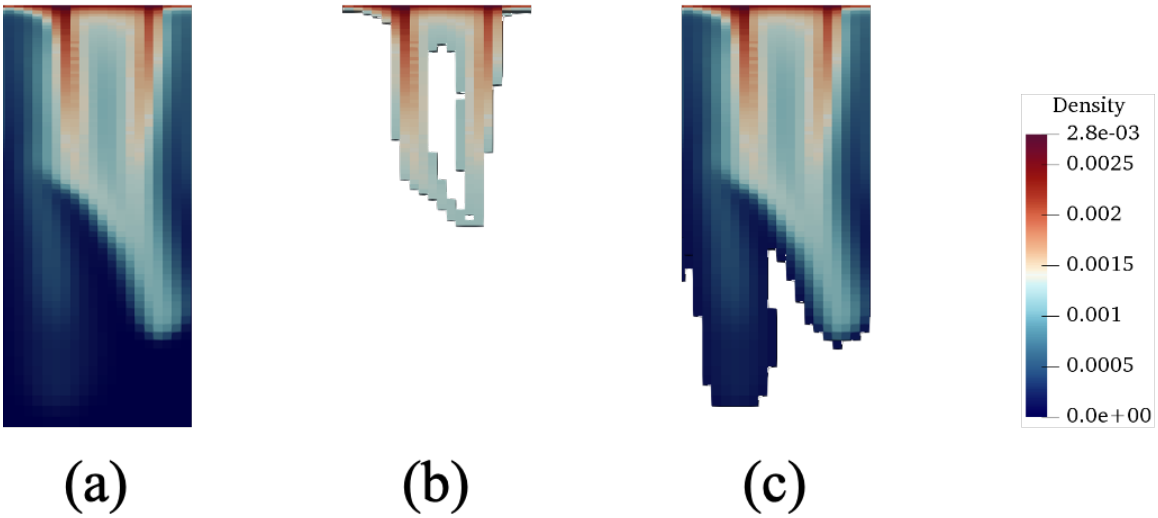


Figure 2.1: (a) Raw volume. (b) Super-level sets with high-density threshold, where low-density branches are missing. (c) Super-level sets with low-density threshold, where high-density branches are not separated.

The previous detection methods [5,49,52,98,99,148] typically detect each finger as a single entity by using density thresholding. Even though the previous methods obtained good results, the density thresholding does not capture detailed geometric structures of fingers, which are essential for earth sciences (R1.1). Specifically, fingertips usually

have much lower densities than the finger roots. As shown in Fig. 2.1, if the used threshold is too high, the fingertips with low density values may not be detected; if the used threshold is too low, different finger branches may not be segmented because they are connected by low-density regions.

2.1.3.2 Limitations of Current Tracking Graphs for Fingers

When it comes to the visualization of time-varying fingers, none of the existing tracking visualizations focus on the geometric evolution of the fingers. The tracking graphs provided by Aldrich et al. [5] display fingers at each timestep as points in a column. Aldrich et al. [5] used hues to encode different fingers and linked related fingers between adjacent timesteps by curves. In recent work, Lukasczyk et al. [100] proposed nested tracking graphs, which can depict the evolution of level-sets of density fields and visualize the evolution of fingers across multiple specified density thresholds. The lack of tracking graphs for the geometric evolution of fingers that encodes the branching information further motivates us to develop a new geometry-driven tracking graph for the analysis of the fingering process.

2.2 Related Work of Distributed Union-Find

We distinguish shared-memory [9, 145] and distributed-memory [37, 51, 72, 80, 102] parallelization of union-find, and this paper is focused on the distributed-memory settings. Shared-memory parallel union-find focuses on computing environments that share the same memory space, such as multi- and many-core processors. Distributed parallel union-find is designed to perform computations in independent processes with distributed memory spaces.

2.2.1 Distributed Union-Find Algorithms

For data visualization and analysis, most distributed union-find algorithms [37, 51, 72, 80, 102] are implemented with the bulk synchronous parallelism [160], which has been the orthodox parallel programming model for distributed iterative algorithms since the 1980s. In the context of union-find, the bulk synchronous implementations alternate two stages: (1) each process performing serial union-find computations locally and (2) all processes synchronized to merge and update disjoint sets across processes; the two-stage iterations continue until convergence. The benefits of using bulk synchronous parallelism include: (1) ensuring disjoint sets are consistent among the processes at each iteration, (2) ensuring messages are exchanged in a predetermined order without causing deadlocks of communications, and (3) offering straightforward termination detection. The drawbacks include disallowing overlapped computations and communications.

Differences of existing distributed union-find algorithms mainly vary in two aspects: (1) data distribution and (2) communication patterns.

Data distribution: There are two ways to distribute data: full replication and data partitioning. Full replication refers to duplicating all set elements among participating processes [37], while data partitioning refers to subdividing the input data and redistributing the partitioned data among the processes. Harrison et al. [72] partitioned mesh data and balanced the mesh cells across the processes using a binary space partitioning approach.

Communication patterns: To merge and update disjoint sets across the processes, existing distributed union-find methods use global synchronizations with three different communication patterns: (1) master/slave, (2) parallel merging, and

(3) neighbor exchange. First, with the master/slave pattern in [51], after all the processes finish local computations, all inter-process mergings of sets are sent to the master process to resolve. Then, the master process broadcasts the sets after the mergings to all processes. Second, with the parallel merging pattern, Cybenko et al. [37] built a tree of processes to merge disjoint sets in paired processes every time by sending all disjoint sets of a process to the process's partner. As a result, the root in the tree of processes produces the disjoint sets merged from all processes. Third, with the neighbor exchange pattern [72, 80, 102], disjoint sets spanning over adjacent processes are merged and updated at every iteration. The neighbor exchange can be implemented under Message Passing Interface (MPI) standards [61] by using either a collective synchronous operation `MPI_Alltoall` involving all processes [72] or point-to-point synchronous operations [80, 102].

2.2.2 Distributed and Parallel Visualization and Analysis

Distributed union-find is mostly used for connected component labeling (CCL) in scientific visualization and analysis algorithms, including distributed percolation analysis [51], interval volume extraction [72], statistical analysis of connected regions [104], contour tree computation [116,126], and merge tree computation [115,123]. Although the implementation of CCL [73] can be either union-find or breadth-first search [40, 41, 121, 122, 146], union-find is proved to be more scalable in distributed and parallel settings [73,80] because union-find organizes elements of connected components using tree structures, explained in Section 4.1.1, which can efficiently synchronize

labels of elements in the connected components across processes. The rest of this section samples typical scientific visualization and analysis applications with distributed union-find.

Region extraction: Visualizing and analyzing spatial regions in scientific data usually offer important insights to scientists, where union-find can play the role of grouping voxels or mesh cells into connected regions. In distributed percolation analysis for turbulent flows [51], the percolation function requires quantifying the volume of regions with values higher than a threshold, where the regions are extracted using distributed union-find. Interval volume, the volume between two isosurfaces, is important for distributed analysis of supernova simulation, and can be extracted using distributed union-find [72]. In the distributed statistical analysis of fluid flows [104], connected phase regions are first extracted using the existing distributed union-find [72, 80]; then, each compute node launches asynchronous threads to compute statistics on the extracted regions.

Critical point tracking: In critical point tracking, union-find is used to connect critical points sharing the same spacetime mesh cells. Tricoche et al. [159] and Garth et al. [57] generalized the spatial mesh of the input data into spacetime mesh, with time being the additional dimension. By assuming continuities in the spacetime domain, critical points can be tracked based on spacetime mesh connectivities.

Scalar field topology: Distributed union-find has been used in the distributed computation of merge tree [115, 123] and contour tree [116, 126]. Given scalar values defined on mesh vertices, union-find is used to group vertices on the mesh paths where vertices' values on the paths are increasing.

2.3 Related Work of Parallel Particle Tracing

We summarize related works on parallel particle tracing in distributed-memory systems. In general, parallel particle tracing can be categorized into distributed-memory [17, 27, 33, 34, 64, 68, 84, 97, 117, 125, 131, 134, 152, 177, 178, 180] and shared-memory [24, 26, 89, 90, 136, 143] settings, where our paper is focused on the distributed particle tracing; the former focuses on computations in independent processes with separate memory spaces, and the latter is done in computing environments that share the same memory space, including many- and multi-core processors. In distributed-memory settings, two strategies exist, including data-parallel and task-parallel. We refer to literature [135, 181] for a comprehensive review of parallel particle tracing.

2.3.1 Load Balancing for Distributed Particle Tracing

There are two basic load balancing strategies: (1) static and (2) dynamic load balancing, where this paper studies the dynamic one.

Static load balancing: The data partition is predetermined and optimized before parallel particle tracing is performed. Peterka et al. [131] used a static round-robin strategy to assign data blocks to parallel processes to balance workload. Alternatively, Nouanesengsy et al. [125] built a matrix-based optimization model to assign blocks to processes for the workload balancing.

Dynamic load balancing: Workloads of processes are periodically optimized. Existing dynamic load balancing algorithms have three categories: (1) domain (re-)partitioning, (2) master/slave, and (3) work stealing/requesting. First, Peterka et al. [131] applied the recursive coordinate bisection (RCB) [15] to partition data dynamically to make each process have a similar estimated workload. Zhang et

al. [178,180] improved the RCB based method using a constrained k-d tree [178] and a workload prediction model [180]. Second, Pugmire et al. [134] proposed a master/slave based algorithm, where master processes dynamically move particles to idle slaves when slave processes have no work to do. Third, Müller et al. [117] and Lu et al. [97] applied work stealing/requesting scheme [18, 44] for distributed and parallel particle tracing, where idle processes repeatedly steal particles from random processes to reduce the total idle time. Binyahib et al. [17] applied the lifeline technique [142] to connect processes with a one-bit difference in process ranks so that particles can be redistributed among connected processes after random stealing fails.

2.3.2 Workload Estimation

Workload estimation is usually used to assist balancing workloads among processes. Existing estimation strategies have two categories: (1) blockwise workload estimation and (2) particle-wise workload estimation, where this paper studies the blockwise workload estimation in Section 5.3.

Blockwise workload estimation: One may estimate each data block's workload and assign data blocks to processes to have a balanced estimated workload. Nouanesengsy et al. [125] advected a set of uniformly seeded particles in a preprocessing stage to determine the flow characteristics of each vector-field data block. Peterka et al. [131] used historical number of advection steps per particle within a data block to estimate the workload of incoming particles of that block as the future workload of that block.

Particle-wise workload estimation: One may estimate each particle's workload from the current time to the tracing termination and assign particles with a similar

estimated workload to each process. One may assume each particle requires similar computation time and balance workloads of processes by balancing particle counts [97, 117, 134]. Zhang et al. [180] estimated the workload of each particle from the particle exits the current accessed data block to advection completion, with the construction of access dependency graphs (or, flow graphs) [31, 32, 173].

2.3.3 I/O for Distributed Particle Tracing

I/O for data block loading is a common bottleneck for the scalability and performance of distributed and parallel particle tracing methods. Two types of techniques are proposed to reduce the I/O cost for loading and fetching data blocks in parallel particle tracing.

On-demand data loading: Data blocks are loaded from disks as long as no more work can be completed based on the blocks in memory. Pugmire et al. [134] proposed the on-demand data loading strategy, which is followed by various parallel particle tracing studies [26, 97].

Prefetching: One can prefetch data blocks at each disk access with knowledge of data access dependencies to reduce the overall I/O cost. Chen et al. [31] constructed an access dependency graph [32] to organize data blocks with strong access dependencies closely in disk file layout to reduce disk seek time, and proposed an out-of-core method to prefetch consecutive data blocks with access dependencies for efficient pathline computation. The work of Guo et al. [69] builds hint graphs to record the data access dependencies between fine-grained blocks and prefetches data blocks with dependant access patterns into a parallel key-value store to reduce the latency of data access for unsteady flow visualizations. Zhang et al. [179] studied Markov-chain based high-order

access dependencies in unsteady flow fields, enabling high-order data prefetching for the computation of pathlines and acquiring improved prefetching accuracy. Hong et al. [78] studied predicting data accesses using Long Short-Term Memory (LSTM) models for data block prefetching.

2.3.4 Flow Visualization and Analysis

Distributed and parallel particle tracing methods can be used to perform texture-based [91] and geometry-based [105] flow visualizations, as well as for flow analyses with feature extraction and tracking [133] in distributed-memory systems. The distributed visualization and analysis studies can be categorized into two groups: local-range and full-range according to particle seed distribution.

In local-range flow analyses, particles are seeded sparsely in local regions, focusing on analyzing the data's local phenomenon. For example, in distributed source-destination queries [69, 84, 178], particles are seeded within a local region and are traced in distributed memory for the identification of destinations. In distributed streamsurface computation [97], particles are seeded on a curve for the analyses of 3D flows.

Full-range analyses, where particles are seeded densely in the entire spatial domain, are used to analyze global features in the vector fields. For example, in texture-based visualization, distributed LIC [23, 118] and FTLEs [70, 124, 178] require distributed particle advection for the computation of streamlines and pathlines, respectively, originating from all positions. The large-scale analyses of Lagrangian coherent structures (LCSs) and FTLEs [70] require scalable distributed and parallel particle tracing [124],

even though techniques, including adaptive refinement [12] and partial path reuse [76], have been used to improve the tracing computation of densely-seeded particles.

Chapter 3: Geometry-Driven Detection, Tracking and Visual Analysis of Viscous and Gravitational Fingers

In the context of fluid flow in subsurface porous media (e.g., rock formations), *fingering* refers to flow instabilities when either an invading fluid has a much lower viscosity than the displaced fluid (i.e., viscous fingering), or when a denser fluid flows on top of a lighter fluid (i.e., gravitational fingering). Fingering instabilities lead to a highly non-linear and complex evolution of the displacement front between different fluids [7, 110]. Understanding and tracking flow instabilities is critical in a variety of scientific fields including fluid mechanics [6, 77], computational fluid dynamics (CFD) [98], and hydrogeology [110, 148]. Fingering is generally detrimental when the objective is to displace a viscous fluid (e.g., oil recovery through waterflooding) but can be beneficial when the aim is to mix two fluids, for instance, in the sequestration of carbon dioxide (CO_2) [8, 149] in deep water-saturated formations.

In this work, we focus on the latter application, in which gravitational fingering helps to mix dissolved CO_2 throughout the aquifer. This, in turn, helps to guarantee the storage permanence of CO_2 [7, 150]. We illustrate the challenges and high-level motivations for detection and visualization of the fingering process from two perspectives: (1) domain-specific requirements obtained from an expert, and (2)

limitations of previous evolutionary analyses for viscous and gravitational fingers. Below we expand on each of these.

We involved an expert in Earth Sciences to help us comprehend the domain-specific requirements and challenges of this work. Flow instabilities, whether in the subsurface or space, have recently been found to obey specific universal scaling laws (e.g., [7, 150]); such scaling laws can be used to estimate the severity and evolution of flow instabilities for different sets of conditions, without having to redo high-resolution and thus computationally expensive simulations. To reveal those scaling laws, the geometric analysis of finger formations is critical, i.e., the connectivity between fingers in space and the onset, growth, merging, and splitting of fingers over time. However, it is difficult for the earth scientist to use standard visualization tools to track and quantify these 3D features.

Detecting fingers is challenging because fingers are unstable structures in the fluids, and result from complex fluid interactions. Different detection techniques have been proposed [5, 49, 52, 98, 99, 148]. The existing detection techniques mostly use a density (or concentration) thresholding based method to detect the complete volume of fingers. However, the features detected by such density thresholding capture limited information on the internal geometric structures of fingers.

In this study, we propose a geometry-driven solution to satisfy the requirements of scientists and analyze geometric structures of fingers. Guided by a ridge voxel detection method, we first extract finger cores from the data of 3D density fields. Based on the extracted finger cores, we obtain the complete volume of fingers, and produce finger skeletons to acquire the overall geometric features of fingers in space. We then propose a spanning tree based algorithm to construct finger branches from

finger skeletons. We visualize fingers and their branches as geometric glyphs, and nest the glyphs into a tracking graph so that we can track and compare fingers and branches efficiently and effectively. Hence, our contributions are twofold:

1. We propose voxel-based ridge detection, which is inspired by Steger's pixel-based method [151], to guide the extraction of finger cores on 3D scalar fields. Furthermore, we provide a spanning tree based heuristic algorithm for the construction of finger branches from finger skeletons.

2. We offer an interactive visual analytics system that allows efficient and effective exploration of fingers over space and time with minimized occlusion. Our system incorporates a novel geometric-glyph augmented tracking graph that reveals the temporal evolution of the fingers and their branches.

This work has been published in IEEE Transactions on Visualization and Computer Graphics [170] in 2020.

3.1 Preliminaries: Ridges and Reeb Graphs

We offer the background of two concepts: ridge and reeb graph, which are the bases of the geometry-driven approaches applied in this paper.

3.1.1 Ridges

3.1.1.1 Ridge Definition

Ridges originally are structures of surface topography whose mathematical properties were studied by de Saint-Venant [43]. The concept of k-dimensional ridges is generalized by Haralick [71], and re-formulated by Lindeberg [95] and Eberly [48].

Intuitively, the ridges of a smooth function are a set of curves or surfaces whose points are local maxima of the function in certain dimensions.

A formal description of ridges is given in the following. Given a scalar field f: $\mathbb{R}^n \rightarrow \mathbb{R}$, we define $\vec{v_1}, ..., \vec{v_n}$ as the eigenvectors of the Hessian matrix $H = [\frac{\partial^2 f}{\partial x_i \partial x_j}]$, where x_i and x_j represent any two axes of the coordinate system. The n eigenvectors are ordered based on their corresponding eigenvalues $\lambda_1, ..., \lambda_n$. Two alternative ways were proposed to order the eigenvalues. Lindeberg [95] ordered the eigenvalues by $|\lambda_1| \geq ... \geq |\lambda_n|$, and Eberly [48] ordered the eigenvalues by $\lambda_1 \leq ... \leq \lambda_n$; the results presented in this paper are generated by using the Lindeberg's ordering [95]. The *k-dimensional ridge* is defined as a set of points where the following two conditions are satisfied: given any point $\boldsymbol{p}$ in the set,

Condition One for extreme point determination: Intuitively, $\boldsymbol{p}$ is a local extreme point along the directions of the first $n-k$ eigenvectors [48]. Mathematically, the first-order directional derivatives at $\boldsymbol{p}$ along the first $n-k$ eigenvectors are all zeros,

$$D_{\vec{v_1}} f(\boldsymbol{p}) = ... = D_{\vec{v_{n-k}}} f(\boldsymbol{p}) = 0 \tag{3.1}$$

Condition Two for feature type differentiation: This condition differentiates ridge points from other types of extreme points. Intuitively, the scalar value at $\boldsymbol{p}$ is maximal along the first $n-k$ eigenvectors. Mathematically, the second-order directional derivatives at $\boldsymbol{p}$ along the first $n-k$ eigenvectors are all smaller than zero,

$$D^2_{\vec{v_1}} f(\boldsymbol{p}), ..., D^2_{\vec{v_{n-k}}} f(\boldsymbol{p}) < 0 \tag{3.2}$$

Since $\lambda_i = \nabla^2_{\vec{v_i}} f(\boldsymbol{p})$, we also have

$$\lambda_1, ..., \lambda_{n-k} < 0 \tag{3.3}$$

In this paper, as the space is $\mathbb{R}^3$, *ridge voxels* are defined as voxels that contain 1-dimensional ridge points, and are where the ridge lines pass through.

3.1.1.2 Ridge Detection

The concept of ridges has been used to detect and extract curvilinear structures in computer vision [95, 151] and scientific visualization [53, 129]. In computer vision, one can regard a greyscale image as a 2D scalar field, and extract curvilinear structures such as roads [151] and human fingers [95] based on ridge detection. In scientific visualization, definitions of ridges are extended to 3D scalar-field data [53, 129]. Although the whole ridge structure is complex, ridge points are local features; namely, they are the features that satisfy the local criterion [48, 95] consisting of the two conditions mentioned above. So, the locally approximated function and derivatives can detect the ridge points and approximate the entire ridge structures well [53, 129, 151]. When the scalar value is defined at every grid point, the marching ridge technique [53] estimates the derivatives at grid points of a given grid cell; then, the derivatives at the grid points are used to interpolate the ridge lines within the grid cell by using zero-crossing tri-linear interpolation.

As a scalar value is defined at the center of every grid cell, such as the pixel-based image in $\mathbb{R}^2$, Steger [151] constructed a Taylor polynomial for every pixel to detect the pixels that contain ridge points. In our application, we generalize the Steger's method from $\mathbb{R}^2$ to $\mathbb{R}^3$ to detect 3-dimensional grid cells that contain 1-dimensional ridge points, and explain the challenge for the ridge voxel detection in $\mathbb{R}^3$ as follows.

Given a grid cell, we estimate the positions of the ridge points locally, and determine whether the position of any local ridge point is in or on the boundary of this grid cell. Since the function of the scalar field, f, is unknown and only discrete scalar values at cell centers are observed, it is hard to acquire the positions of the ridge points directly. To remedy this, Steger [151] approximated $f(\boldsymbol{p})$ locally by using the second-order Taylor polynomial $g(\boldsymbol{p})$ centered at every grid cell. According to Taylor's theorem, the Taylor polynomials can approximate the scalar function with low error, and it is efficient to compute the derivatives from the Taylor polynomials. Hence, the Taylor polynomials are useful for locating the ridge points. The second-order Taylor polynomial $g(\boldsymbol{p})$ is defined by

$$g(\boldsymbol{p}) = f(\boldsymbol{p_0}) + \nabla f(\boldsymbol{p_0})^\intercal(\boldsymbol{p} - \boldsymbol{p_0}) + \frac{1}{2}(\boldsymbol{p} - \boldsymbol{p_0})^\intercal H(\boldsymbol{p_0}) \cdot (\boldsymbol{p} - \boldsymbol{p_0}) \tag{3.4}$$

where $\boldsymbol{p_0}$ with coordinates $(x_{1,0}, x_{2,0}, x_{3,0})^\intercal$ is the center point of the given grid cell. $\nabla f(\boldsymbol{p_0}) = (\frac{\partial f(\boldsymbol{p_0})}{\partial x_1}, \frac{\partial f(\boldsymbol{p_0})}{\partial x_2}, \frac{\partial f(\boldsymbol{p_0})}{\partial x_3})^\intercal$ is the estimated gradient vector at $\boldsymbol{p_0}$ by using the central difference. Without loss of generality, we give the central difference formula for $\frac{\partial f(\boldsymbol{p_0})}{\partial x_1}$:

$$\frac{\partial f(\boldsymbol{p_0})}{\partial x_1} \approx \frac{f(x_{1,0} + s, x_{2,0}, x_{3,0}) - f(x_{1,0} - s, x_{2,0}, x_{3,0})}{2s}, \tag{3.5}$$

where s is the the side length of the voxel. $H(\boldsymbol{p_0}) = [\frac{\partial^2 f(\boldsymbol{p_0})}{\partial x_i \partial x_j}]$ is the estimated Hessian matrix at $\boldsymbol{p_0}$ by using the central difference. Without loss of generality, we give the central difference formulas for $\frac{\partial^2 f(\boldsymbol{p_0})}{\partial x_1 \partial x_1}$ and $\frac{\partial^2 f(\boldsymbol{p_0})}{\partial x_1 \partial x_2}$, respectively:

$$\frac{\partial^2 f(\boldsymbol{p_0})}{\partial x_1 \partial x_1} \approx \frac{1}{s^2}(f(x_{1,0} + s, x_{2,0}, x_{3,0}) - 2f(x_{1,0}, x_{2,0}, x_{3,0}) + f(x_{1,0} - s, x_{2,0}, x_{3,0})). \tag{3.6}$$

$$\begin{aligned}\frac{\partial^2 f(\boldsymbol{p_0})}{\partial x_1 \partial x_2} \approx \frac{1}{4s^2}(&f(x_{1,0} + s, x_{2,0} + s, x_{3,0}) - f(x_{1,0} + s, x_{2,0} - s, x_{3,0}) \\ &- f(x_{1,0} - s, x_{2,0} + s, x_{3,0}) + f(x_{1,0} - s, x_{2,0} - s, x_{3,0}))\end{aligned} \tag{3.7}$$

Deriving from $g(\boldsymbol{p})$ in Equation 3.4, we approximate the gradient vector at $\boldsymbol{p}$ by

$$\begin{aligned} \nabla f(\boldsymbol{p}) &\approx \nabla g(\boldsymbol{p}) \\ &= (\frac{\partial g(\boldsymbol{p})}{\partial x_1}, \frac{\partial g(\boldsymbol{p})}{\partial x_2}, \frac{\partial g(\boldsymbol{p})}{\partial x_3})^\intercal \\ &= \nabla f(\boldsymbol{p_0}) + H(\boldsymbol{p_0}) \cdot (\boldsymbol{p} - \boldsymbol{p_0}) \end{aligned} \tag{3.8}$$

The first-order directional derivatives are approximated by

$$\begin{aligned} D_{\vec{v_i}} f(\boldsymbol{p}) &\approx D_{\vec{v_i}} g(\boldsymbol{p}) \\ &= \vec{v_i}^\intercal \cdot \nabla g(\boldsymbol{p}) \\ &= \vec{v_i}^\intercal \cdot [\nabla f(\boldsymbol{p_0}) + H(\boldsymbol{p_0}) \cdot (\boldsymbol{p} - \boldsymbol{p_0})] \end{aligned} \tag{3.9}$$

Also, the Hessian matrix is estimated by

$$H(\boldsymbol{p}) \approx [\frac{\partial^2 g(\boldsymbol{p})}{\partial x_i \partial x_j}] = H(\boldsymbol{p_0}) \tag{3.10}$$

To locate local ridge points efficiently, Steger's method [151] has one additional assumption: the local ridge points should lie on the line that passes the center $\boldsymbol{p_0}$ of the grid cell and is at the direction of $\vec{v_1}$. Based on this assumption, whether any point $\boldsymbol{p}$ on this line satisfies Condition One, i.e., the extreme point determination, can be efficiently identified first. Second, for Condition Two (i.e., the feature type differentiation), one examines whether the first two eigenvalues of $H(\boldsymbol{p})$ are all smaller than zero or not to guarantee the extreme point $\boldsymbol{p}$ is a ridge point. Third, if the position of any identified local ridge point is within or on the boundary of the grid cell, the grid cell is claimed to contain ridge points. Although the abovementioned assumption gets success in $\mathbb{R}^2$, a challenge is that this assumption brings more restriction to the detection of ridge points within grid cells in $\mathbb{R}^3$ and may lead to missing ridge voxels. Hence, in order to solve this challenge, our approach derives inequalities solely from the basic ridge criterion [48, 95] to detect ridge voxels without additional assumptions for the locations of ridge points.

3.1.2 Reeb Graphs

The *Reeb graph* obtains the topology of a compact manifold by following the evolution of the level-sets of a scalar function defined on the manifold. Nodes in the Reeb graph represent critical points of the scalar function, and edges correspond to connections between critical points [16, 36].

Skeletonization: The Reeb graph based skeletonization is one of the standard skeleton-extraction approaches [36]. When we consider the height function of an object, the corresponding Reeb graph can capture topological features of the object. The corresponding Reeb graph is not a skeleton; however, Lazarus and Verroust [92] embedded the Reeb graph into the space to get the skeleton of the original object which also can indicate the height of the object. If an object has no holes inside, the contour tree [157, 161], as a special instance of the Reeb graph, is sufficient for the skeletonization of the object.

Trim: To trim Reeb graphs, Bauer et al. [13] suggested removing features with persistence smaller than a threshold. Carr et al. [29] suggested pruning away small leaves. In addition to short leaves, Doraiswamy and Natarajan [45] also suggested removing short cycles in Reeb graphs.

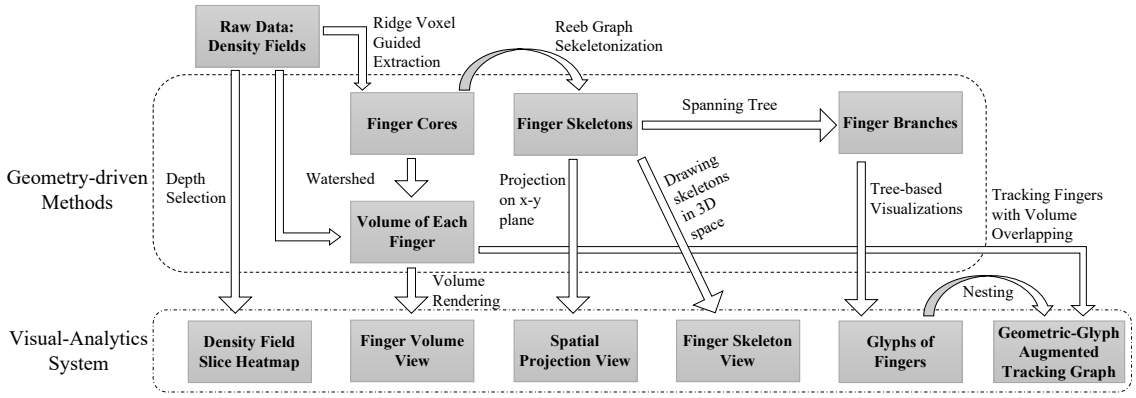


Figure 3.1: We present a schematic diagram of our geometry-driven approaches for viscous and gravitational fingering.

3.2 Approach Overview

In this work, we focus on the detection of geometric structures of fingers (R1) and the spatio-temporal visualizations of the fingers with minimized occlusion (R2 and R3). Fig. 3.1, a schematic diagram, presents the pipeline of our solution. From the density fields, we apply a ridge voxel guided detection method to extract finger cores (Sect. 3.3.1). We expand the finger cores to obtain the complete volume of fingers (Sect. 3.3.3.1), and display the volume of fingers in the spatial domain by using volume rendering (Sect. 3.4.1.3). We skeletonize the finger cores into finger skeletons by using a Reeb graph based skeletonization (Sect. 3.3.2.1), and draw the finger skeletons in 3D space to provide the overall geometric structures of fingers (Sect. 3.4.1.3). Also, we project finger skeletons onto a plane to observe spatially relative positions between fingers (Sect. 3.4.1.2). We construct finger branches from the finger skeletons by using a variant of spanning tree method (Sect. 3.3.2.2). Given the branches of fingers, we adopt various tree-based visualization methods to represent different aspects of finger

branches (Sect. 3.4.2.1 and 3.4.2.2). Finally, we develop a geometric-glyph augmented tacking graph to study how the fingers evolve geometrically (Sect. 3.4.2). On the tacking graph, we link temporally related fingers (Sect. 3.4.2.3) and minimize link crossings (Sect. 3.4.2.4). We also design interactions to identify the change of branches over time (Sect. 3.4.2.5).

3.3 Geometry-Driven Detection of Viscous and Gravitational Fingering

In this section, we present the geometry-driven detection technique for the viscous and gravitational fingering process in detail. Since the fingering process is complex and non-linear, precise descriptors for the fingers are unavailable in the earth science domain. In order to develop a reliable technique for extracting fingers, we model the central regions of the fingers as ridges based on the diffusive process of fingers. In our work, the finger cores include both the ridge regions and the volume near the ridges to capture the connections among neighboring finger branches robustly. The complete finger volume consists of the finger cores and the connected lower-density regions covering the finger cores. To capture the geometric features of fingers (R1), we skeletonize finger cores and construct finger branches.

3.3.1 Ridge Voxel Guided Extraction of Finger Cores

The extraction of finger cores is guided by a ridge voxel detection method. According to our domain scientist, one of the causes of fingering is the diffusion process. The diffusion of CO_2 is driven by compositional gradients (i.e., the difference of CO_2/water composition). Through the diffusion process, CO_2 volume tends to spread out from high-density regions to low-density regions; the regions with higher densities usually

form the center of finger structures. In this context, it is important to note that the ridges are regions with locally higher densities in the density fields. Therefore, the extraction of finger cores can be guided by the ridge detection technique, as discussed below.

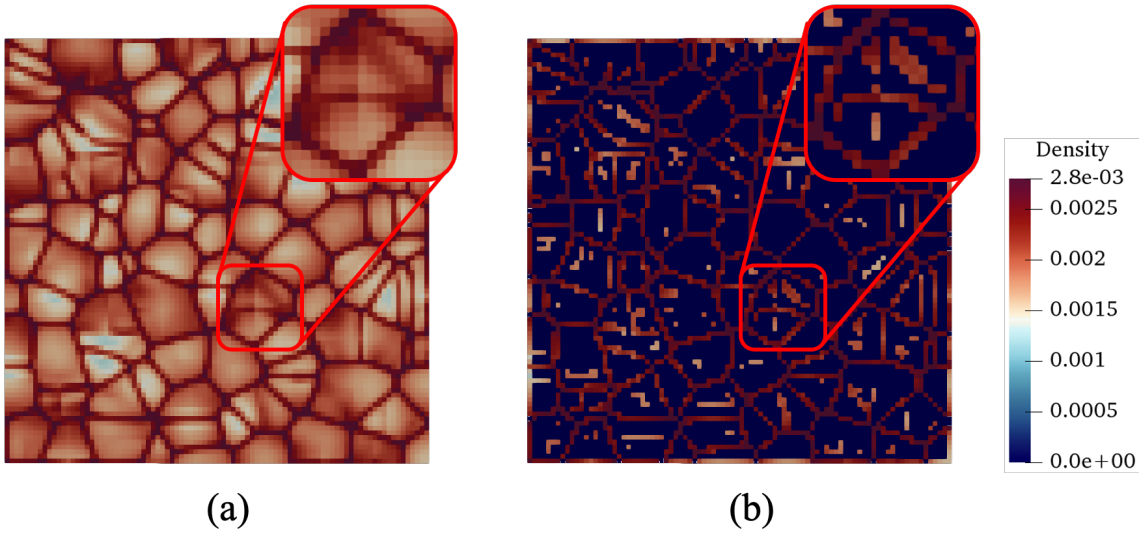


Figure 3.2: (a) We show the top view of the 3D density field. (b) We show the top view of the extracted ridge voxels by setting the density values of other voxels to be zero. As the same places are highlighted with a red box in (a) and (b), the ridge voxels capture high-density hexagonal cells in the top grid cells and high-density curvilinear structures inside the hexagonal cells.

3.3.1.1 Ridge Voxel Detection

To obtain regions with locally higher densities, we filter the 3D density field to identify ridge voxels following the generalized ridge detection framework explained in Sect. 3.1.1.2. The critical part of the detection framework is to robustly detect the ridge voxels, i.e., how to examine whether a given voxel contains any local ridge point

without any additional assumptions for the locations of ridge points. In the following, we provide a solution to this.

Intuitively, given a voxel, we first estimate the locations of extreme points surrounding the voxel. Then, we examine whether any extreme point is located inside or on the boundary of the voxel. Finally, we determine whether any extreme point contained by the voxel is a ridge point to conclude that the voxel is a ridge voxel. Note that Condition One and Two of the ridge criteria used below are explained in Sect. 3.1.1.1.

We first estimate the positions of extreme points in the following. We estimate the density function $f(\boldsymbol{p})$ locally by using the second-order Taylor polynomial $g(\boldsymbol{p})$ in Equation 3.4; the gradients and the Hessian matrix at the center of the voxel in Equation 3.4 are approximated using the central difference formulas in Equation 3.5, 3.6, and 3.7. Derived from Equation 3.4, the estimate of Hessian matrix $H(\boldsymbol{p})$ is given in Equation 3.10. The estimate of $H(\boldsymbol{p})$ is decomposed to obtain eigenvalues λ_1,λ_2, and λ_3 with corresponding eigenvectors $\vec{v_1}$,$\vec{v_2}$, and $\vec{v_3}$, where the eigenvalues are ordered as $|\lambda_1| \geq |\lambda_2| \geq |\lambda_3|$ following the Lindeberg's ordering rule [95]. Given the eigenvectors, we obtain the two first-order directional derivatives, $D_{\vec{v_1}}g(\boldsymbol{p})$ and $D_{\vec{v_2}}g(\boldsymbol{p})$, through Equation 3.9. We equate the two first-order directional derivatives to zeros according to Condition One (i.e., the extreme point determination) of the ridge criteria to obtain:

$$D_{\vec{v_i}}g(\boldsymbol{p}) = \vec{v_i}^{\mathsf{T}} \cdot [\nabla f(\boldsymbol{p_0}) + H(\boldsymbol{p_0}) \cdot (\boldsymbol{p} - \boldsymbol{p_0})] = 0, i = 1, 2 \tag{3.11}$$

where $p_0(x_{1,0}, x_{2,0}, x_{3,0})$ is the center of the given voxel and $p(x_1, x_2, x_3)$ represents estimated extreme point. Equation 3.11 constrains the coordinates of estimated extreme points, and defines the region formed by the extreme points in space.

We examine whether any extreme point is contained by the given voxel through computing the intersection between the region formed by extreme points and the region covered by the voxel; if the intersection is not empty, the voxel contains extreme points. We define the region covered by the voxel below:

$$x_{j,0} - \frac{s}{2} \leq x_j \leq x_{j,0} + \frac{s}{2},\ j = 1, 2, 3 \tag{3.12}$$

where s is the length of the side of the voxel; any point $p(x_1, x_2, x_3)$ within or on the boundary of the voxel should satisfy Equation 3.12. We compute the intersection between the region covered by the voxel and the region formed by extreme points through plugging Equation 3.11 into Equation 3.12 as follows. Note that, Equation 3.11 gives two polynomial equations when i is replaced by 1 and 2 respectively, and has three unknowns (x_1, x_2, and x_3 of $\boldsymbol{p}$), hence, we can represent two of the unknowns by polynomials of the third unknown; without loss of generality, we represent x_2 and x_3 by polynomials of x_1 through transforming Equation 3.11. Thus, we can substitute x_2 and x_3 with the polynomials of x_1 for the three inequalities in Equation 3.12, which produces the intersection region represented by three inequities that are only associated with x_1. If the union of the three inequities of x_1 is not empty, we claim that the voxel contains extreme points.

We examine whether any extreme point $\boldsymbol{p}$ contained by the voxel passes Condition Two, i.e., the feature type differentiation. If the first two eigenvalues of $H(\boldsymbol{p})$, estimated by Equation 3.10, are all smaller than zero, we declare $\boldsymbol{p}$ is a ridge point and this voxel is a ridge voxel.

We demonstrate the results of the ridge voxel detection in Fig. 3.2 and Fig. 3.3. Fig. 3.2 shows the top view of the extracted ridge voxels, and the "$r = 1$" column

of Fig. 3.3 displays the side view of the ridge voxels. The ridge voxels of several individual fingers are shown in Fig. 3.4a, Fig. 3.5a, and Fig. 3.6a.

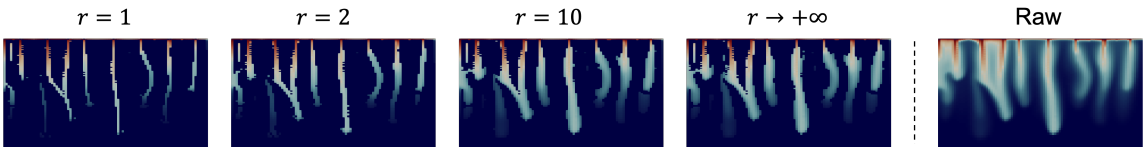


Figure 3.3: We display, on the left side, the sensitivity of the extracted finger cores about different r values by comparing with, on the right side, the corresponding raw density field. We observe that the finger cores generally become more connected and thicker as increasing r. The extracted features become little changed when r is larger than 10. When r becomes large enough, the voxels, whose local information cannot imply any outside ridge structures, are filtered out.

3.3.1.2 Acquisition of Additional Branch Connections

In addition to ridge voxels, we include voxels that are close to the ridges into the finger cores to obtain more connections between branches (R1.1). According to the study of Damon [39], ridges do not preserve the connections between branches well enough because ridge lines can only cross at critical points. Even though the extracted ridge voxels group ridge lines when they are closer than the size of a voxel, certain finger branches are still disconnected. For instance, the fingers in Fig. 3.4a and Fig. 3.5a have disconnected branches, although the connections of branches in Fig. 3.6a are well-preserved. To remedy this, we acquire additional necessary branch connections by including the voxels that have ridge points nearby.

We define the nearby region of a given voxel by a cube and examine whether that cube contains any ridge point. Specifically, given the center $p_0(x_{1,0}, x_{2,0}, x_{3,0})$ of a voxel, we create a cube centered at $\boldsymbol{p_0}$ with side length $r \cdot s$ where $r \geq 1$ and s is the

side length of voxel. We define the region covered by the cube:

$$x_{j,0} - \frac{r \cdot s}{2} \leq x_j \leq x_{j,0} + \frac{r \cdot s}{2},\ j = 1, 2, 3 \tag{3.13}$$

The cube fully covers the given voxel in space, and, intuitively, if r is large, the cube contains more surrounding regions of the voxel. We examine whether the cube contains any ridge points by the same method in Sect. 3.3.1.1 but replacing Equation 3.12 with Equation 3.13. If the cube centered at the voxel contains any ridge point, we include the given voxel into finger cores.

Scientists can control how many additional voxels are needed for the preservation of branch connections through visual inspection, as the images are shown in Fig. 3.3. In this application, we keep increasing r until the results remain unchanged to maximize the acquired branch connections. The produced results are demonstrated in Fig. 3.4b, Fig. 3.5b, and Fig. 3.6b respectively. The limitation of using r is that it makes the branch connections sensitive to the additional parameter, r, which is demonstrated in Fig. 3.3. However, a benefit is that by controlling the value of r, scientists can flexibly study other similar datasets without changing the finger core detection algorithm.

3.3.2 Construction of Geometric Structures of Fingers

In this section, we construct geometric structures of fingers (R1). We first obtain finger skeletons from finger cores. Finger branches are constructed from finger skeletons then. We further trim finger skeletons by removing short branches and cycles.

3.3.2.1 Reeb Graph Based Skeletonization

We use the Reeb graph based method [16, 36, 62, 92, 99] to obtain finger skeletons. In computational geometry, Reeb graphs are well-known for their ability to show

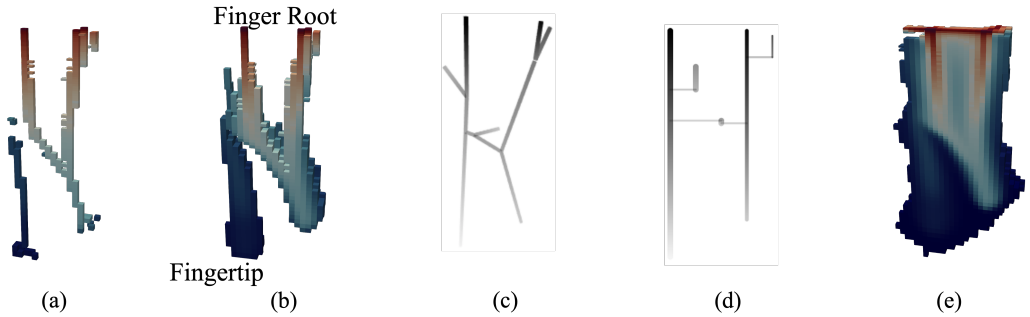


Figure 3.4: We display the results of our methods for a simple finger. We show five images: (a) the ridge voxels, (b) the finger core whose finger root and fingertip are indicated, (c) the finger skeleton, (d) the linear glyph of the finger, and (e) the complete finger volume.

the skeletons of a 3D object [36] and preserve the branching and the height of a 3D object accurately (R1). Contour tree [161] is an alternative approach for shape skeletonization. However, in our application, voxels that do not belong to finger cores are removed, which may leave holes in the volume of finger cores (e.g., Fig. 3.6b) and result in torus-like structures. Hence, the contour tree based method is not appropriate for this application, although it is more efficient than the Reeb graph based method.

The skeleton of a simple finger is shown in Fig. 3.4c. Complex examples are shown in (e) of both Fig. 3.5 and Fig. 3.6.

3.3.2.2 Spanning Tree Based Extraction of Finger Branches

We extract finger branches from finger skeletons by a spanning tree based heuristic algorithm. We create finger branches explicitly for two reasons. First, we need to identify branches according to R1.1. Second, finger branches, as tree structures, can be visualized with minimized occlusion (R2.3). Due to the stretching process, finger branches grow downward; hence, branches are assumed to be vertical linear

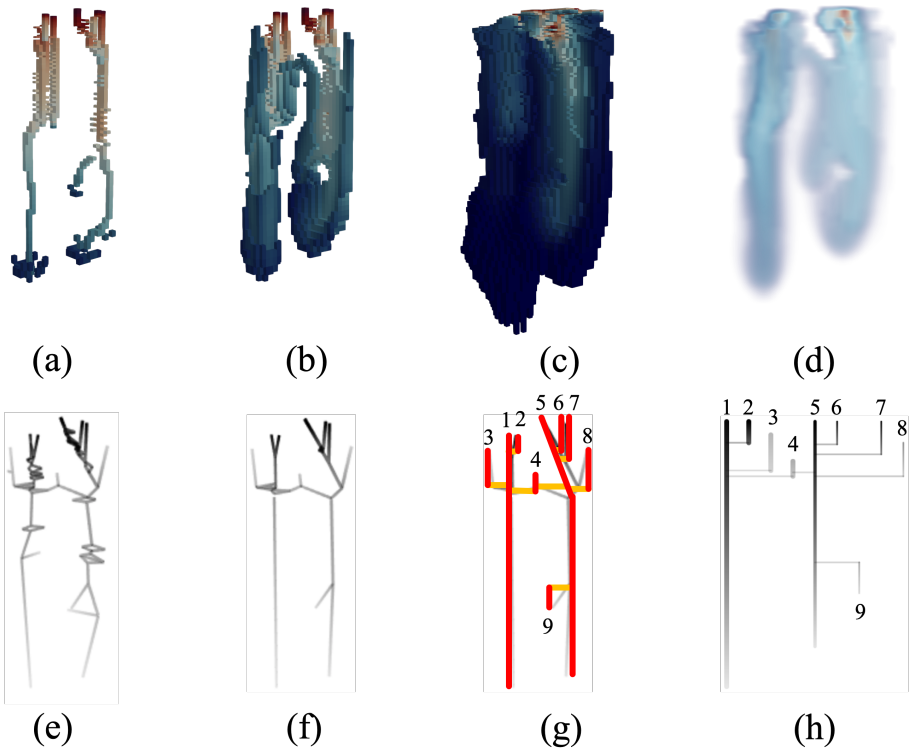


Figure 3.5: We display a complex finger with (a) the ridge voxels, (b) the finger core, (c) the complete finger volume, (d) the volume rendering image of (c), (e) the finger skeleton extracted from (b), (f) the trimmed finger skeleton, (g) the constructed branches from (f), and (h) the linear glyph that is based on (g). In (g), the branches are represented by red lines, and the connections between branches are represented by orange lines. In (g) and (h), the same numbers label corresponding branches.

structures in the finger skeletons. The algorithm of the spanning tree based finger branch extraction is:

Step 1: We identify vertical structures from the Reeb graph of a given finger skeleton. From the graph, we search the longest downward path (i.e., the path with the longest height persistence) to be a new branch by using the breadth-first search. Next, we delete the points and edges of the new branch from the given graph, and repeat Step 1 until the given graph becomes empty. After obtaining all the

branches, we insert the longest one into a first-in-first-out (FIFO) queue, and mark this one to be enqueued.

Step 2: We build connections between branches. We obtain a branch from the queue, and identify which unmarked branches have edges connecting with it in the original graph; we then record such connecting edges. We insert the newly identified branches into the queue, and mark these branches to be enqueued.

Fig. 3.5g shows the constructed branches and the connections between the branches. We visualize constructed branches by tree-based visualizations to minimize occlusion, such as Fig. 3.4d and Fig. 3.5h.

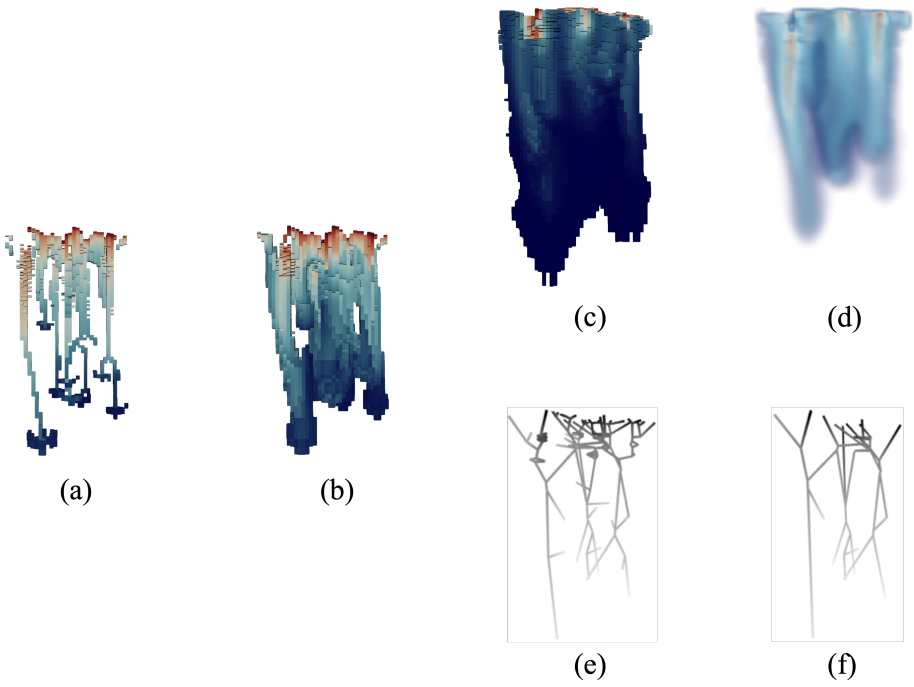


Figure 3.6: We display a complex finger including (a) the ridge voxels, (b) the finger core, (c) the complete finger volume, (d) the volume rendering image of (c), (e) the finger skeleton extracted from (b), and (f) the trimmed finger skeleton.

3.3.2.3 Trim of Finger Skeletons with Removal of Short Branches and Cycles

We trim the Reeb-graph based finger skeletons. When a finger structure is complex, essential geometric information may be occluded in the full skeleton. For example, in Fig. 3.6e, short branches occlude persistent branches and hinder the perception of the overall geometric structure. Hence, we prune the finger skeletons to preserve the most relevant geometric information of the fingers and minimize the occlusion (R2.3). The trim of the Reeb-graph based skeletons is based on the height persistence (R1.2). Suggesting by previous works discussed in Sect. 3.1.2, we remove short branches and short loops to prune finger skeletons.

The trimmed skeletons are shown in (f) of both Fig. 3.5 and 3.6. As a result of the trim, the geometric structure of fingers can be readily understood.

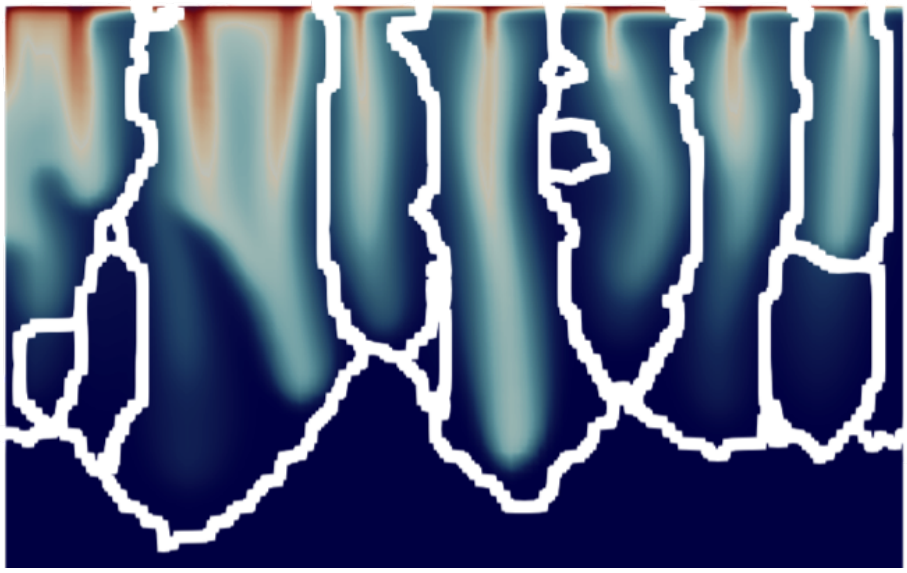


Figure 3.7: We show the segmented fingers, where the segmentation boundaries are represented by white lines.

3.3.3 Extraction and Tracking of Finger Volume

We extract complete finger volume, and track the volume of fingers and branches based on volume overlapping.

3.3.3.1 Volume Segmentation for Extraction of Complete Volume of Individual Fingers

We recover the complete volume of individual fingers by segmenting the 3D density field into connected subfields. We segment finger cores first. Almost all finger cores are connected through the high-density hexagonal cells in the top layer by observed from Fig. 3.2, where the *top layer* is the diffusive boundary layer contained in the top grid cells. Hence, when separating finger cores, we ignore the part of finger cores in the top layer, as suggested by the previous works [5, 98, 99]. We select a fixed height value as the separation between the top layer and the bottom layer of the whole domain through visual inspection by following the previous works [5, 98, 99]. Given the height of the whole domain ranging from 40 to 0 in this application, we find that the fingers are separated well throughout all timesteps for a fixed height value of 38. For other datasets, our visual-analytics system provides visualizations for scientists to identify a suitable value. We then obtain each individual finger core as a connected component of the finger core voxels, excluding the top layer. After that, we use the watershed traversal method [164] to extract the complete volume of individual fingers from the 3D regions with non-zero densities by expanding from individual finger cores.

We display boundaries between the segmented fingers in Fig. 3.7 by using white lines. Fig. 3.4e presents the extracted voxels of a finger; more complex examples are displayed in (c) and (d) of both Fig. 3.5 and Fig. 3.6. The segmented fingers satisfy the cognitive requirement of the earth scientist regarding individual fingers.

3.3.3.2 Volume Overlapping Based Tracking of Fingers and Branches

We track fingers and their branches (R3). The volume overlapping based tracking method [147] is adopted because fingers usually flow downward and do not move with a significant horizontal deviation. We first relate fingers between consecutive timesteps with volume overlapping (i.e., that share grid cells). The number of shared cells and densities of these shared cells reflect the strength of connections between related fingers. Also, the shared cells have positions that reveal where the volume of the fingers overlaps. After identifying the correspondence between fingers, we further relate branches of corresponding fingers also based on the volume overlapping.

3.4 Spatio-Temporal Visualizations of Viscous and Gravitational Fingers

We create an interactive visual-analytics system to perform spatio-temporal analyses on the geometric structures of the viscous and gravitational fingers. In our system, we present juxtaposed visualizations so that users can compare fingers over space and time. Fig. 3.8 shows the complete visual-analytics system, which consists of six panels marked as (a)-(f). Regarding spatial exploration (R2), (a) the depth selection panel allows users to select a depth of interest in the spatial domain. Then, (b) the density field slice view displays the density field at a selected depth as a 2D heatmap. (c) the spatial projection panel presents a high-level spatial view, and uses convex hulls to indicate the extent of fingers projected on the x-y plane. Users can also observe different geometric features (R1) of fingers in detail through (d) a volume rendering image and (e) a 3D skeleton visualization. Finally, at the bottom of Fig. 3.8, (f) the geometric-glyph augmented tracking graph displays the evolution of fingers (R3). All

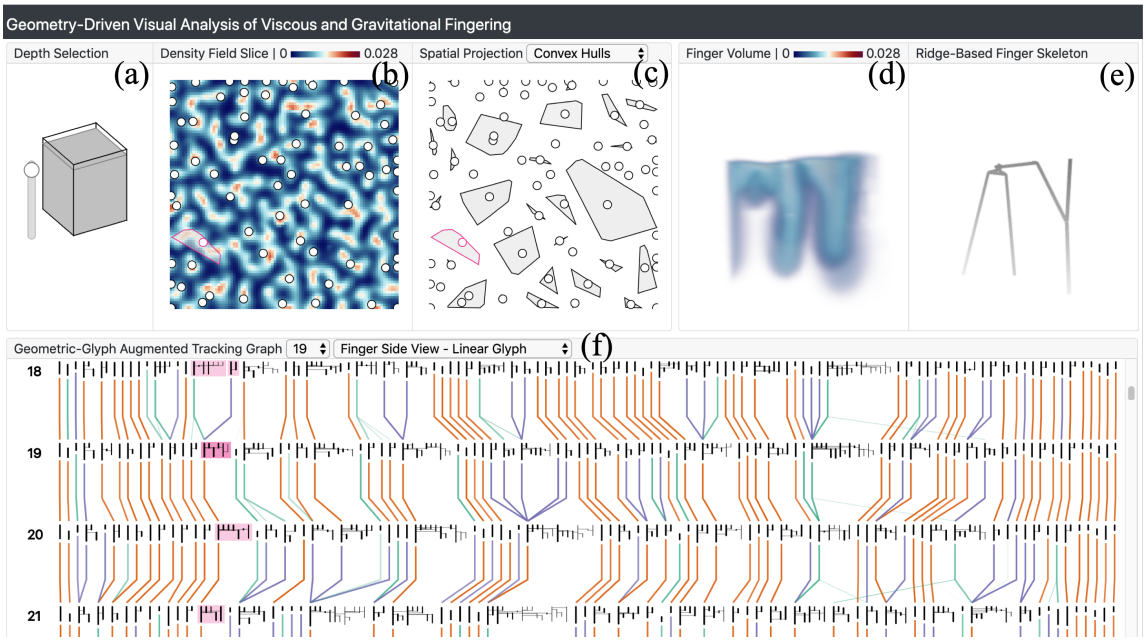


Figure 3.8: A geometry-driven visual-analytics system is used to study viscous and gravitational flow instabilities. (a) Depth slider is used to specify a depth of interest. The 3D domain under the specified depth is shown for analysis. (b) View of a density slice at the specified depth, which also shows centroid points of fingers projected onto the x-y plane. (c) The spatial projection view displays the fingers projected onto the x-y plane using scattered convex hulls. (d) The finger volume view displays the density field of a selected finger. (e) The finger skeleton view shows the geometric structure of a selected finger in 3D space. (f) Geometric-glyph augmented tracking graph visualizes the geometric evolution of fingers based on geometry-driven planar glyphs.

the views in our system are interactive and coordinated together to enable coherent visual analyses. In the following, we describe each of these panels in detail.

3.4.1 Spatial Visualizations of Fingers

3.4.1.1 Depth Selection and Density Field Slicing

Density field slicing is an important tool for scientists to study the change of densities inside fingers in space (R2). In the depth selection panel (Fig. 3.8a), our

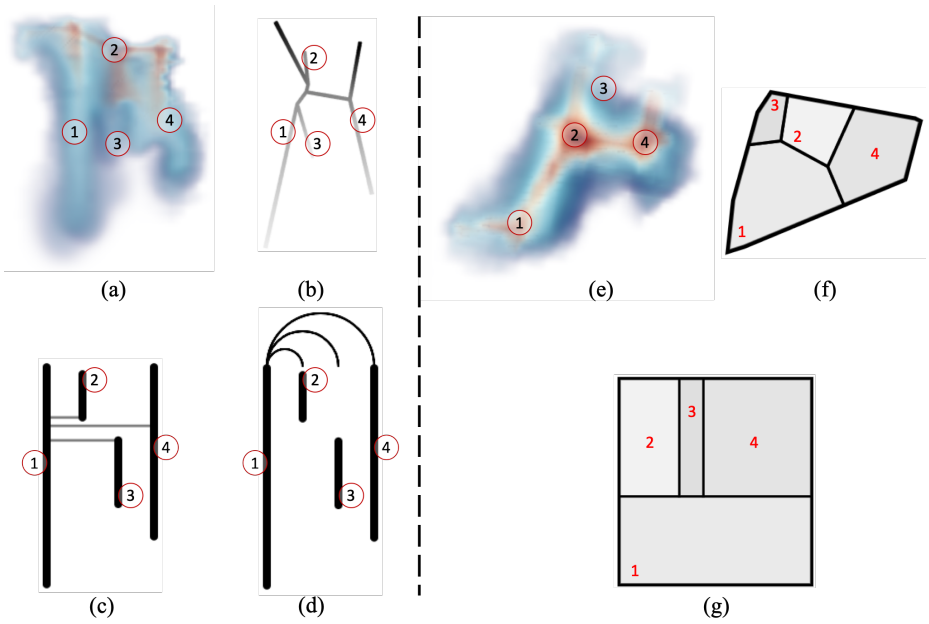


Figure 3.9: We showcase glyph designs of a finger. The same numbers label corresponding branches. The left four views display side views of the finger. Image (a) shows the side view of the finger volume. Image (b) is the skeleton of the finger. The linear glyph (c) displays connections between branches of the finger. When horizontal connections are too close or even overlap, the alternative design (d) shows the tree structure with less ambiguity than (c). Right three views display top views of the finger. Image (e) is the top view of the finger volume. The Voronoi glyph (f) is shown on the spatial projection panel; Voronoi cells that are inside the convex hull are corresponding to the finger branches. The rectangular glyph (g) is for contrasting the topological complexity between the finger branches on the tracking graphs.

system allows users to drag a slider to select a depth of interest; the 3D spatial region below the selected depth is highlighted by gray color in a 3D cube. The density field slice at the selected depth is then extracted and shown as a 2D heatmap in Fig. 3.8b. The finger volume and skeleton views (Fig. 3.8 d and e) are updated to show finger structures under the selected depth only. The selected depth is set to the top of the 3D domain initially, because CO_2 is injected from the top of the aquifer.

3.4.1.2 Spatial Projection of Fingers

A high-level spatial projection view is necessary to reveal how fingers distribute horizontally (R2.2). Convex hull and Voronoi treemap are further applied to display the projected individual fingers, as detailed below.

Convex hull: We project critical points of finger skeletons onto the x-y plane. We then draw a convex hull to enclose the projection of the critical points of each finger. The convex hull representation, shown in Fig. 3.8c, demonstrates the coverage of each finger in space, and help scientists identify large fingers instantly. This view also supports the selection of a finger of interest to display the volume and skeleton of the selected finger. The convex hull of a selected finger is also superimposed on the density field slice view, as shown in the lower-left corner of Fig. 3.8b to confirm the existence of the selected finger in the density field. In addition, we project centroid points of finger skeletons onto the x-y plane for both the density field slice and spatial projection view.

Voronoi treemap: To reveal information of finger branches (R1.1), we use the Voronoi treemap [11] to represent each finger branch by a Voronoi cell, and embed the Voronoi cells in the convex hulls of fingers instead of centroid points as shown in Fig. 3.9f. The relative positions of branches of a finger are revealed by the positions of Voronoi cells within the convex hull of the finger. The extent of a branch is represented by the cell extent. Dark Voronoi cells mean that the corresponding finger branches exist in the deep region of the 3D domain. More precisely, the darkness of Voronoi cells encodes the average depth of critical points that are in the corresponding finger branches. Users can select to observe either centroid points for simplicity or Voronoi cells for details of finger branches.

3.4.1.3 3D Spatial Visualizations of a Selected Finger

We provide 3D volume- and skeleton-based visualizations for geometric analyses (R1) and spatial exploration (R2) after selecting a finger of interest. Users can interactively rotate both of the volume and the skeleton to study a selected finger from different viewpoints, and zoom into the representations to study details of the finger structure.

Volume visualization: The volume rendering image of an individual finger, as shown in Fig. 3.8d, visualizes how the density of the finger distributes in the physical domain, which allows scientists to interpret the finger intuitively. Moreover, the ray casting based volume rendering remedies the occlusion (R2.3) on the direct display of voxels (e.g., Fig. 3.6c) with transparency.

Geometric skeleton visualization: To visualize the overall geometric structures of fingers in 3D space (R1) and analyze how fingers grow vertically (R2.1), we display finger skeletons on the screen using orthogonal projection such as Fig. 3.9b. To recover the density information of fingers, we use intensity gradients along lines to indicate the density changes of finger branches. As a result, high-density branches are highlighted, and low-density branches are under-emphasized.

3.4.2 Geometric-Glyph Augmented Tracking Graph for Temporal Visualization of Fingers

The geometric-glyph augmented tracking graphs, as shown in Fig. 3.8f and Fig. 3.10, visualize the evolution of fingers to facilitate temporal exploration of fingers (R3). The tracking graphs help scientists identify various evolutionary events of fingers, and analyze how the evolutionary events happen in detail through interactions (R3.1).

Each row of the tracking graph displays the fingers at one timestep; the timestamp is labeled at the left of the panel. Widths of finger glyphs are adjusted according to the topological complexity of finger structures; more complex fingers have wider space for drawing. Users can choose to filter out the glyphs of short fingers if tracking persistent fingers is preferred. Also, users can switch the style of the glyph between the linear glyph (Fig. 3.9c) and the rectangular glyph (Fig. 3.9g) to observe different facets of fingers. Below we describe the tracking graph in more detail, including the generation of glyphs for showing the geometric structures of fingers, the color of links encoding the evolutionary events of fingers, the minimization of link crossings for reduction of visual clutter, and the interactions for exploration of temporal events.

3.4.2.1 Linear Glyph for Finger Side View

To display the evolution of geometric structures of fingers (R3.1), we nest geometry-driven glyphs on the tracking graphs. To visualize the side view of fingers over time, we draw finger branches and their connections in a plane (R1.1) with minimized occlusion (R2.3) and also preserve the height of branches (R1.2). Two possible designs are discussed in the following.

Projection: We may project finger skeletons onto a plane. However, the traditional orthogonal projection suffers from edge crossing problems. To remedy the edge crossing for the projection of tree-like structures, Marino and Kaufman [103] produced radial planar embeddings of the structures. However, fingers are vertical objects, and radial planar embeddings of fingers may lose the depth of fingers. Although the method of Marino and Kaufman [103] is good at preserving the curvature of 3D objects, the curvature is not an essential feature of fingers.

Tree drawing: We draw fingers as linear glyphs to capture abstract forms of fingers, as an example illustrated in Fig. 3.9c. Heine et al. [74] proposed a graph-drawing method to plot branches of contour trees in a plane with minimized edge crossings (R2.3). The method of Heine et al. [74] represents branches by vertical lines and denotes links between branches by horizontal lines. Connections between branches (R1.1) and vertical features of branches (R1.2 and R2.1) are shown clearly in the results of this method. Thus, we augment the tree-drawing method of Heine et al. [74] to display the side view of fingers, such as in Fig. 3.9c. Note that, although trees are planar graphs, crossings of tree edges are inevitable for some instances by using this design; the reason is that only the x-axis is free to arrange tree branches, and the y-axis is used to encode the height of branches. We draw the longest branch of a finger at the leftmost of the glyph so that we can locate the principal branch quickly. Users can choose to encode change of densities along branches by using the gradient darkness of vertical lines such as in Fig. 3.4d. When hovering over a linear glyph on the tracking graph, connections between branches of this finger become arcs hanging at the top of the glyph to reduce visual clutter (R2.3), such as in Fig. 3.9d. The corresponding tracking graph is shown in Fig. 3.8f.

3.4.2.2 Rectangular Glyph for Finger Top View

We draw the top view of fingers to display quantitative geometric attributes (R1) and relative positions (R2.2) of finger branches by using treemap based rectangular glyphs. An example is illustrated in Fig. 3.9g. The treemap technique [82] maps elements onto a rectangular region in a space-filling manner and displayed quantity values of elements effectively. The spatially ordered treemap [169] extended from squarified treemaps [20] arranges the rectangles of elements based on both the spatial

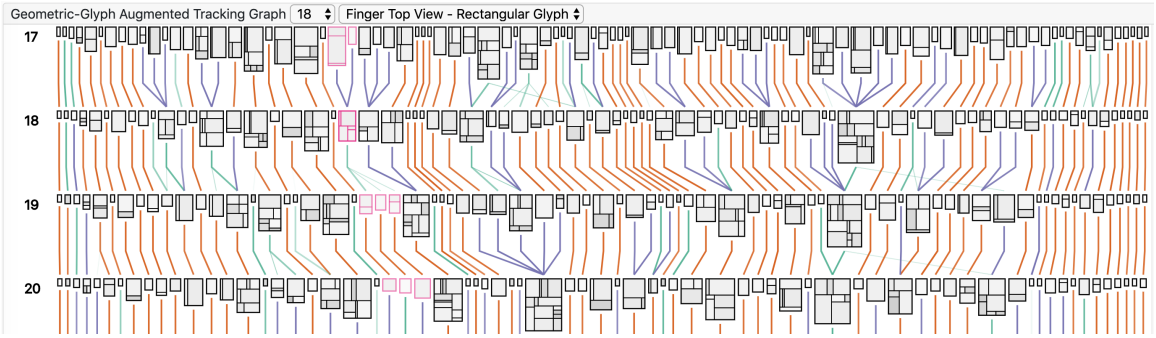


Figure 3.10: We display the geometric-glyph augmented tracking graph with rectangular glyphs. A finger associated with merging and splitting events at timestep 18 is highlighted by red-violet and analyzed in Fig. 3.13 further.

proximity and the balance of aspect ratio, and, can display the spatial distribution of elements (R2.2). Hence, we use the spatially ordered treemap [169] to generate rectangular glyphs for finger branches, as shown in Fig. 3.9g. Each branch is represented by a rectangle on the glyphs. Areas of rectangles are able to encode the numeric attributes of branches, including statistics or topological measurements. In this work, the area encodes the topological complexity of the corresponding finger branch (R1.1). Intuitively, a finger branch having more critical points usually indicates that this finger branch is connected to more other branches and thus is more complex in terms of its topological structure. Hence, we define *topological complexity of a finger branch* by the number of critical points on the branch skeleton. Moreover, to compare fingers in the tracking graph and assign more space for topologically complex fingers, we encode the size of rectangles of whole fingers by the topological complexity of fingers. The *topological complexity of a finger* is defined by the number of critical points on the finger skeleton; intuitively, a finger has more critical points, usually indicates this

finger has more branches and hence is more complex in its topological structure. The corresponding tracking graph is shown in Fig. 3.10.

3.4.2.3 Link Encoding for Evolutionary Events of Fingers

Links represent the temporal relationships between fingers. When time varies, fingers may grow, merge with other fingers, or split into multiple fingers. To indicate these three types of finger evolution, we use three hues of links following the qualitative color advice of ColorBrewer [19].

Brown link: Brown links indicate that fingers grow with minor changes between consecutive timesteps. Specifically, at least seventy-five percent volume of the finger is preserved in one of the connected fingers at the subsequent timestep.

Blue link: A blue link indicates the case when multiple fingers merge into one finger at the subsequent timesteps. Specifically, the finger at the subsequent timestep incorporates seventy-five percent volume of each of the fingers at the previous timestep.

Green link: A green link indicates that a finger splits into multiple fingers at the subsequent timestep, and none of the fingers at the subsequent timestep have seventy-five percent volume of the finger at the previous timestep.

We use the opacity of links to encode link weights. The *weight of a link* is the size of the overlapping volume between linked fingers. If fingers have weak connections, their links are almost transparent so that visual clutter is reduced.

3.4.2.4 Iterative Minimization of Link Crossings

We reduce link crossings to alleviate the visual clutter of the tracking graphs. Since links are weighted, the reduction of link crossings between every two rows of glyphs is a graph drawing problem: edge crossing minimization for weighted bipartite graphs. Çakıroğlu et al. [25] enhanced and tested five heuristic methods for this graph drawing problem, and concluded that W-GRE [25, 175] method produces the best results. Hence, we utilize W-GRE method in our application to minimize intersections of weighted links.

We propose a W-GRE based iterative algorithm for the crossing minimization of multiple rows, which arranges finger glyphs of each row iteratively until obtaining a good result. We describe the algorithm in the following.

Step 1: The finger glyphs in the first row are arranged by the ascending order of the x coordinates of the finger centroids initially.

Step 2: We order glyphs from the second row to the last row. We minimize the crossings of the links between a given row and its previous row by using the W-GRE [25] method.

Step 3: We order glyphs from the second-to-last row to the first row. We minimize the crossings of the links between a given row and its following row by using the W-GRE [25] method.

Step 4: Repeat Step 2 and Step 3 to order the fingers of each row until reaching the convergence of the finger order. Usually, repeating two times can produce a good result in our experiments.

3.4.2.5 Interactive Finger Tracking

We offer three interactions for scientists to track fingers.

Finger selection: Scientists can select a finger of interest by click on the finger glyph from the tracking graphs. Then, the glyphs of the selected finger and the other related fingers are highlighted in the tracking graphs by coloring the backgrounds (Fig. 3.8f) or the strokes (Fig. 3.10). Additionally, the details of the selected finger are displayed in the volume and skeleton views.

Finger volume tracking: The earth scientist is also interested in tracking the volume of a finger. Note that each link is associated with the overlapping volume between temporally related fingers. Users can click on a finger of interest first and click on one of the associated links to observe overlapping volume between this finger and the other finger in the finger volume view, which is illustrated in Sect. 3.5.2.2 and Fig. 3.13 in detail.

Branch tracking: The earth scientist requires designs to track geometric structures of fingers (R3.1). If a complex finger separates into multiple smaller fingers over time, it is essential to track each branch of the complex finger comes to which smaller finger entity afterwards; also, if multiple fingers fuse into a complex finger, it is important to identify the correspondence between the individual fingers and the branches that they got merged to. On our tracking graph, we identify corresponding branches between linked fingers interactively. When hovering over a link between two connected fingers, the relevant branches between the fingers are highlighted by the red-violet color on glyphs, as shown in Fig. 3.13.

3.5 Case Studies and Scientist Feedback

As we were developing our system, we were in close contact with two earth scientists including the one who is referred to in Sect. 2.1.2. In the following, we discuss the use cases that were studied by the earth scientists when they used our geometry-driven visual-analytics system to explore the spatio-temporal phenomena of viscous and gravitational fingers. Also, we present their feedback and suggestions after they performed domain-specific tasks.

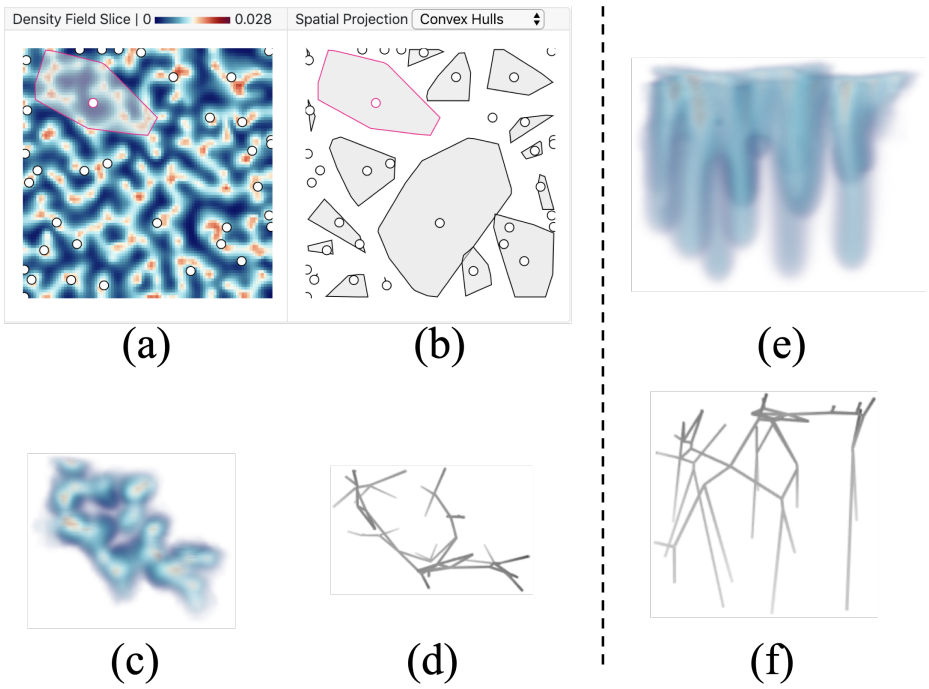


Figure 3.11: The earth scientists identified a finger of interest from (b) the spatial projection panel. Then, they observed the high-density hexagons under the convex hull of the finger in (a) the density field slice view. They further confirmed the correspondence between (c) the top view of the finger volume and the hexagons in (a). Also, they found the finger skeleton (d) preserved the geometric structure of (c). Next, they looked at the side view of the finger volume (e) but felt difficult for observing finger branches due to the occlusion. They then interacted with the 3D skeleton (f) and obtained how these branches form in space over time.

3.5.1 Case 1: Spatial Analysis of Geometric Features of Fingers

Because the existing methods have limitations in capturing geometric patterns of fingers, the earth scientists mentioned the difficulty in identifying branching structures of fingers in 3D space. There are two challenging problems. The first is to identify where the fingers and branches are in space. After the identification, the second problem is to comprehend how the high-density hexagonal sheets in the top (e.g., Fig. 3.11 c and d) split and develop into columnar fingers (e.g., Fig. 3.11 e and f) spatio-temporally. Specifically, given hexagonal cells (e.g., Fig. 3.11 c and d), it is important to know that whether the fingers tend to form along with the boundaries of hexagonal cells or form below the centers of the hexagonal cells. By using our system, the earth scientists were equipped to solve the problems efficiently.

The earth scientist identified where are the fingers and branches by observing spatial visualizations. They first looked at the density field slice view, Fig. 3.11a. The scientists thought the projection points added on the view is an effective addition to the traditional density-slice views, which allowed them to identify finger locations on any slice with the corresponding hexagonal features at the slice. Furthermore, from the spatial projection panel, Fig. 3.11b, the earth scientists found two fingers that occupy a large space. They clicked on one of the two for analysis, and the convex hull of the finger was superimposed on the density field slice view, Fig. 3.11a. They focused on the high-density hexagons under the convex hull in the slice view, and adjusted the z-value slider of to understand the change of densities along the z coordinate. Note that the finger volume (Fig. 3.11c) and skeleton (Fig. 3.11d) views were displayed after the selection of the finger. They verified the correspondence between the hexagons under the convex hull in Fig. 3.11a and the top view of the volume, Fig. 3.11c. Next, they

confirmed the correspondence between the finger volume and skeleton. The skeleton Fig. 3.11d captured the geometric structure of the volume Fig. 3.11c. Afterwards, they rotated the finger volume and skeleton to see side views of the finger. The finger branches occluded severely in the volume visualization Fig. 3.11e. However, when interacting with the skeleton in Fig. 3.11f, they intuitively acquired the finger branches in space. Hence, they thought the extracted finger skeletons (e.g., Fig. 3.11f) were effective in identifying the branching structures of fingers.

The scientists afterwards understood how CO_2 volume flows and how fingers form in space by using our system. Since fingers result from CO_2 flows in space over time, by observing the skeleton (Fig. 3.11f), the earth scientists quickly captured the downward flowing track of CO_2 by following the skeleton and comprehended how the finger and its branches grow. Moreover, from the finger skeletons (e.g., Fig. 3.11f), the experts obtained an insight that the fingers usually formed along the boundaries of hexagonal cells rather than forming below the centers of hexagonal cells.

3.5.2 Case 2: Temporal Analysis of Fingers

We present how to perform temporal analysis of fingers by using the geometry-driven visual-analytics system.

3.5.2.1 Case 2.1: Recognition of Evolutionary Phases

According to the earth scientists, the evolution of the instability has (at least) three important phases: onset, growth, and termination. The earth scientists identified the three phases based on the spatial projection panel, as shown in Fig. 5.5. In the onset phase (Fig. 5.5 a_1 and a_2), the instability is triggered, and the displacement front breaks up into many small-scale fingers. After the onset phase, the dense fingers

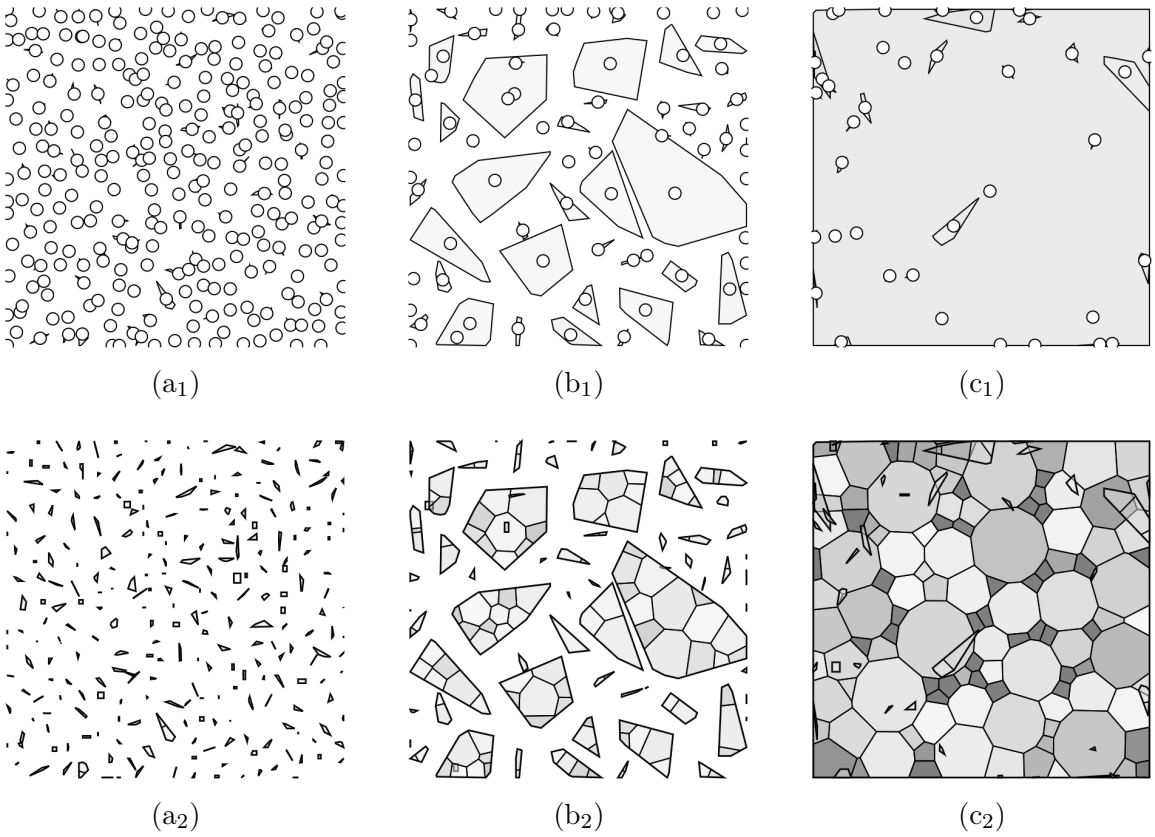


Figure 3.12: There are three important phases for the evolution of fingers: (a) the onset phase, (b) the growth phase, and (c) the termination phase. Convex hulls of (a_1), (b_1), and (c_1) display the spatial coverage of individual fingers. Voronoi glyphs of (a_2), (b_2), and (c_2) show finger branches.

grow non-linearly and penetrate the underlying lighter fluid. In Fig. 5.5 b_1 and b_2, many medium-sized fingers form in this growth phase. These medium-sized fingers grow, merge, and/or split over time. Eventually, multiple fingers merge to form a few 'mega-plumes' that span most of the reservoir, as shown in Fig. 5.5 c_1 and c_2. Once those fingers reach the bottom of the domain, the instability shuts down or terminates.

3.5.2.2 Case 2.2: Exploration of Evolutionary Events

The earth scientists identified the growth, merging, and splitting of time-varying fingers from the geometric-glyph augmented tracking graph. During our evaluation session, first, we illustrated the encodings of our designs to the earth scientists. After we explained the three colors of links used to represent the growing, merging, and splitting events, they appreciated the temporal evolution of the fingers was intuitively displayed. We next explained the two kinds of finger glyphs to the earth scientists. Compared with previous tracking graphs [5, 98, 99] that only use dots to represent fingers, the geometric glyphs of our tracking graphs offered more details and facets of fingers to the earth scientists. With respect to the linear glyphs (e.g., Fig. 3.9c), at first glance, they were confused with the encoding of horizontal lines initially. However, after our explanation, they were able to understand the horizontal lines representing connections between vertical branches and thought this design was effective. They thought drawing fingers in a plane would cause distortion, but found that the branches and their connections were shown clearly, and it was easy to count the number of vertical branches. In regards to the rectangular glyphs (e.g., Fig. 3.9g), the expert captured the horizontal distribution of branches and quickly counted the number of complex branches. Also, they suggested highlighting the correspondence between the rectangles and the branches in the volume so that they could understand how the structures of fingers from the glyphs better.

Here we illustrate how the earth scientists analyzed the evolution of fingers and branches by using our system. First, they found a finger of interest with merging and splitting events, which is highlighted by red-violet in Fig. 3.10 and extracted in Fig. 3.13. Concentrating on the merging event of this finger, they hovered on the left

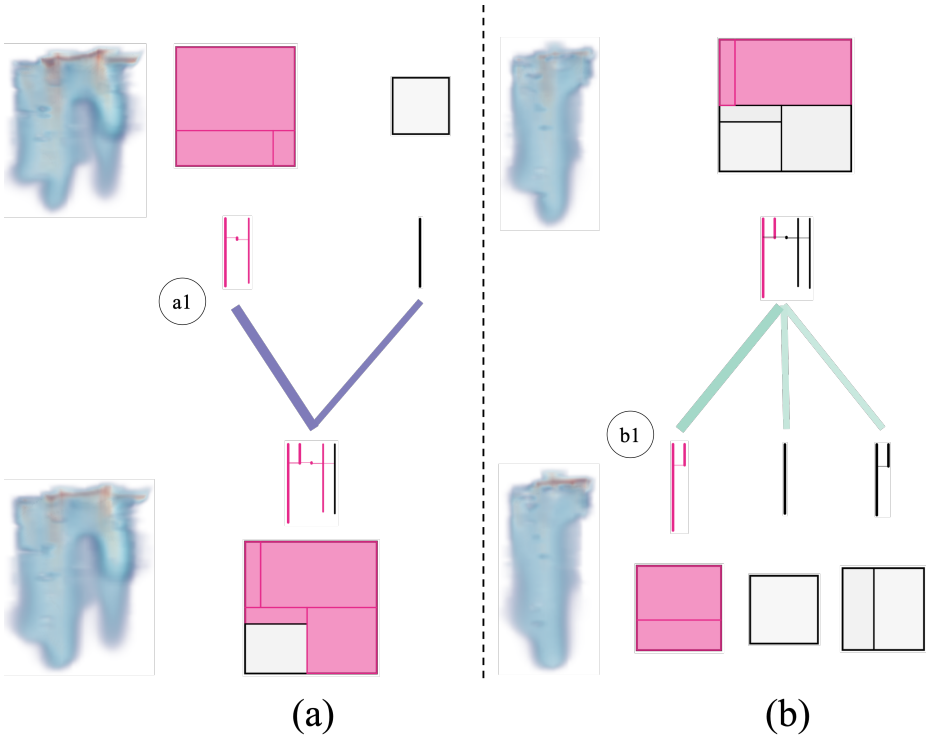


Figure 3.13: After evolving one timestep, (a) fingers may merge into a complex finger, or (b) a complex finger may split into more fingers. To track branches, users can hover over a link; the link becomes thick, and associated branches on glyphs are highlighted by red-violet color. To track the volume of fingers, users can click on a link; then, the overlapping volume between linked fingers is shown to be compared.

link above the merged finger (Fig. 3.13a), and hovered on the leftmost link below the finger (Fig. 3.13b) to track the corresponding branches that were highlighted by red-violet in glyphs. The corresponding linear glyphs and rectangular glyphs between consecutive timesteps were found to be consistent. To track the volume, they first clicked on (a1), the upper-left finger of Fig. 3.13a, to look at its volume; then, they clicked on the merged finger at the next timestep and clicked on the link to (a1) to observe the partial volume of the merged finger that comes from (a1). The two volume images are shown on the left of Fig. 3.13a, and the temporal change of the

corresponding volume was observed. Similarly, they tracked the overlapping volume of fingers in the splitting event, Fig. 3.13b, and observed that the corresponding branches grew longer. After using our system, the earth scientists thought our tracking graphs were valuable to identify when and where the fingers grow, merge, and split over time. Compared with the previous works that do not construct branches of fingers explicitly, our methods tracked finger branches more efficiently.

3.5.3 Scientist Feedback

After using our system, the earth scientists thought that our geometry-driven system provided a new perspective on the analysis of viscous and gravitational fingering. Specifically, the visual-analytics system helped scientists acquire branches in space and branching behaviors over time efficiently and effectively, which relieved cumbersome work for the geometric analyses of fingers manually, since the branch tracking offered by our system is new and not directly available by using previous methods [5,49,98,99]. They also acknowledged that the extraction of fingers by our method was effective and that the minimal occlusion in visualizations leading to clear representations of fingers.

3.6 Discussion

We discuss limitations of our methods.

Sensitivity of the ridge voxel detection: If the quality of the data is low, for example, resulting from noise or coarse discretization, the approximation of derivatives may have higher errors leading to inaccurate detection of the ridge voxels. In addition, if the density fields are not smooth enough, the detected ridge structures can be fragmented. Although the quality of the data used in this paper is sufficient to extract high-quality ridges, methods for improving the data quality like noise removal

could be used to improve the stability of the ridge detection algorithm. Moreover, as detected ridges are disconnected for other applications, one may acquire more connected features by using a similar way in Sect. 3.3.1.2.

Limitation of using a static top layer: Although previous works [5, 98, 99] and this paper use a static top layer (Sect. 3.3.3.1) and acquire good results, physically, the top layer initially grows diffusively in time, i.e., as the square root of time, which is much slower than the convective time scale over which the fingers grow. Once the instability is triggered, this diffusive boundary layer becomes thin. Since the top layer is varying in nature, one limitation of using a static top layer is which may not accurately segment the structures near finger roots.

Chapter 4: Asynchronous and Load-Balanced Union-Find for Distributed and Parallel Scientific Data Visualization and Analysis

As the scale of scientific data generated by experiments and simulations grows, it becomes a common practice to use High-Performance Computing (HPC) clusters to analyze and visualize data in parallel. In such distributed and parallel computing environments, data parallelism is a default paradigm; input data are partitioned into data blocks, which are distributed among parallel processes. With data-parallelism, intermediate results are produced from individual data blocks before they are merged into the final result.

This paper focuses on union-find, which is widely used in many scientific visualization algorithms [28, 51, 72, 104, 115, 116, 123, 126] and plays a key role in merging disjoint sets. For example, in the extraction of super-level sets [51], regions with scalar values larger than a given threshold are extracted. Specifically, in 3D volumetric data, union-find merges neighboring voxels that are greater than the given threshold into connected components. In critical point tracking [57, 159], union-find connects critical points detected in a spacetime gird, which allows effective tracking of time-dependent phenomena.

In the parallelization of the abovementioned visualization algorithms, we identify two scalability bottlenecks in distributed union-find: (1) high synchronization costs and (2) imbalanced workloads. First, because each disjoint set is usually distributed in a subgroup of processes, the use of global synchronizations may block the rest of the processes, causing busy waits in program execution. Up to date, distributed union-find [37, 51, 72, 80, 102] has been implemented with the bulk-synchronous parallel programming model [160], which manages parallel processes to alternate local computations and global synchronizations iteratively until distributed algorithms converge. In the context of distributed union-find, local computations consist of performing set operations on local data, and global synchronizations are used for merging and updating disjoint sets across processes. Second, imbalanced workloads between parallel processes lead to additional busy waits. The workload imbalance is caused by processes that possess imbalanced disjoint-set elements. For example, in super-level set extraction, data blocks in some processes may find more voxels above the given threshold than others. Likewise, in critical point tracking, critical points may be non-uniformly distributed in a domain. In this work, we present novel solutions to reduce synchronization costs and balance processes' workloads for distributed union-find, as described below.

First, we eliminate the use of global synchronizations for distributed union-find based on the fact that set unitings are order-independent. We prove that it is possible to eliminate global synchronizations by overlapping synchronizations with local computations and guarantee algorithm convergence and final-result correctness. It enables the use of asynchronous point-to-point communications to merge and update

disjoint sets across processes in practice, which can be fully overlapped with local computations.

Second, we balance the disjoint-set elements in each process for workload balancing in distributed union-find because the time complexity of local computation is proportional to the number of set elements [54]. In our implementation, a k-d tree space decomposition is used to redistribute the set elements evenly among processes because k-d trees can be used to effectively balance spatial data, which is the common type for scientific datasets.

We demonstrate the scalability of our distributed union-find algorithm by measuring the performance with up to $1,024$ processes for both critical point tracking and superlevel set extraction. Benchmark datasets include 3D spatial synthetic data and 2D time-varying application data output by high-speed imaging and fusion plasma simulations. We show that our algorithm achieves shorter execution time and better scalability than the existing distributed union-find methods in scientific datasets. In summary, the main contributions of this paper are twofold:

- We prove global synchronizations between processes can be eliminated in distributed union-find, and present a method that allows distributed union-find to overlap communications and local computations, which reduces processes' busy-waiting time.

- We redistribute the disjoint-set elements across processes evenly for load balancing of distributed union-find using a k-d tree decomposition scheme, which improves algorithm scalability in scientific applications.

This work has been published in IEEE Transactions on Visualization and Computer Graphics [171] in 2021.

4.1 Preliminaries

This section reviews the serial union-find and the bulk synchronous parallelism based distributed union-find methods.

4.1.1 Serial Union-Find

In general, the input of union-find algorithms is a graph, $G =< V, E >$, where V is the collection of all elements and E is the collection of all edges between elements. The output consists of disjoint sets between which no connecting edges exist. Union-find has two basic operations: `Union`(v, v') and `Find`(v), where `Union` unites disjoint sets that two edge-connected elements belong to and `Find` returns the representative element of the set containing the given element. A `Union` operation is performed on every two edge-connected elements to output final disjoint sets. To manage disjoint sets efficiently, the internal implementation of union-find uses a tree data structure [54, 55] as described below.

Disjoint-set trees: Each disjoint-set tree corresponds to a disjoint set and has a *root* element, which is the representative element and the identifier of the set. Every element has a parent pointer pointing to a *parent* element and is a *child* of that parent. Every root's parent is itself.

`Union` and `Find` with disjoint-set trees: A `Find`(v) operation is implemented by following the parent pointers starting from the given element to identify the root. A `Union`(v, v') operation is performed on a pair of edge-connected elements using two sub-operations: (1) *set uniting* and (2) *edge passing*. If an edge connects two roots, a

set uniting is performed to point one root to the other to form a single disjoint-set tree. Otherwise, the edge is passed through parent pointers of the given elements to `Find` the roots; after the edge is passed to the roots, a set uniting will be invoked so that the sets of the given elements are merged. The set uniting can follow either *uniting by rank* [54, 72, 102, 155] or *uniting by size* [54] rule, which unites one set into the other with a higher rank or a greater size.

Path compression: Path compression is a process to shorten the paths from elements to roots in disjoint-set trees by pointing elements to either (1) their grandparents [162, 167] or (2) their roots [79], where the two choices were proved to have the same amortized time complexity in [54]. Path compression can make `Find` and the edge passing operations identify elements' roots with fewer iterations, hence, more efficient.

4.1.2 Distributed Bulk-Synchronous Union-Find

Distributed bulk-synchronous union-find [37, 72, 80, 102, 123] iterates over local computations, synchronous communications, and synchronous termination detection.

Local computations: Each process is responsible for handling local work, including processing incoming messages, performing `Union` operations and path compressions for local elements stored in the process, and queuing outgoing messages.

Communications: Processes use synchronous communications to perform inter-process (1) set uniting, (2) edge passing, and/or (3) path compression. First, the set uniting following uniting by rank or uniting by size rule requires processes' synchronizations to exchange ranks or sizes of sets to be united and avoid creating cycles in resulting disjoint-set trees. Second, processes may pass edges of local elements

to the processes of elements' parents. The element parents' processes receive the edges and continue passing edges until reaching roots for future set unitings. Third, for path compression, processes may issue queries about elements' grandparents to elements' parents. Then, the processes of elements' parents send the information of grandparents back after receiving the queries.

Termination detection: A *collective synchronous communication*, such as `MPI_Allreduce` used in [80], is applied to check whether all processes have completed assigned work periodically for iterations to terminate. The "collective" means all available processes participate in the communication. The "synchronous" means the communication blocks the participating processes for an agreement of termination and will not return until all the participating processes respond, ensuring all the participating processes agree when it is ready to terminate.

4.2 Overview and Design Considerations

Each process's input consists of elements, a subset of V, and the input elements' edges. Elements in different processes are not overlapping. Every element has a unique ordinal *identifier (ID)* and numeric coordinates. The output consists of disjoint-set trees distributed among the processes.

Algorithm overview: We give examples for involved operations in Fig. 4.1. The algorithm initialization includes redistributing input elements and edges for the load balancing (Section 4.4) and creating a disjoint set for every element. After the initialization, each process iterates over local computations (Algorithm 1), asynchronous communications, and asynchronous termination detection (Section 4.3.4); each process remains in the iterations as long as there is still remaining local work or incoming new

message. In the local computations, each process first consumes incoming messages and then repeats set uniting (Section 4.3.2), path compression (Section 4.3.3), and edge passing (Section 4.3.2) until local work is complete. The outgoing messages are immediately sent after they have been produced, and the incoming messages promptly drive a new iteration of operations after the incoming messages have been delivered. After all processes terminate iterations, we have acquired correct disjoint sets. For scientific visualization and analysis, a final local path compression (Section 4.3.3) is performed to label all elements in the same set by the same identifier, i.e., the ID of the set root. As not all distributed iterative algorithms can necessarily be designed to overlap communications and local computations, we prove our algorithm's convergence and correctness in Section 4.3.5.

Algorithm design considerations: We explain four considerations for the design of distributed asynchronous union-find with the challenges when global synchronizations are eliminated.

First, we adopt an asynchronous parallelism pipeline with asynchronous communications and asynchronous termination detection to overlap communications and local computations. Because asynchronous communications do not block participating processes, processes can continue doing local computations without waiting for communications to complete.

Second, we use a *uniting by identifier (ID)* rule for the set uniting by uniting a set into the other with a smaller root ID to support overlapping inter-process and local merging of sets. Although uniting by rank rule and uniting by size rule are frequently used in the literature, when multiple sets with the same rank or size are united, the two rules may cause cycles in resulting disjoint-set trees and lead to deadlocks if processes

are not synchronized [9]. Hence, in shared-memory asynchronous union-find [9], when the sets to be united have equal ranks or sizes, Anderson and Woll used the set records in the shared memory to prevent cycles such that one set is united into the other with a greater record index. Because distributed-memory processes do not have such shared records of sets, elements' identifiers are used instead in our algorithm.

Third, we create local-tree data structures to reduce communications in contrast to the existing distributed union-find [37, 72, 80, 102] without such local trees. In our algorithm, only path compression for local-tree roots require communications with the set roots' processes, and the path compression for non-root elements does not involve communications. For example, in Fig. 4.1d, local root 6 represents the subset consisting of 6, 7, and 10. After 2 unites with 0, only the local root 6 involves communications to point to the new set root 0 for path compression using our algorithm; however, without local trees, process 0 may exchange messages with all the three elements (6, 7, and 10) for path compression, which involves more communications.

Fourth, we consider boundary cases in path compression to solve deadlocks of communications when the asynchronous termination detection is used. After local roots already point to set roots, the local roots may still need to communicate with the set roots for path compression because the set roots may point to other elements after applied union operations. For example, in Fig. 4.1d, local root 6 may need to keep communicating with set root 2 for path compression because 2 may point to another element, i.e., 0 in this example later. A communication deadlock happens when multiple processes with local roots may keep performing such communications with each other even though correct disjoint sets have been built, leading to one of the

processes that may always be active so that iterations may never finish when using asynchronous termination detection. We present a solution in Section 4.3.3.

4.3 Distributed Asynchronous Union-Find

We detail our distributed asynchronous union-find.

4.3.1 Distributed Disjoint-Set Trees

We extend serial disjoint-set trees to create additional data structures for our distributed union-find algorithm.

Local disjoint-set trees are local subtrees formed by local elements of processes, where the *local elements* of a process are elements stored in the process. *Local roots* represent the roots of such local subtrees. Each process can identify local roots by examining local elements that have non-local parents. For example, in Fig. 4.1c, 6, 7, and 10 form a subtree in process 2, and 6 is the local root of the subtree.

A *distributed disjoint-set tree* is formed by all elements of the corresponding disjoint set and may consist of multiple local trees. The roots of the distributed disjoint-set trees are called *disjoint-set tree roots*, or *set root* in short hereafter. Each process can identify the set roots by checking local elements that point to themselves.

Non-root elements represent the elements that are neither local roots nor set roots.

4.3.2 Distributed Union Operations

Similar to the serial union-find, a distributed `Union` operation is performed on a pair of edge-connected elements with two sub-operations: (1) distributed set uniting and (2) distributed edge passing. Distributed edges are passed to set roots following paths in distributed disjoint-set trees so as to be used for set unitings.

Distributed set uniting: The set uniting is based on a *uniting by ID* rule, where two disjoint sets are united by setting the parent of the root of one set to an element with a smaller ID in the other set. For example, from Fig. 4.1 (b) to (c), element 6 sets its parent pointer to element 2; however, 2 will not point to 3 because 3 is bigger than 2, and hence, 2 becomes a (temporary) set root. As a result, a parent has a strictly smaller ID than its children in disjoint-set trees, and each set root is the smallest element in a disjoint set, which ensures no cycles.

The set uniting does not produce outgoing messages. In our implementation, each process points every stored set root to the smallest neighbor element if the process stores or receives any edge connecting the set root with a smaller element. Examples are illustrated in Fig. 4.1c.

To make elements be aware of smaller neighbor elements for set unitings, we store each edge in the same process of its larger endpoint after the data redistribution (Section 4.4) and during the following distributed edge passings. For example, in Fig. 4.1b, the edge between element 5 and element 9 is stored in process 3 so that 9 is aware of 5 and can point to 5 for the set uniting.

Distributed edge passing: Edge-passing transports edges of elements to set roots so that the roots are aware of the edges connecting to other disjoint sets for following set unitings.

Edges are passed through endpoints' parent pointers as follows repeatedly. We replace an edge's larger endpoint with the endpoint's parent. For example, an edge between element 3 and element 2 in Fig. 4.1c is updated so that connecting 0 and 2 in Fig. 4.1d after the replacement. We deprecate the edge if its current endpoints have

the same ID; otherwise, the edge is sent to and stored in the process of its current larger endpoint for a set uniting or additional passings.

4.3.3 Distributed Path Compression

Path compression makes edge passing more efficient by shortening the paths between elements and those connected in the disjoint-set trees with lower IDs until each element's parent pointer points to either a set root or a local root. We detail the path compression algorithm for different types of elements as below.

Non-root elements: Path compression for non-root elements does not produce outgoing messages either. At every iteration, non-root elements modify their parent pointers from their parents to the parents' parents, i.e., their grandparents, if there exist grandparents in the same processes. For example, non-root element 10 changes its parent pointer from 7 to 6 in Fig. 4.1d and remains pointing to 6 until iterations finish.

After the iterations terminate, each process points all the non-root elements to the current set roots, as illustrated in Fig. 4.1f; as a result, all elements within the same set have a common label: the ID of the set root.

Local roots: At every iteration, each local root communicates to its parent, which is in a different process, to query its grandparent. The parent who receives the query sends the queried information back to the local root for it to update its parent pointer. For example, in Fig. 4.1c, after element 8 queries its grandparent, element 4 return element 1 to element 8. When the local root receives the feedback about its grandparent, the local root updates its parent pointer accordingly.

A boundary case is when a local root has a set root parent, the set root records the local root's process ID after receiving a grandparent query from the local root and sends feedback indicating not to request the grandparent again, which reduces future communications and avoids possible deadlocks of communications due to the use of asynchronous termination detection. If the set root later points to another element, the set root notifies its non-local children, which are guaranteed to be local roots in other processes, for path compression. After the notification, the local roots are allowed to send additional grandparent queries if they do not point to set roots. For example, in Fig. 4.1d, set root 2 records the process ID of local root 6 after receiving the grandparent query from 6. When set root 2 later unites with 0, 2 notifies 6 of 0 for path compression.

4.3.4 Asynchronous Nonblocking Communications and Termination Detection

Processes exchange messages using asynchronous communications, which do not block participating processes.

Communication protocols: Three types of messages are exchanged across processes for edge passing and path compression:

Transferred edge message is used to transfer the data of an edge. This message contains element IDs and process IDs of the two endpoints of the edge. This message is used in (1) edge redistribution after load balancing and (2) edge passing across processes.

Grandparent query message is used when a local root requests for its grandparent. This message contains the element ID and process ID of the local root. This message is sent from the process of the local root to the process of its parent.

Grandparent message is used to answer a grandparent query issued from a local root. This message either contains the element ID and process ID of the corresponding grandparent or indicates the local root's current parent is a set root and inhibit the local root from sending grandparent queries again. This message is sent from the process of the local root's parent to the process of the local root.

Asynchronous termination detection: In our asynchronous union-find algorithm, each process exits from iterations when all processes finish local work and no messages are being exchanged. The iteration termination detection is straightforward in bulk-synchronous parallelism but requires careful designs for distributed asynchronous algorithms to ensure each process knows all other processes have finished asynchronously and no messages are being transferred. We follow the `iexchange` module [114] of DIY library [111,113] and the nonblocking termination detection mentioned in [41]. Each process undergoes four states: (1) *active*, (2) *idle*, (3) *ready-to-terminate*, and (4) *terminate*, and exchanges their states using asynchronous communications to achieve a consensus for correct termination. We refer to [41] for details about the transitions between the states, which are also summarized in the supplementary appendices.

4.3.5 Convergence and Correctness

We prove our algorithm converges to the correct result with a finite number of iterations in each process. Our proof has three assumptions. First, input data contain

Algorithm 1: local_computations(comm, in_msgs, elements, edges) // The "comm" is a communicator, such as `MPI_COMM_WORLD`. The "isend" is a point-to-point asynchronous operation for message sending, such as `MPI_Isend`.

```
   /* consume incoming messages                                          */
 1 for each msg in in_msgs do
 2 |   if "transferred edge" in msg then
 3 |   |   add_local_edge(msg, edges)
 4 |   end
 5 |   else if "grandparent query" in msg then
 6 |   |   grandparent ← retrieve_grandparent(msg, elements)
 7 |   |   comm.isend(grandparent)
 8 |   end
 9 |   else if "grandparent" in msg then
   |   |   /* pass compression of local roots                           */
10 |   |   point_to_grandparent(msg, elements)
11 |   end
12 end
   /* perform distributed asynchronous union-find operations             */
13 while have local work do
14 |   set_uniting(elements, edges)  ▷Section 4.3.2
15 |   grandparent_queries, grandparents ← path_compression(elements)
   |    ▷Section 4.3.3
16 |   comm.isend(grandparent_queries)
17 |   comm.isend(grandparents)
18 |   transferred_edges ← edge_passing(elements, edges)  ▷Section 4.3.2
19 |   comm.isend(transferred_edges)
20 end
```

a finite number of elements and edges. Second, every message can be delivered, but the delivery order of messages is not guaranteed, for example, when MPI asynchronous communications are used. Third, the asynchronous termination detection can inform each process to exit iterations when all processes finish local work and no message is being transferred.

4.3.5.1 Converging in Finite Iterations

We explain our algorithm converges within a finite number of total iterations of processes, where the convergence is achieved when (1) all edges are consumed and (2) disjoint-set trees have no further changes.

(1) All input edges are consumed within finite total iterations.

Proof. At each iteration of a process, if a set root has edges connecting with smaller elements, at least one edge is consumed for set uniting; all other edges are passed to a new set root. When passing edges, any edge is either (1) deprecated because the endpoints already belong to the same set or (2) passed to a new set root within finite such passes because each disjoint set tree has no cycles, guaranteed by the uniting by ID rule, and contains finite elements. Each pass is either a local operation completed in one iteration or using a message delivered within finite iterations. Because set uniting and edge passing keep reducing the number of edges, which is finite, all input edges are consumed within finite total iterations. □

(2) Disjoint-set trees converge to trees with at most two layers within finite iterations, and no messages are exchanged after the convergence of disjoint-set trees.

Proof. In the loop of iterations, disjoint-set trees converge when all non-root elements point to (local/set) roots and all local roots point to set roots. Non-root elements point to local/set roots within finite local computations given that, at every iteration, non-root elements of a process will point to their local grandparents for path compression. Each local root points to a set root within a finite number of message deliveries, given that the local root points to its grandparent after one grandparent query process.

Because every message can be delivered within finite iterations, the local root points to a set root within finite iterations. Additionally, local roots do not send grandparent queries again after pointing to set roots because the boundary cases in Section 4.3.3 are considered, ensuring no messages are exchanged after disjoint-set trees converge and iterations can properly terminate when asynchronous termination detection is used. After the final local path compression, which is illustrated in Fig. 4.1f and forms one local iteration at each process, all elements point to set roots, leading to trees with at most two layers. □

4.3.5.2 Outputting Correct Disjoint-Set Trees

The correctness is guaranteed by (1) ensuring connected elements in input belong to the same sets in the output and (2) producing unique disjoint set trees; both are independent of the order of set unitings.

(1) Every two edge-connected elements in an input graph belong to the same disjoint set in output.

Proof. At the first iteration of each process, if an edge is used for a set uniting, then the two elements connected by the edge are united to the same set; every other edge of the input graph undergoes distributed edge passing. Although we cannot guarantee the order of edge passings, we have shown that any edge is consumed within finite iterations. If a passed edge is applied for a union operation, the original two elements of the edge are guaranteed to belong to the same set due to the tree topology. If the edge has never been considered for a union operation and is deprecated during the edge passing, the only reason is the original two endpoints of the edge have already

belonged to the same set before the edge is passed to set roots. Therefore, any two elements connected in an input graph belong to the same set in the output. □

(2) The output consists of ideal disjoint-set trees, where elements, within each set, point to the set's smallest element.

Proof. We have shown that, the disjoint-set trees have at most two layers after the convergence, and all elements within a set point to a set root, regardless of the execution order of union operations. Because the uniting by ID rule guarantees each disjoint-set tree's root to be the smallest element within each set, our algorithm can produce the ideal disjoint-set trees after convergence. □

Algorithm 2: balancing_elements(comm, elements, edges)

```
1  rank ← comm.rank() // ID of this process
2  nproc ← comm.size() // process count
3  ndim ← domain dimensionality
4  dim ← 0 // current splitting dimension
5  group ← {0, 1, ..., nproc−1} // all processes belong to the same group
     initially
6  for i ∈ [0, log2(nproc)) do
7  |   m ← select_median(comm, elements, group, dim)
8  |   i_bit ← 1 << i
9  |   group ← {x ∈ group | x AND i_bit = rank AND i_bit}
10 |   partner ← rank XOR i_bit
11 |   elements ← swap_elements(comm, partner, elements, m)
12 |   dim ← (dim+1) MOD ndim
13 end
14 edges ← redistribute_edges(comm, elements, edges)
15 return elements, edges
```

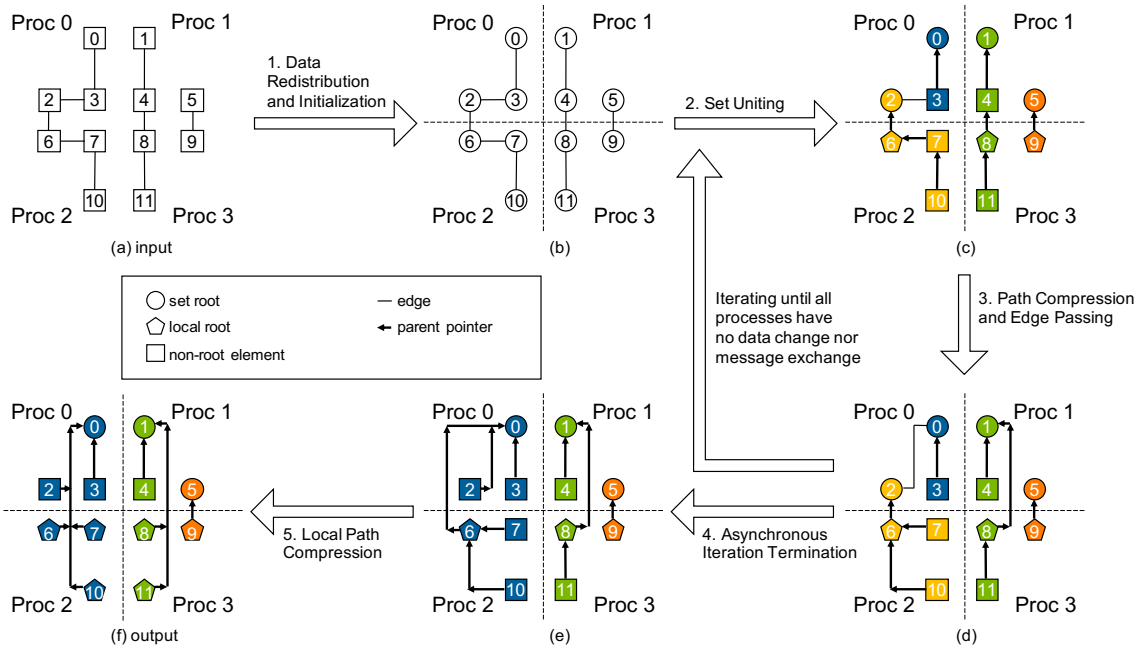


Figure 4.1: Examples of different operations. (a): An input graph consists of twelve elements. The IDs of the elements are from 0 to 11. (b): Elements and edges are redistributed evenly across processes with disjoint sets initialized. (c): Disjoint sets are united in parallel. Colors represent (temporary) disjoint sets. (d): Paths from elements to roots are compressed from (c) to (d), such as the path between 8 and 1 and the path between 10 and 6. Remaining edges are passed toward set roots; for example, an edge between 3 and 2 in (c) is passed to connect 0 and 2 in (d). Each process performs the path compression and edge passing independently without blocking other processes. (e): After processes terminate iterations asynchronously, correct disjoint sets are acquired. Certain non-root elements (e.g., 7, 10, and 11) point to local roots rather than set roots. (f): After all non-root elements point to set roots using a local path compression, three disjoint sets are produced.

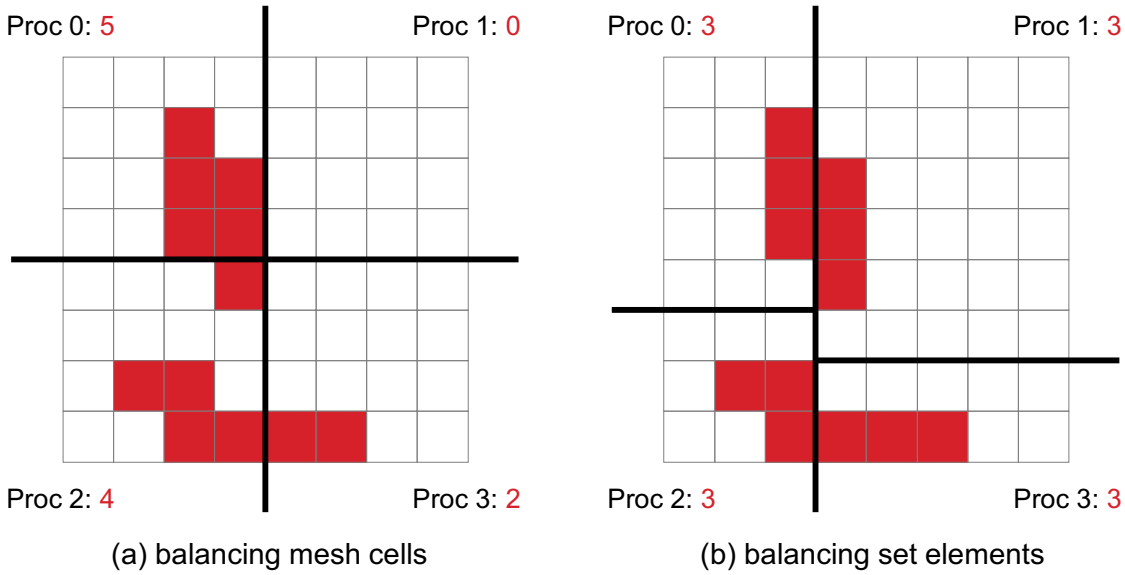


Figure 4.2: Two possible load balancing schemes for distributed union-find in scientific visualization and analysis. We color set elements by red, and the element counts assigned to processes are indicated in respective corners. (a): Balancing the number of mesh cells in each process using binary space partitioning [72]. However, balancing cells may not solve workload imbalance effectively. For example, Process 1 has zero elements, and hence has no work to do. (b): Balancing the number of elements in each process based on a k-d tree decomposition. In this example, each process is assigned three elements and attains a balanced workload.

4.4 Load Balancing Using K-D Tree based Element Redistribution

We redistribute elements to balance the number of elements in each process for workload balancing in distributed union-find, as illustrated in Fig. 4.2b. We follow the k-d tree space decomposition method proposed by Morozov and Peterka [112] for distributed computing environments. Modified pseudocode is presented in Algorithm 2 and described below to support the load balancing in distributed union-find.

Given distributed elements with numeric coordinates, the decomposition approach partitions the domain so that each process is assigned a partition that contains a similar number of elements. Initially, all processes belong to a single group, which represents the whole domain. The domain is then decomposed based on medians of the coordinates of elements. After a median is selected, the group of processes is split into two subgroups: the first subgroup contains all processes whose IDs' least significant bit is 0, and the second subgroup contains all processes whose IDs' least significant bit is 1. Elements are then exchanged between the two subgroups so that the first subgroup corresponds to the lower half of the domain and the second subgroup corresponds to the upper half of the domain. Each process in the first subgroup sends local elements that lie above the selected median to a partner process in the second subgroup; the partner process shares the same bits in the ID as the first group process except that the least significant bit is 1. Also, the partner process sends the first group process the elements below the selected median. In the following iterations, the domain decomposition happens within every subgroup based on the second, third, and so on significant bits of process IDs until each subgroup contains one process.

After domain decomposition, we redistribute edges such that every edge is stored in its larger endpoint's process.

4.5 Algorithm Evaluation

We evaluate distributed union-find algorithms under a scientific feature extraction and tracking framework consisting of four stages: (1) domain partitioning, (2) feature detection, (3) connected component labeling (CCL) using distributed union-find, and (4) finalization, following existing scientific studies in the Feature Tracking Kit (FTK) [63, 65].

First, we evenly partition the input spatiotemporal data domain into spacetime blocks, which are then distributed over the participating processes. The spacetime blocks are constructed using spacetime meshing [57, 159] to support detecting scientific features with time continuity [159] and capturing topological events [81, 159]. Each spacetime block includes one layer of ghost cells in both space and time dimensions to associate spatiotemporal features across blocks.

Second, each process independently detects features and connections between the features. For example, in super-level set extraction, we detect spacetime cells with values higher than a specified threshold. In critical point tracking, we follow Tricoche et al. [159] to detect critical points on the faces of spacetime cells: we estimate derivatives at mesh grid points using central difference and locate critical points on the faces using inverse interpolation. The connections exist between features in adjacent spacetime cells or faces.

Third, we perform CCL using a distributed union-find algorithm. CCL involves detecting the connectivity between features and labeling each connected component

by a unique identifier. We regard spatiotemporal features as elements in disjoint sets and connections between the features as edges between the elements. As a result of the distributed union-find algorithm, features within the same connected component are merged into a single set and labeled by a common identifier.

Fourth, we finalize feature extraction and tracking by gathering features within the same connected component to the same process for further visualization, analysis, and storage.

Our study below focuses on measuring the performance of CCL using distributed union-find algorithms.

Baseline: We implement the distributed union-find (DUF) method of Iverson et al. [80] load-balanced by balancing mesh cells [72] as the baseline for comparison. The study of [80] evaluates five state-of-the-art distributed approaches on the CCL task and indicates that, except the breadth-first search based label propagation exhibiting considerably worse results, the other four approaches, including the DUF, have similar strong scaling efficiency in each test data.

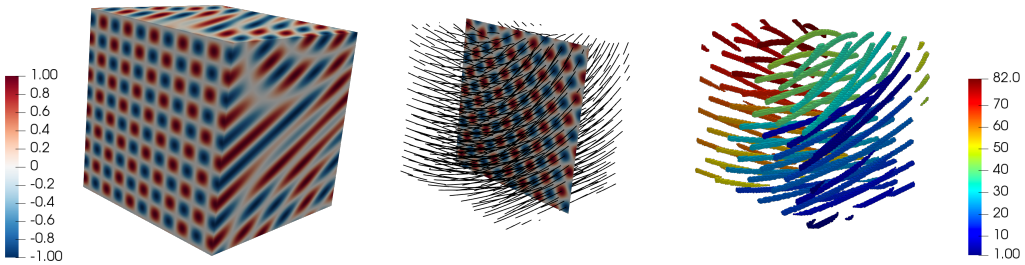
Compared with the baseline, the improvement caused by overlapping communications with computations using asynchronous parallelism is demonstrated on synthetic data (Section 4.5.1). The benefit of redistributing elements for load balancing is highlighted in both experimental and simulation datasets due to which have non-uniformly distributed features (Section 4.5.2).

Computing platform: We run experiments on an HPC cluster, which has 664 compute nodes. Every compute node has Intel Xeon E5-2695v4 CPUs with 32 cores and 128 GB memory. The HPC cluster uses an Intel Omni-Path interconnect network.

Message passing is supported by the Intel MPI library. The file storage system of the cluster is IBM General Parallel File System.

4.5.1 Benchmark on Synthetic Data

We begin by measuring the influence of overlapping communications and computations using asynchronous parallelism with asynchronous communications and asynchronous termination detection for distributed union-find. In the experiments, we track and extract two types of features, including (1) critical points (local maxima, local minima, and saddle points) and (2) super-level sets, on synthetic data. Synthesizing data allows us to control data resolution and the number of features to support different evaluations. An example of the synthetic data we use is visualized in Fig. 4.3.



(a) (b) (c)

Figure 4.3: Visualizations of a synthetic case. (a): A synthetic volume with scalar values ranging from -1 to 1. We track critical points and extract super-level sets on the same synthetic volume. (b): Trajectories of the critical points represented by black lines. (c): Super-level sets with a threshold of 0.8. 82 connected components are labeled, and each is assigned a unique hue.

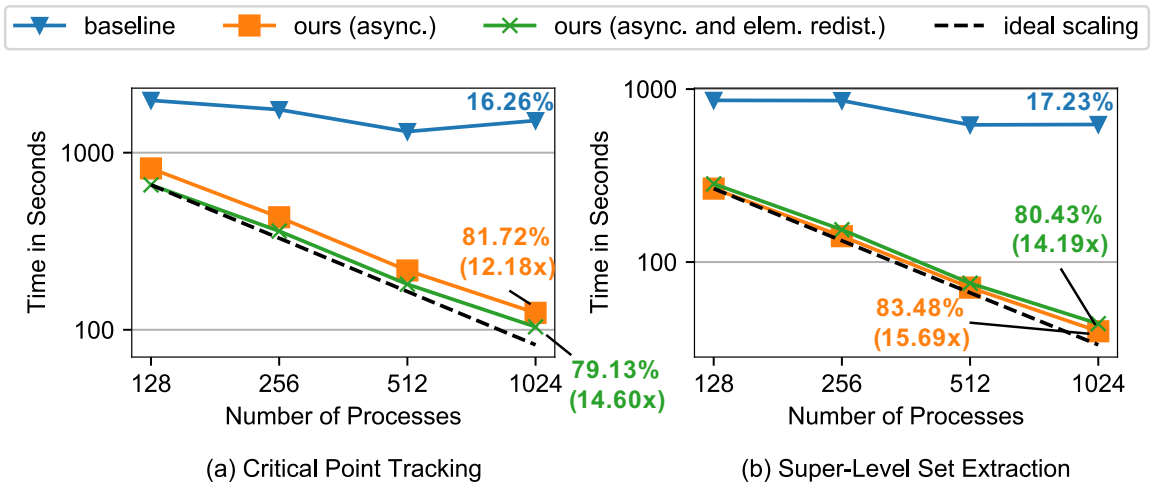


Figure 4.4: Strong scaling of distributed union-find on $1,024^3$ synthetic data using 128 to $1,024$ processes for (a) tracking critical points and (b) extracting super-level sets. Both axes are log scales. The baseline is the distributed union-find (DUF) of Iverson et al. [80] with balanced mesh cells [72]. Our methods consist of the distributed asynchronous (async.) union-find without/with the k-d tree based element redistribution (elem. redist.).

4.5.1.1 Strong Scalability Study

We conduct a strong scaling experiment using synthetic data with a $1,024^3$ resolution, which has $161,338,942$ critical points and $72,097,212$ voxels with values larger than 0.8.

Compared with the baseline using the bulk-synchronous parallelism, the use of asynchronous parallelism leads to significant improvement in strong scaling efficiency. Strong scaling results using up to $1,024$ processes are shown in Fig. 4.4. The baseline attains a strong scaling efficiency of 16.26% in the critical point tracking benchmark

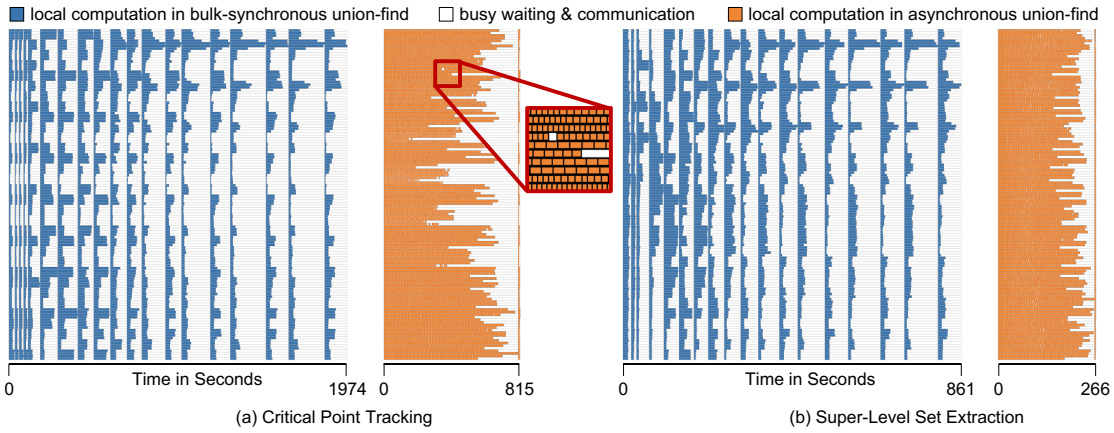


Figure 4.5: Gantt charts of bulk-synchronous baseline [80] and our asynchronous algorithm using $1,024^3$ synthetic data distributed among 128 processes. The horizontal axis encodes time. Each row corresponds to a process.

and 17.23% in the super-level set extraction benchmark. Our algorithm with the asynchronous parallelism attains 81.72% and 83.48% for the two benchmarks, respectively, with a speedup of 12.18x and 15.69x over the baseline.

To investigate why overlapping communications and computations using asynchronous parallelism is better than the distributed bulk-synchronous parallelism in detail, we list processes' computation time at each iteration in Fig. 4.5. In the distributed bulk-synchronous baseline, a global synchronization is performed at the end of each iteration to merge and update sets across the processes and detect iterations' termination. Due to the use of global synchronizations, processes with less computational work become busy-waiting for processes with more work at each iteration round. After overlapping communications and computations using asynchronous communications and asynchronous termination detection, each process performs computations at their own pace without being blocked by other processes, leading to reduced waiting time

and improved computational resource usage. After the end of the iterations, our algorithm performs an additional local path compression, however, which occupies only a small portion of the total execution time.

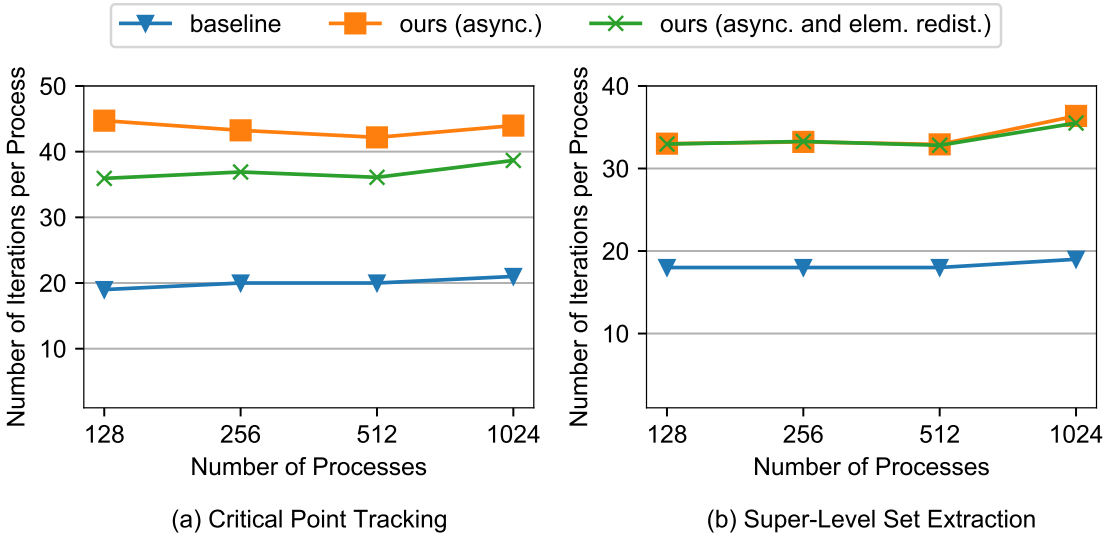


Figure 4.6: Iteration count per process of distributed union-find on $1,024^3$ synthetic data. The horizontal axis is a log scale.

We also evaluate the influence of using asynchronous parallelism on the number of iterations per process. Results in Fig. 4.6 illustrate that our distributed asynchronous approach has approximately doubled iterations compared with the bulk-synchronous baseline in both the critical point tracking and super-level set extraction. Because asynchronous communications are nonblocking, processes can iterate without waiting for other processes. As long as new messages come, processes can immediately come to the next iteration and handle new work. In contrast, the bulk-synchronous parallelism synchronizes processes between iterations. Hence, each process may receive

work from all other processes after a synchronization, leading to more work to do during the next iteration and requiring fewer iterations. Although using asynchronous parallelism results in more iterations, our algorithm's total execution time is less than the bulk-synchronous baseline, as seen in Fig. 4.4.

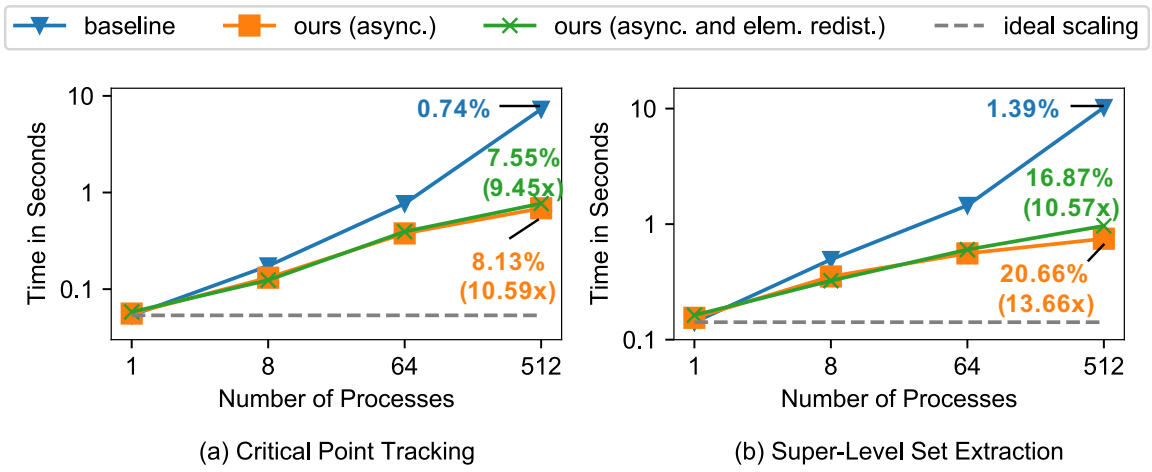


Figure 4.7: Weak scaling of distributed union-find on synthetic data. We use four combinations of data resolutions and process counts: 32^3 with 1 process, 64^3 with 8 processes, 128^3 with 64 processes, and 256^3 with 512 processes. Both axes are log scales.

4.5.1.2 Weak Scalability Study

We measure the influence of using asynchronous parallelism on weak scaling of distributed union-find. The weak scaling evaluates the performance when each process is assigned a constant-size problem as the number of processes increases. In this study, each process is assigned with a 32^3 mesh grid with a constant feature density. Test data with sizes of 32^3, 64^3, 128^3, and 256^3 are processed using 1, 8, 64, and 512

processes respectively. The ideal weak scaling is when the execution time is constant in all the runs.

Compared with the baseline using bulk-synchronous parallelism, the asynchronous parallelism leads to significant enhancement in weak scaling efficiency. The results are displayed in Fig. 4.7. When using 512 processes, the baseline achieves weak scaling efficiency of 0.74% in the critical point tracking benchmark and 1.39% in the super-level set extraction benchmark. Our distributed union-find with the asynchronous parallelism attains 8.13% and 20.66% efficiency in the two benchmarks, respectively, with a speedup of 10.59x and 13.66x over the baseline.

4.5.2 Scientific Applications

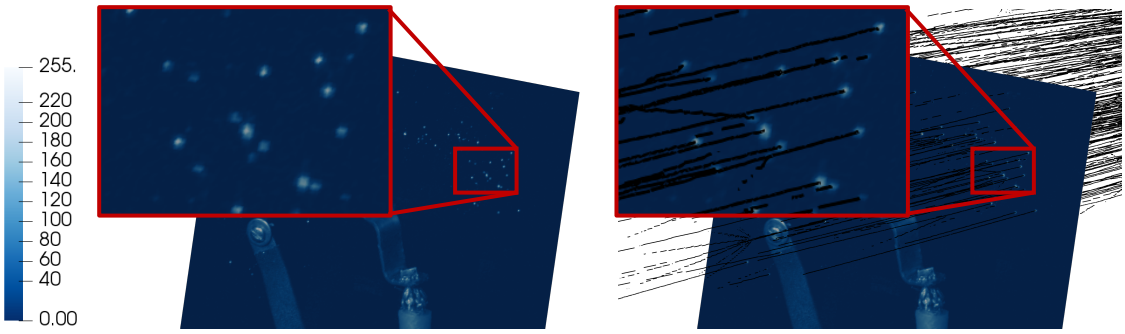
We evaluate distributed union-find, especially the load balancing performance, in two scientific applications: (1) tracking critical points in exploding wire experimental data and (2) extracting super-level sets in fusion plasma simulation data. Both of the scientific data are time-varying.

4.5.2.1 Application Background and Benchmark Setting

We give background and benchmark settings for the two scientific applications.

Exploding wire experiments: Scientists can use an exploding-wire apparatus to generate many high-temperature microparticles [165, 166]. High-speed imaging cameras can capture the movement of these particles and produce high-resolution images. Tracking these particles on the images helps physicists understand the particles' physical properties and helps computational scientists enhance theoretical models for simulation development.

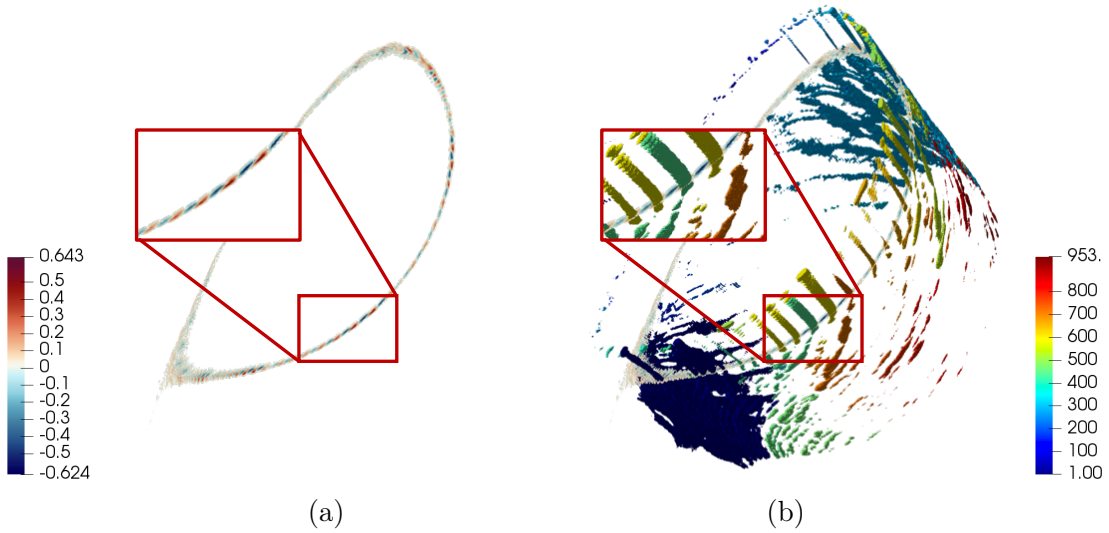
We model particles as local maximum points in each frame and track the points' movement across frames of the exploding wire imaging data. A frame of the time-varying data is shown in Fig. 4.8a. The test data have a 384×384 spatial resolution and $4,745$ timesteps. $3,197,333$ maximum points are detected from the data. The maximum points' trajectories are shown in Fig. 4.8b, which pass through the particles on the image.



(a) (b)

Figure 4.8: Tracking critical points in the exploding wire experimental data. The intensity value ranges from 0 to 255. (a): One image frame, where bright particles are detected as maximum points. (b): Trajectories of maximum points represented by black lines.

Fusion plasma simulations: A Tokamak torus device is the mainstream fusion reactor to confine plasma magnetically in order to achieve fusion energy production magnetically. The turbulent transport from the edge plasma usually takes a ubiquitous form of filaments, defined as density-enhancement coherent structures, also referred to as blobs. The blob movement may cause loss of plasma and severe damage to the device, and hence, has been subject to intensive researches [87, 120, 141].



(a) (b)

Figure 4.9: Tracking super-level sets in fusion plasma simulation data. (a): 2D density field at a timestep, where blobs are high-density regions and are detected as super-level sets. (b): Extracted super-level sets having 953 labeled connected components colored by unique hues.

We track blobs in ion density fluctuation data produced by a BOUT++ electromagnetic fluid simulation [46, 174] for the fusion reactor. A separatrix slice of the data is shown in Fig. 4.9a. Blobs usually are the regions of high ion density. Hence, following the work of [120], we model blobs as regions with densities larger than 2.5 standard deviation than the average density and track super-level sets across timesteps. The test data have 425×880 spatial resolution and 701 timesteps. $1,708,341$ high-density voxels are extracted from the data. The results are shown in Fig. 4.9b.

4.5.2.2 Benchmark in Scientific Applications

We demonstrate the performance of distributed union-find in the two scientific applications. Fig. 4.10 displays the strong scaling results. When $1,024$ processes are used, the bulk-synchronous baseline approach achieves strong scaling efficiency

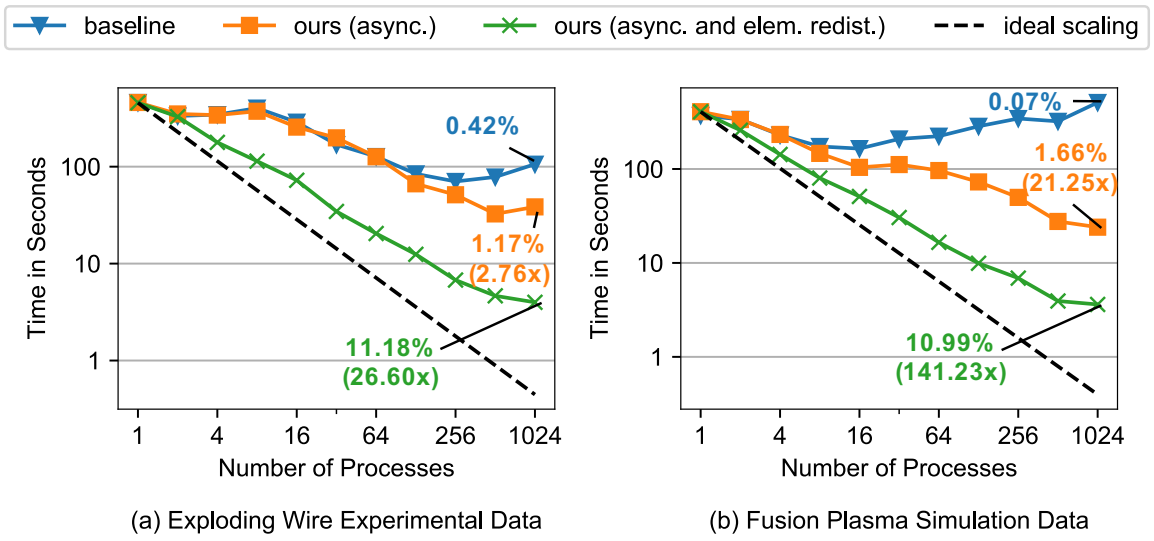


Figure 4.10: Strong scaling of distributed union-find for tracking and extracting features in two application datasets: (a) exploding wire experimental data and (b) fusion plasma simulation data. Both axes are log scales. We compare a baseline (distributed union-find of Iverson et al. [80] with balanced mesh cells [72]) with our distributed asynchronous union-find without/with the redistribution of feature elements.

of 0.42% in the exploding wire data and 0.07% in the fusion plasma data. The asynchronous parallelism improves the efficiency to 1.17% and 1.66% with a speedup of 2.76x and 21.25x over the baseline, respectively. However, the scalability is still limited by the imbalanced features.

Evaluation of load balancing: We evaluate the load balancing of distributed union-find using the k-d tree based element redistribution in the two scientific applications. As visualized in Fig. 4.8 and Fig. 4.9, the features are not uniformly distributed in domain for both of the scientific applications. Hence, balancing mesh cells may not be effective enough for these cases, and balancing feature elements is expected to improve the distributed union-find performance. After redistributing feature elements

to balance the number of features in each process, the strong scaling efficiency increases significantly to 11.18% and 10.99%, respectively. Compared with the baseline, our distributed asynchronous union-find with the feature element redistribution attains 26.60x speedup for tracking critical points in the exploding wire data and 141.23x speedup for tracking super-level sets in the fusion plasma data. Fig. 4.11 displays the cost breakdown of our distributed asynchronous union-find with the feature element redistribution.

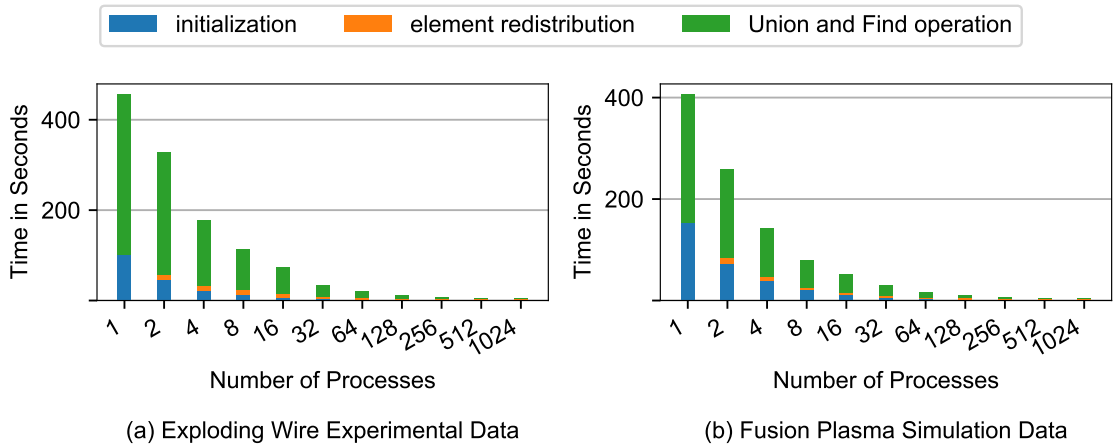


Figure 4.11: Breakdown of the time cost of our distributed union-find algorithm with both the asynchronous parallelism and the element redistribution in two application datasets. The horizontal axis is a log scale. The "initialization" includes the initial assignment of element IDs and the initialization of data structures.

4.6 Discussions

We discuss limitations of our distributed union-find.

Limitation of extreme pathological cases: The performance of our k-d tree based element redistribution may be reduced in two pathological cases.

First, when connected components lie on data block boundaries, our k-d tree based element redistribution decomposes domain into axis-aligned blocks, leading to high communication costs when snake-shaped connected components run precisely down processes' block boundaries. Although such a case does not appear in our test data, this case may happen in, for example, helix protein data. In the future, we will explore other load-balancing (e.g., graph-partitioning-based) schemes to address this pathological case.

Second, in extreme cases, the k-d tree decomposition may cause a process that has a large number of neighboring processes, leading to an increase in communication costs. To control the number of neighboring processes, in the future, we may explore a decomposition based on quadtrees or octrees.

Limitation of element identifier: Our distributed asynchronous union-find assumes elements have unique and sortable IDs, which may not be the case for all datasets. In scientific applications using mesh data, we use IDs of mesh cells or mesh faces as elements' IDs, where the cost of such ID assignment is included in "initialization" of Fig. 4.11. However, for a graph structure without element IDs as input, an additional preprocessing for element ID assignment is required. A possible way to remedy this could be to collect element counts of all processes, compute a numeric ID range for each process, and each process assigns IDs to local elements in parallel within the ID range, which introduces additional cost.

Limitation of memory capacity: Out-of-core algorithms may be needed if the memory capacity cannot hold a single data block. A possible solution to remedy the limitation is described in the following. First, we decompose the data further into smaller data blocks with ghost layers such that the memory of each process

is able to hold a single data block. Each process then loads a smaller data block. Second, each process extracts elements and edges of interest in the loaded data block to perform distributed union-find. Third, processes release current blocks and load new unprocessed blocks. We repeat the second and the third step until all data blocks are processed.

4.7 Appendix: Additional Algorithm Evaluations

4.7.1 Effect of Varying Sizes of Data Blocks

We include additional studies, which show that the performance improvements (i.e., speedups) of our algorithm over the baseline are consistent with varying sizes of data blocks in both strong and weak scaling studies.

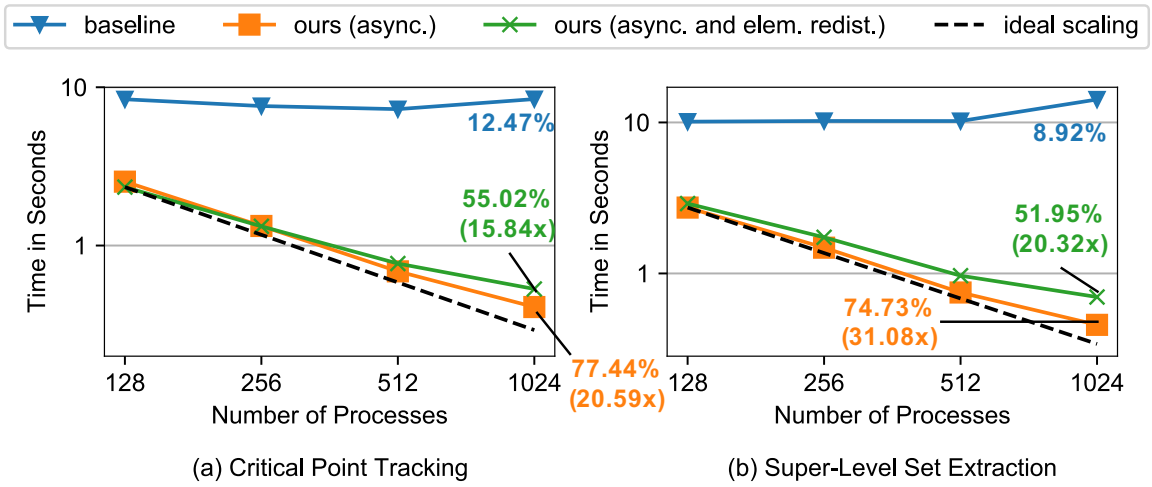


Figure 4.12: Strong scaling of distributed union-find on 256^3 synthetic data using 128 to $1,024$ processes for (a) tracking critical points and (b) extracting super-level sets. Both axes are log scales.

4.7.1.1 Strong Scaling Study

With respect to the strong scaling, our algorithms have consistent performance improvements (i.e., speedups) over the baseline when the sizes of data blocks vary by comparing Fig. 4.4 using $1,024^3$ data and Fig. 4.12 using 256^3 data.

We included an additional strong scaling study for 256^3 synthetic data in Fig. 4.12. Compared with the baseline, our asynchronous algorithm *without* the element redistribution attains 20.59x speedup in the critical point tracking benchmark and 31.08x speedup in the super-level set extraction benchmark when $1,024$ processes are used; our asynchronous algorithm *with* the element redistribution attains 15.84x and 20.32x speedup in the two benchmarks.

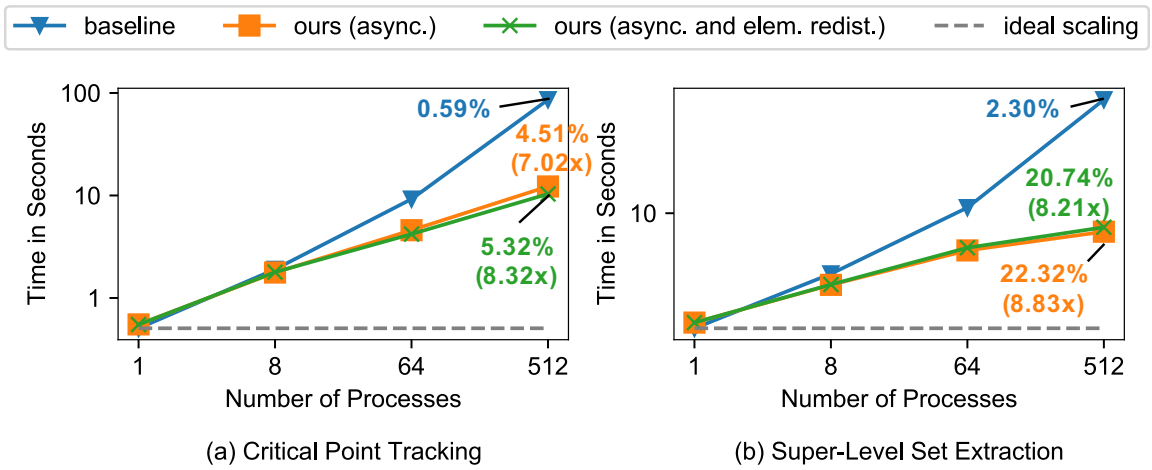


Figure 4.13: Weak scaling of distributed union-find on synthetic data. Each process is assigned with a 64^3 mesh grid with a constant feature density. We use four combinations of data resolutions and process counts: 64^3 with 1 process, 128^3 with 8 processes, 256^3 with 64 processes, and 512^3 with 512 processes. Both axes are log scales.

4.7.1.2 Weak Scaling Study

With respect to the weak scaling, our algorithms also obtain consistent performance improvements over the baseline when varying the sizes of data blocks by comparing Fig. 4.12 using 64^3 mesh grid per process and Fig. 4.7 using 32^3 grid per process.

We included an additional weak scaling study with a 64^3 grid per process in Fig. 4.13. Compared with the baseline, our asynchronous algorithm *without* the element redistribution attains 7.02x speedup in the critical point tracking benchmark and 8.83x speedup in the super-level set extraction benchmark when 512 processes are used; our asynchronous algorithm *with* the element redistribution attains 8.32x and 8.21x speedup in the two benchmarks.

4.7.2 Effect of Varying Feature Densities

We evaluate the performance of the baseline approach and our distributed union-find algorithms when the feature density of the data varies in space and time. Results indicate the performance improvements of our algorithm over the baseline are consistent with varying feature densities.

We test the methods on $1,024^3$ synthetic data with three levels of feature density: (1) low feature density, (2) medium feature density, and (3) high feature density; the data are generated by fixing the data resolution to be $1,024^3$ and varying the number of features in space and time. For tracking critical points, the high feature density case has $161,338,942$ critical points in space and time, the medium-density one has $40,280,575$ critical points, and the low-density one has $10,066,949$ critical points. For tracking super-level sets, the high feature density case has $72,097,212$ extracted

voxels, the medium-density one has $35,071,247$ voxels, and the low-density one has $17,309,120$ voxels.

The results are shown in Fig. 4.14. For critical point tracking, when using $1,024$ processes, compared with the baseline, our asynchronous algorithm *without* the element redistribution achieves 12.18x, 12.89x, and 14.33x speedup on the data with the three levels of feature density; our asynchronous algorithm *with* the element redistribution achieves 14.60x, 14.34x, and 15.53x speedup, respectively. For super-level set tracking, when using $1,024$ processes, compared with the baseline, our asynchronous algorithm *without* the element redistribution achieves 15.69x, 16.70x, and 15.66x speedup on the data of low, medium, and high feature density; our asynchronous algorithm *with* the element redistribution achieves 14.19x, 14.36x, and 15.06x speedup, respectively.

4.7.3 Cost Breakdown of Feature Extraction and Tracking Framework

We included a cost breakdown of different stages of the feature extraction and tracking framework with respect to different counts of processes in Fig. 4.15. For the critical point tracking in exploding wire experimental data, the majority of the cost is the critical point detection. For the super-level set extraction in fusion plasm simulation data, as compared with other steps, the time of the distributed union-find based CCL shrinks as we use more processes, which shows that our distributed union-find algorithm has a good scaling.

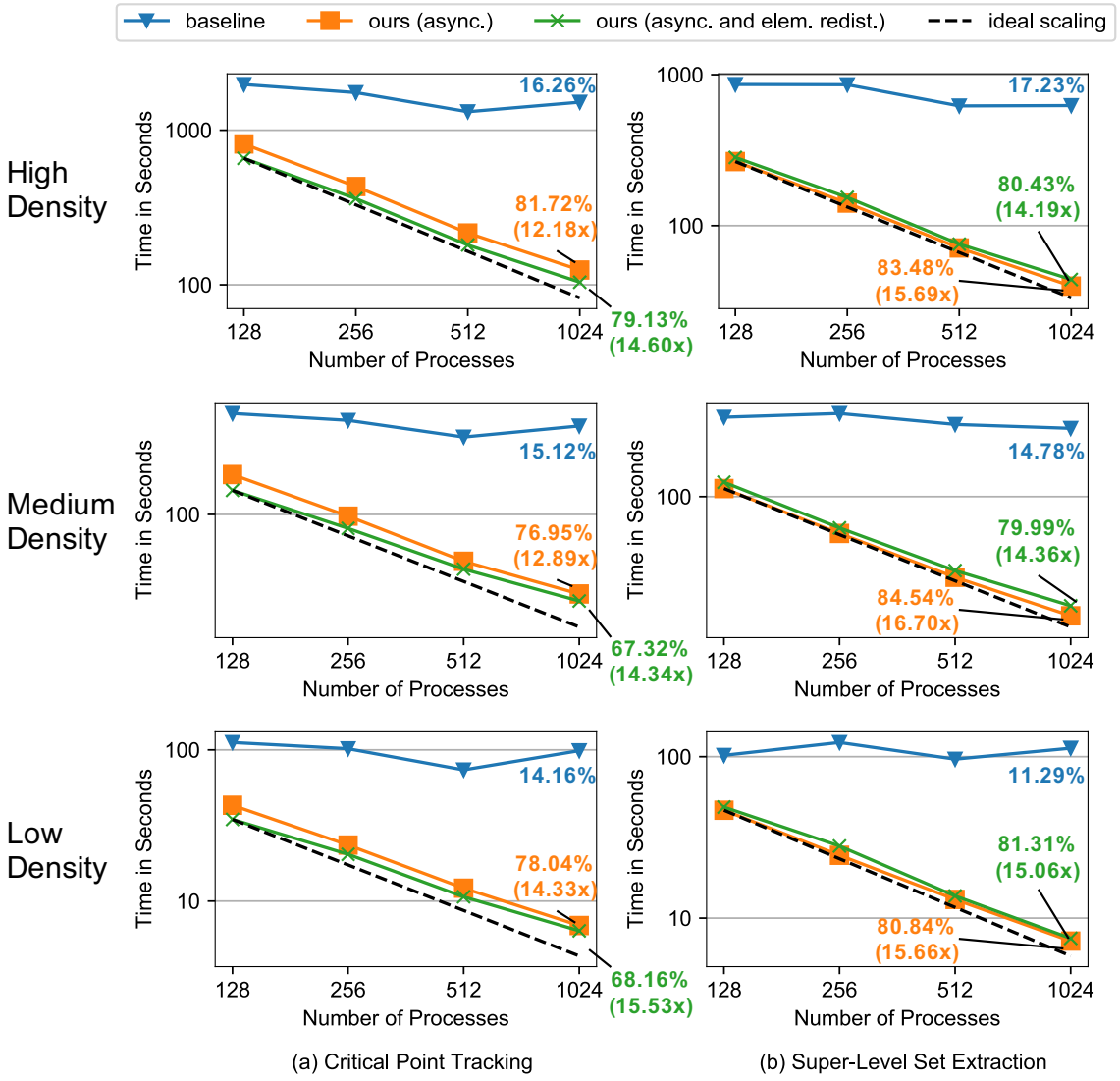


Figure 4.14: Strong scaling on $1,024^3$ synthetic data with three feature density levels: high, medium, and low density in three rows for (a) tracking critical points and (b) tracking super-level sets in two columns.

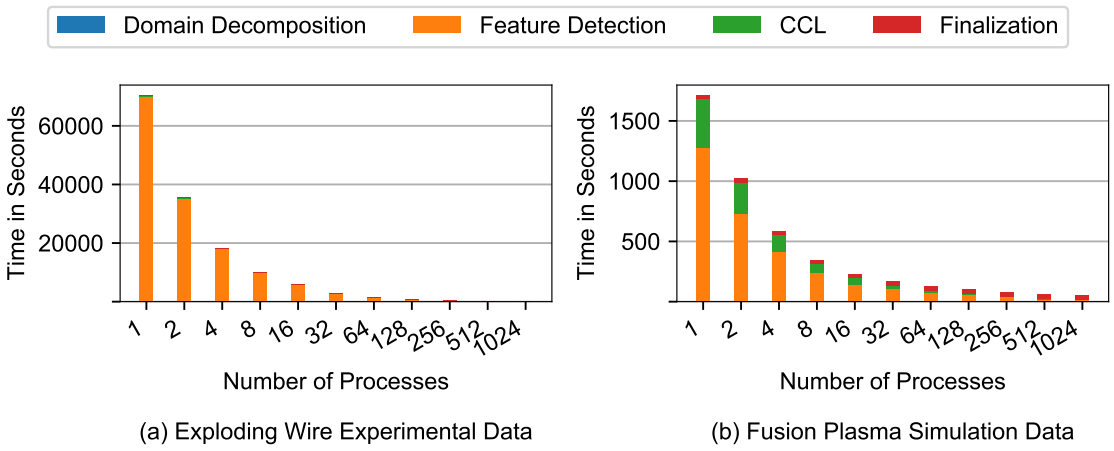


Figure 4.15: Breakdown of the time cost of different stages of the used feature extraction and tracking framework in two application datasets. The horizontal axis is a log scale.

Chapter 5: Reinforcement Learning for Load-balanced Parallel Particle Tracing

As the size and complexity of vector-field data increase, distributed and parallel particle tracing becomes essential for visualizing and analyzing large-scale data from scientific simulations. For example, distributed texture-based visualizations, such as line integral convolution (LIC) [23, 118] and finite-time Lyapunov exponents (FTLEs) [70, 124, 178], benefit from distributed particle advection for efficient computation of streamlines and pathlines. Other applications of distributed and parallel particle tracing also play essential roles in scientific data analysis, such as distributed streamsurface computation [97] and source-destination queries [69, 84, 178], just to name a few.

According to existing studies [131, 134], the scalability and performance of distributed particle tracing are highly dependent on two aspects: (1) the workload balance of parallel processes and (2) the cost of communications. First, the workload of processes in distributed and parallel particle tracing can be imbalanced. Uneven distributions of complex flow features (e.g., critical points and vortices) in space usually lead to uneven distributions of particle positions during the tracing. Second, the cost of interprocess communications can be high due to the exchange of particles or data blocks. For example, the circular flow patterns in input vector field data usually

lead to frequent particle transfer among data blocks, causing a high communication overhead.

This work is motivated by the need to simultaneously optimize load-balancing and minimize communication for the purpose of reducing parallel execution time for particle tracing. Two challenges exist to build such an optimization model and to optimize online performance with minimal overhead. First, such a model must be able to balance workloads as much as possible while avoiding frequent and unnecessary data movements. Second, the optimization must be achieved in real-time so as not to slow down parallel particle tracing. To the best of our knowledge, solving these two challenges is still an open problem.

We perform simultaneous optimization of workload balance and communication efficiency based on reinforcement learning (RL) studies. The optimization requires distributing data blocks and particles among processes to maximize the workload balance and minimize the communication overhead, which can be categorized into the integer programming problem and is NP-Complete [56, 58]. Additionally, the particle positions continue to change during the parallel execution, leading to volatile information for optimization decisions; hence classic methods such as dynamic programming, are not effective. RL approaches are developed to create agents to adapt decision-making with learning from the dynamic environments and maximize reward functions [153]. Reward and cost functions are designed in this paper to incorporate both workload balance and communication costs.

To address the aforementioned challenges, we introduce three models that work hand in hand to enable online performance optimization for parallel particle tracing in distributed-memory systems: (1) dynamic work donation model, (2) workload

estimation model, and (3) communication cost model. First, we propose an RL-based workload donation model to allow processes to balance their workloads periodically. We associate an RL agent to each process. An agent is trained and used to move work from processes with more workload to those with less workload. Rewards guide the agents' behaviors, and are designed according to the distributions of workloads and cost of data transfer in order to create balanced workloads among processes with minimized costs. Second, we design an online and high-order workload estimate model to estimate blockwise workload in the units of integration steps given the incoming particles based on historical data that is recorded during the run time. The model learns the historical data from high-order data access patterns of particles, and dynamically adapts to different flow behaviors. Third, we construct a communication cost model to estimate the data transfer time of both blocks and particles based on the historical data transfer since the beginning of the run, hence, allowing the model to adapt to the available network bandwidth. The cost model is constructed based on a linear transmission model [30, 85, 158] that models the cost to be a constant latency plus time proportional to the data size.

To manage and optimize the activities across processes, we orchestrate the pipeline of workload-balanced parallel particle tracing by taking the communication cost into account. First, we decompose the entire input vector-field domain into data blocks, which are then assigned to the parallel processes. We seed particles in the data blocks and trace the particles in the participating processes. Second, when particle tracing runs in parallel, processes gather statistics on workloads and communication time and pass them to processes' RL agents. Third, agents of processes move data blocks and particles among processes to balance the workload with minimized data transfer costs.

After agents take action, feedback rewards considering workload distribution and data transfer cost are computed to improve agents' decision-making ability for future decisions. We tailor a policy-gradient based reinforcement learning algorithm, which can train the agents and make the block assignment decision efficiently. Fourth, the processes continue to collect statistical information of data transfer and establish the communication cost model to estimate the transfer cost of data blocks and particles in the subsequent processing rounds for agents to make accurate decisions.

We evaluate our method with applications from fluid dynamics, ocean, and weather. We run our prototype implementation on a supercomputer with up to $16,384$ processors. Our method outperforms the state-of-the-art work stealing/requesting in terms of parallel efficiency, load balance, and the cost of I/O and communications. The contributions of this paper are threefold:

1. We propose a reinforcement learning based work donation model for distributed-memory parallel particle tracing that can optimize block assignment for load balancing and dynamically adapt to flow behaviors and available network bandwidth.

2. We design a high-order workload estimation model to predict the blockwise advection workload of particles.

3. We introduce the use of a linear transmission model to estimate the costs of interprocess communications.

This work is available in arXiv [172].

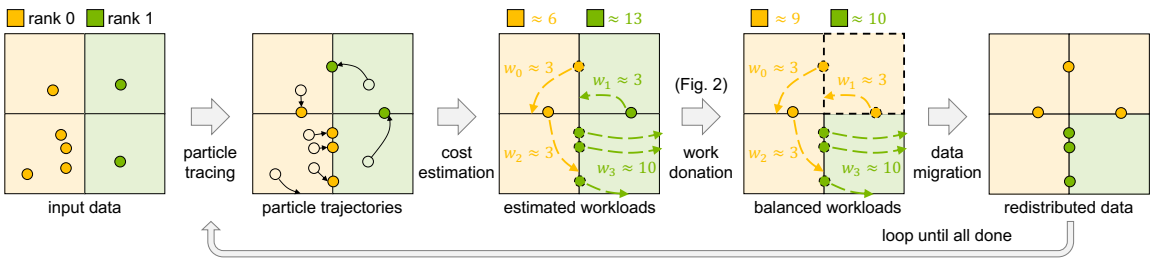


Figure 5.1: A schematic diagram of our RL based load-balanced parallel particle tracing. Two processes with rank 0 and 1 are colored in ■ and ■, respectively. We use w_i to indicate block i's estimated advection workloads in seconds. The two processes' estimated advection workloads are labeled at the top, which are the sums of the owned blocks' workloads.

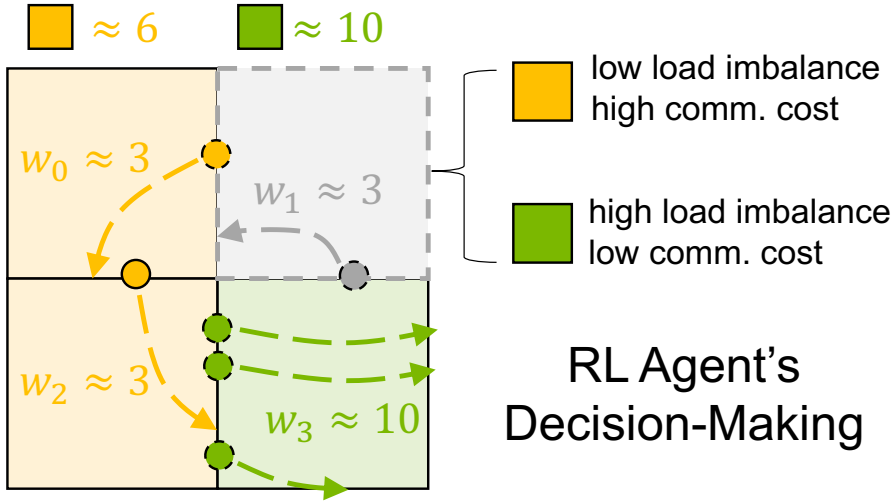


Figure 5.2: An illustration of an agent's decision-making in Fig. 5.1. The agent is deciding to assign the block colored by gray to which process.

5.1 Algorithm Overview

The goal of our method is to adjust the block assignment among processes to minimize the total execution time of particle tracing.

5.1.1 Optimization Problem Statement

The minimization of execution time can be abstracted into two components: the particle advection time and the communication time. We seek to acquire an assignment of data blocks, $\mathscr{B}_t$ dependent on execution time t, such that to minimize the future advection and communication time after time t,

$$_{\mathscr{B}_t}(T_{\text{advection}} + T_{\text{communication}}), \tag{5.1}$$

where $\mathscr{B}_t = (B_{t,0}, B_{t,1}, ..., B_{t,n_p-1})$. n_p denotes the number of processes, and $B_{t,l}$ denotes the block set assigned to the l-th process at time t. The block sets satisfy that $B_{t,0} \cup B_{t,1} \cup ... \cup B_{t,n_p-1} = B$, where B being the entire set of data blocks, and $B_{t,l} \cap B_{t,l'} = \emptyset$ for $l \neq l'$. We consider decomposing the objective into two requirements:

R1 Minimizing advection time: The advection time is bounded by the process with the highest computation time. When other processes complete their work, they become idle and wait for the process with the most computation to complete. We can balance the workload of processes to minimize the workload of the process with the maximal computation time to speed up the execution, where the *advection workload of a process* is defined to be the time of tracing particles in its blocks.

R2 Minimizing communication cost: We minimize the communication time for data transfer to reduce the total execution time.

R2.1 Minimizing block transfer: As exchanging data blocks across the processes to balance the workload incurs an additional cost, we seek to minimize the data block transfer cost.

R2.2 Minimizing particle transfer: As particles leave their current blocks and enter the next blocks, if the new blocks reside on different processes, the particles have to be transferred, which incurs the cost that we seek to minimize.

This optimization problem can be categorized as integer programming, which is NP-Complete [56, 58]. Hence, the solution space is too huge to be solved by a polynomial-time algorithm. We use an RL-based optimization algorithm to approximate the optimal solution for the dynamic load balancing.

5.1.2 Algorithm Pipeline

Our method builds on top of a data-parallel particle tracing pipeline as illustrated in Fig. 5.1. The input to the pipeline includes the vector field data and a set of particle seeds. The output of the pipeline is the trajectories of the traced particles.

Traditional load-balanced parallel particle tracing pipeline: The pipeline has three stages: (1) initialization, (2) blockwise particle tracing computation, and (3) termination detection. First, during the initialization stage, we distribute the input data and seeds among processes. We decompose the input vector field domain into axis-aligned data blocks with similar sizes where the number of the data blocks is higher than the number of processes; specifically for unsteady flow, a time-dependent vector field is decomposed into spacetime data blocks following previous works [31,78,178–180], with time being considered as an additional dimension to space as [57, 159]. We

then assign the blocks to the parallel processes where processes are assigned the same numbers of data blocks using the static round-robin assignment [131]. We distribute the input seeds of particles into the corresponding processes based on the distribution of blocks.

In the second stage, we perform blockwise particle tracing. For every *round* of particle tracing, we trace particles within the data blocks until all particles either stop prematurely because of hitting critical points in steady flows or go out of their block boundaries. At this point, blocks are exchanged if necessary. Then the particles are sent to the next blocks that they are entering before a new round of particle tracing is performed.

Third, we repeat the tracing and workload balancing and terminate the parallel program until all the particles go out of the global domain boundary or exceed the maximum advection steps.

In the following, we introduce our dynamic load balancing scheme into the above pipeline that runs after each iteration of blockwise particle tracing.

Pipeline of our dynamic work donation algorithm: We target developing a dynamic work donation algorithm in distributed-memory systems to minimize the total execution time of parallel particle tracing. The input to the our algorithm is the particle and data block distribution among processes. The output is the redistribution of particles and data blocks to have a more balanced workload and lower execution time.

First, we estimate processes' workloads and communication costs. A workload estimation model is constructed to estimate the workload of data blocks for the next round of computation. When particles go out of their current blocks' bounds, processes

first exchange the counts of particles that will be transferred to the other processes that have adjacent blocks. Then, we use the counts of incoming new particles to each block to estimate the workload at the next round. The method is described in Section 5.3. A communication cost model is constructed to estimate the costs of transferring a data block and transferring a particle, described in Section 5.4.

Second, an RL-based work donation optimizer is built to learn on-the-fly to optimize the workload balance of processes for the subsequent round of computations with minimized communication costs. Processes with high estimated workloads (donors) donate data blocks to processes with low workloads (receivers) demonstrated as below. The agent of process with a rank l uniformly samples a block b_i from local blocks with non-zero workloads. In our algorithm, we consider moving b_i to process l's friends; friends of process l are processes' ranks with a one-bit difference with l following the lifeline technique [17, 142]. We then sample an action from the current policy to determine whether a block should be donated to a underloaded process or not. Donors send donation requests to donation receivers. The donation receivers first assume all donations would be accepted, and send feedback with their current workloads plus all requested donations for donors to train their agents' policy; if donors' donations would make receives overloaded, corresponding actions are penalized in agents' training based on the feedback. If too many donation requests come simultaneously, the donation receivers then can choose to reject certain donation requests, ensuring (1) the receivers' workloads would not be higher than the donors' after taking donations and (2) the receivers' local memory capacity is sufficient to hold the donations.

Third, donors redistribute data blocks if the donation requests are accepted. After the donation receivers receive the new data blocks, particles are transferred to their

next corresponding processes and start another round of particle tracing. We describe details of the dynamic workload donation algorithm in Section 5.2.

Cold start problem: To prevent the cold start of the estimation models, each process places ten percent seeds at the first round of the parallel tracing. The rest of seeds will be released at the second round after the particles seeded at the first round leaves their seeding blocks. Hence, the particles seeded at the second round can benefit from the historical records saved from the first-round computation.

5.2 Reinforcement Learning based Dynamic Work Donation Model

We propose a reinforcement learning based work donation algorithm, which features (1) balancing processes' workloads with minimized communication costs and (2) proactively instructing overloaded processes to donate workloads to underloaded processes before underloaded processes become idle to minimize idle time. Because the actual workloads of processes are unknown before the particles are advected, workload estimates are computed and then used as the input to our algorithm before the advection occurs, following existing works [97, 117, 125, 131, 134, 180]. Agents monitor estimated workloads distributions among processes, and are trained by taking rewards describing load imbalance improvement and execution cost reduction after making donations. After the training, agents then learn policy strategies to maximize gained rewards to improve load imbalance among processes and minimize execution time.

We explain why agents need to learn strategies for work donations. The straightforward strategy, overloaded processes always donating work to processes with the lowest workload, is highly possible to make donation receivers become overloaded after

the donations are made. Also, it is challenging to consider the effect of communication costs when donations are planned. To pursue a proper strategy for the adjustment of work, an agent is created for each process to learn and adapt its action according to rewards obtained from historical donations.

5.2.1 Policy Gradient based Reinforcement Learning

We create a multi-agent reinforcement learning based optimizer to maximize reward functions.

Preliminary: We first give the background of policy gradient based reinforcement learning, which usually converge fast in applications [153]. The policy (i.e., probabilities of actions) of agents are updated iteratively guided by reward and cost functions to find an optimal decision-making strategy.

The policy-gradient-based methods use a Markov Decision Process (MDP) model, which can be formulated by a tuple $(S, A, \pi_{\boldsymbol{\theta}}, R)$, where S is a finite set of states, A is a finite set of actions, $\pi_{\boldsymbol{\theta}}(a|s)$ is the policy function and denotes the probability taking an action $a \in A$ based on the current state $s \in S$, and $R(s, a, s')$ is the reward function based on the state s, the action a, and the new state s' after the action is taken. In our study, the state of each agent of a process corresponds to which blocks are assigned to the process. Agents can take action to move blocks among processes and change the states. We will explain the details in Section 5.2.1.

The policy gradient based methods parameterize $\pi_{\boldsymbol{\theta}}$, the policy of agents, using a parameter $\boldsymbol{\theta}$. The policy is optimized by updating $\boldsymbol{\theta}$ iteratively using gradient ascent to maximize rewards:

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha \cdot \nabla_{\boldsymbol{\theta}} E_{\boldsymbol{\theta}}[R(s, a, s')], \tag{5.2}$$

where α is the learning rate controlling how quickly the policy parameter is adapted to the rewards. $E_{\boldsymbol{\theta}}[R(s,a,s')]$ is the expectation of rewards, and $\nabla_{\boldsymbol{\theta}} E_{\boldsymbol{\theta}}[R(s,a,s')]$ is the gradient of the expectation. According to Sutton and Barto [153], $\nabla_{\boldsymbol{\theta}} E_{\boldsymbol{\theta}}[R(s,a,s')]$ is usually difficult to compute analytically, but can be efficiently approximated using rewards of actions sampled from the policy function $\pi_{\boldsymbol{\theta}}$ by:

$$\nabla_{\boldsymbol{\theta}} E_{\boldsymbol{\theta}}[R(s,a,s')] \approx \frac{1}{|\widetilde{A}|} \sum_{a \in \widetilde{A}} \nabla_{\boldsymbol{\theta}} \ln(\pi_{\boldsymbol{\theta}}(a|s)) \cdot R(s,a,s'), \tag{5.3}$$

where $\widetilde{A}$ denotes the set of sampled actions. To reduce the computational cost, Williams [168] proposed a stochastic-gradient-ascent based algorithm, called REINFORCE [153], to update the policy based on the reward of one single sampled action by:

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha \cdot \nabla_{\boldsymbol{\theta}} \ln(\pi_{\boldsymbol{\theta}}(a|s)) \cdot R(s,a,s'). \tag{5.4}$$

The stochasticity is raised from the sampling procedure.

RL model for parallel particle tracing: Each process has an agent, which is responsible for assigning the blocks in the process to an appropriate process with a minimized local cost. At a specific execution time, each block movement decision is dependent on the current block assignment. This MDP of an agent can be described by a tuple $(S, A, \pi_{\boldsymbol{\theta}}, R)$, which is explained as follows.

State: The state space S consists of all combinations of data blocks. In particular, the agent of the process indexed by l has a time-dependent state $s_{t,l}$ indicating which blocks are owned by process l at time t. The state $s_{t,l}$ is corresponding to the block assignment at time t, i.e. $s_{t,l} = B_{t,l}$.

Action: The action space A consists of all actions for moving a block to a process. Specifically, the agent of process l has actions to move blocks out of $B_{t,l}$ to other

processes and change the current state; note that keeping a block in process l is a special yet valid action for the agent. Every agent considers the movement of one block at a time. The action $a_{i,l'} \in A$ denotes moving a block b_i, the block under the consideration, to the l'th process.

Policy function $\pi_{\boldsymbol{\theta}}(\cdot)$: Given the current state, the agent of process l has a policy function, $\pi_{\boldsymbol{\theta}}(a_{i,l'}|s_{t,l})$ (Equation 5.13), indicating the probability of the action $a_{i,l'}$ moving b_i from process l to process l'.

Reward function $R(\cdot)$: The reward function,

$$\begin{aligned} & R(s_{t-\Delta t,l}, a_{i,l'}, s_{t,l}) \\ = & \texttt{local\_exec\_cost}(B_{t-\Delta t,l}, B_{t-\Delta t,l}) \\ & -\texttt{local\_exec\_cost}(B_{t-\Delta t,l}, B_{t,l}), \end{aligned} \tag{5.5}$$

is defined by the reduction of a local execution cost function (Equation 5.6), after a block movement action $a_{i,l'}$ has been taken at time t. Here, $B_{t-\Delta t,l} \in \mathscr{B}_{t-\Delta t}$ with state $s_{t-\Delta t,l}$ represents the block assignment before an action $a_{i,l'}$ is taken, and $B_{t,l} \in \mathscr{B}_t$ with a new state $s_{t,l}$ denotes the assignment after block movement has occurred at time t.

5.2.2 Design of Cost Functions

Each process assesses the execution time imbalance locally by monitoring execution costs of friend processes, where processes l and l' are friends if the ranks l and l' only have one-bit difference following [17, 142]. Each process's local execution cost includes, (1) the maximal execution cost and (2) the standard deviation of the execution costs, among friend processes; hence, the decrease of a local execution cost is used for the reward computation, and is corresponding to the decrease of both the local maximal execution time and the imbalance among friend processes. We give the definition of

the local execution cost:

$$\begin{aligned} &\texttt{local\_exec\_cost}(B_{t-\Delta t,l}, B_{t,l}) \\ &= \max_{l' \in \mathscr{N}_B(l)} \texttt{cost}(B_{t-\Delta t,l'}, B_{t,l'}) + \texttt{STD}_l \end{aligned} \tag{5.6}$$

where $\mathscr{N}_B(l)$ represents the set of friend processes of process l. The standard deviation $\texttt{STD}_l$ is

$$\texttt{STD}_l = \sqrt{\frac{\sum_{l' \in \mathscr{N}_B(l)} (\texttt{cost}(B_{t-\Delta t,l'}, B_{t,l'}) - \texttt{AVG}_l)^2}{|\mathscr{N}_B(l)|}}, \tag{5.7}$$

and the local average cost $AVG[l]$ is

$$\texttt{AVG}_l = \frac{\sum_{l' \in \mathscr{N}_B(l)} \texttt{cost}(B_{t-\Delta t,l'}, B_{t,l'})}{|\mathscr{N}_B(l)|}, \tag{5.8}$$

Cost function $\texttt{cost}(\cdot)$ is designed to match the optimization objective in Equation 5.1. The total cost for the block adjustment from $B_{t-\Delta t,l}$ to $B_{t,l}$ is estimated by:

$$\begin{aligned} &\texttt{cost}(B_{t-\Delta t,l}, B_{t,l}) \\ =&\texttt{cost}_a(B_{t,l}) \\ +&\texttt{cost}_b(B_{t-\Delta t,l}, B_{t,l}) \\ +&\texttt{cost}_p(B_{t-\Delta t,l}, B_{t,l}), \end{aligned} \tag{5.9}$$

where $\texttt{cost}_a(B_{t,l})$ measures the estimated advection time for particles in the block set $B_{t,l}$, $\texttt{cost}_b(B_{t-\Delta t,l}, B_{t,l})$ measures the communication cost of block transfer from $B_{t-\Delta t,l}$ to $B_{t,l}$, and $\texttt{cost}_p(B_{t-\Delta t,l}, B_{t,l})$ measure the communication cost of transferring the particles between the block sets when necessary. The three costs are measured by seconds and, hence, can be directly added together to form the total cost. In the following, we discuss the three cost functions in detail.

1. Cost for particle advection: We balance the particle advection time of different processes based on the workload estimation model's outcomes in Section 5.3. Note that we estimate the time cost for advecting particles that enter a block in the

workload estimation model. Here, each process gathers how many particles will be transferred into its owned blocks. Then, each block's workload can be estimated using the counts of incoming particles in Equation 5.18 (explained in detail later), noted as w_j for the data block indexed by j. The cost $\texttt{cost}_a(B_{t,l})$ of a process l is the sum of all workload of the blocks within the process

$$\texttt{cost}_a(B_{t,l}) = \sum_{b_j \in B_{t,l}} w_j \tag{5.10}$$

2. Cost for transferring blocks: We move blocks among processes to balance the workload. The movement of a data block from the current process to the other one requires additional communication cost. The cost of exchanging block to transport $B_{t-\Delta t,l}$ to $B_{t,l}$ is computed by

$$\texttt{cost}_b(B_{t-\Delta t,l}, B_{t,l}) = |B_{t-\Delta t,l} \cup B_{t,l} - B_{t-\Delta t,l} \cap B_{t,l}| \cdot d_b, \tag{5.11}$$

where d_b is the transfer cost per data block given in Section 5.4.

3. Cost for transferring particles: The cost of particle transfer consists of sending and receiving particles. Given a process l, it sends particles to other processes when the particles in data block j owned by process l, for example, are now advected out of the block boundary and enter a neighboring block i but block i is not owned by process l after the block adjustment. Similarly, process l receives particles from other processes when particles in data block i in other processes exit the block boundaries and enter block j that is in process l after the block adjustment. Hence, the total

transfer cost of the process l is:

$$\begin{aligned} &\texttt{cost}_p(B_{t-\Delta t,l}, B_{t,l}) \\ &= \sum_{b_j \in B_{t-\Delta t,l}} \sum_{b_i \in \mathcal{N}_b(j) \text{ and } b_i \notin B_{t,l}} \widetilde{n}(i|j) \cdot d_p \\ &+ \sum_{b_j \in B_{t,l}} \sum_{b_i \in \mathcal{N}_b(j) \text{ and } b_i \notin B_{t-\Delta t,l}} \widetilde{n}(j|i) \cdot d_p \end{aligned} \tag{5.12}$$

where $\mathcal{N}_b(j)$ denotes the set of neighboring blocks of the given data block j, $\widetilde{n}(i|j)$ is the number of particles transferred from b_j to b_i, and d_p is the transfer cost per particle given in Section 5.4.

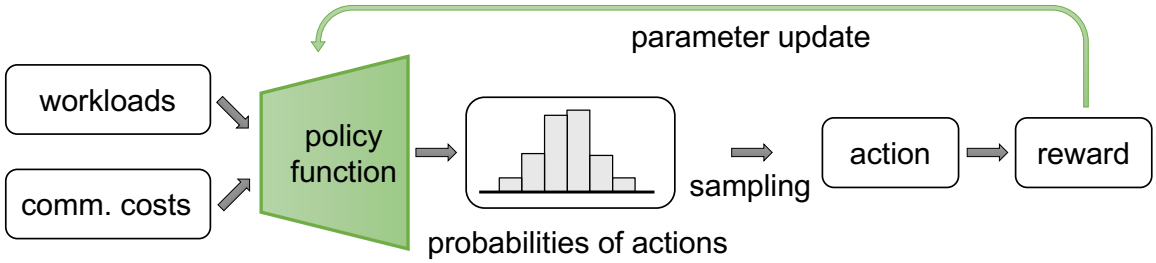


Figure 5.3: An agent's decision-making pipeline through a policy function.

5.2.3 Design of Policy Function and Update of Parameter

We illustrate a decision-making pipeline based on a policy function in Fig. 5.3. In the following, we explain how to parameterize the policy function, which considers both workloads of processes and possible transfer costs by taking a state-action feature vector as an input for the decision making of block movement. We then update the model parameter $\boldsymbol{\theta}$ using the policy gradient based on rewards after block movement occurs.

Policy function: The policy function projects a latent state $z_{a_{i,l'}}$ of an action $a_{i,l'}$ to a probability. We parameterize the policy function,

$$\pi_{\boldsymbol{\theta}}(a_{i,l'}|s_{t-\Delta t,l}) = \frac{e^{z_{a_{i,l'}}}}{\sum_{a_{i,l''} \in A} e^{z_{a_{i,l''}}}}, \tag{5.13}$$

by parameter $\boldsymbol{\theta}$ using `Softmax` policy parameterization framework [2, 3, 106], which is shown to converge fast.

Latent state and function: A latent state of an action represents the value of the action and is a scalar, where the scalar is larger indicating the action is more favored. A latent state of an action $a_{i,l'}$ is mapped from an obaserved feature vector $\boldsymbol{\phi}(s_{t-\Delta t,l}, a_{i,l'})$ using a latent function,

$$z_{a_{i,l'}} = f_{\boldsymbol{\theta}}(\boldsymbol{\phi}(s_{t-\Delta t,l}, a_{i,l'})) = \frac{1}{w_i}\boldsymbol{\phi}(s_{t-\Delta t,l}, a_{i,l'}) \cdot \boldsymbol{\theta}, \tag{5.14}$$

where $\frac{1}{w_i}$, reciprocal of block i's workload, is used for the normalization for making decision across different blocks, and the bias term is omitted for brevity. The latent function is instantiated using a linear form, which represents the weighted combinations of the feature vector components. The weights in $\boldsymbol{\theta}$ are non-negative and indicate different features' importances. The linear latent function is usually applied in studies of decision-making [59, 138]. The linear latent function combined with the `Softmax` policy is categorized into the log-linear policy class in RL [2].

State-action feature vector: A feature vector $\boldsymbol{\phi}(s_{t-\Delta t,l}, a_{i,l'})$ is a three-dimensional vector formed to describe the observed information, including the workload imbalance improvement and the data transfer cost associated with the action to move block b_i to process l'. The feature vector consists of three components.

The first feature component is to encourage to move block b_i out of process l to the other process l' with a lower workload. The value is computed by the workload of

process l (excluding the workload of b_i) reduced by the workload of l'. If the workload of l excluding b_i is higher than the workload of l', this feature component becomes positive, encouraging the action. Note that this feature component is zero if l' equals l, i.e., when the action is to make b_i stay in process l.

The second and third components correspond to the additional block transfer cost and particle transfer cost, respectively, when moving a block b_i from process l to l'. We set the sign of the two components to be negative as indicating they are cost. The second component is related to the additional data block transfer cost

$$\boldsymbol{\phi}_2(s_{t-\Delta t,l}, a_{i,l'}) = \begin{cases} 0, \text{if } l = l' \\ \\ -d_b, \text{otherwise.} \end{cases} \tag{5.15}$$

The third component is the particle transfer cost resulted from the block movement. Block i, after given to process l', needs to receive all incoming particles from other blocks that did not belong to process l' before:

$$\boldsymbol{\phi}_3(s_{t-\Delta t,l}, a_{i,l'}) = - \sum_{b_j \in \mathcal{N}_b(i) \text{ and } b_j \notin B_{t-\Delta t,l'}} \widetilde{n}(i|j) \cdot d_p \tag{5.16}$$

Policy parameter update: The parameter $\boldsymbol{\theta}$ is updated to maximize the expected rewards of sampled actions, encouraging actions to approach the minimum of the local load cost function. Following Equation 5.4, we update the policy parameter $\boldsymbol{\theta}$ using stochastic gradient ascent by:

$$\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha \cdot \nabla_\theta \ln(\pi_{\boldsymbol{\theta}}(a_{i,l'}|s_{t-\Delta t,l})) \cdot R(s_{t-\Delta t,l}, a_{i,l'}, s_{t,l}). \tag{5.17}$$

5.3 High-Order Blockwise Advection Workload Estimation Model

We estimate the particle advection workload in a given block using a high-order estimation model for advection cost computations when RL agents move blocks. Given

a data block, the input of this workload estimation model consists of the statistics of the incoming particles to the data block; the output of the model is the expected advection workload of those particles.

The blockwise workload estimiation is based on the historical records of particles that have been advected previously. When a particle enters a data block, the total number of advection steps is a deterministic number, although unknown before the advection occurs. For a given data block, the existing zeroth-order model [131, 180] assumes every entering particle has the similar advection integration steps. Hence, this model averages the integration steps of particles that have been advected in the given block for the estimation of incoming particles.

Our high-order model utilize the closeness of particle trajectories. By assuming continuities exist in spacetime domain following [57, 159], particles with close-by entry points into a data block usually have similar numbers of advection steps; also, particles whose trajectories are close to one another in the domain before entering the block tend to have similar entry points. Because matching the full trajectories of particles leads to high overhead, we record only the sequence of blocks accessed by those particles as an approximation of the trajectories, at a much lower cost. This is inspired by the particle access dependency modeling work of Zhang et al. [179], which discretizes the trajectories of particles into sequences of data blocks accessed by the particles and then uses a high-order Markov chain [137] to estimate the particles' data access patterns.

An additional parameter r is used to control how many data blocks most recently accessed by each particle are to be recorded, where r is the *order* of our estimation model and controls the model complexity. Existing works [131, 180] use the historical

average of the number of advection steps to estimate the workload of incoming particles, which is a special case when r is zero (zeroth-order) in our model.

Our method has two steps, as illustrated in Fig. 5.4. First, we match the trajectories of incoming particles with those of the previously advected particles. Second, our method uses the historical particles' numbers of advection steps that have the best-matched trajectories with the particles in question to estimate the workload of advection in the current block.

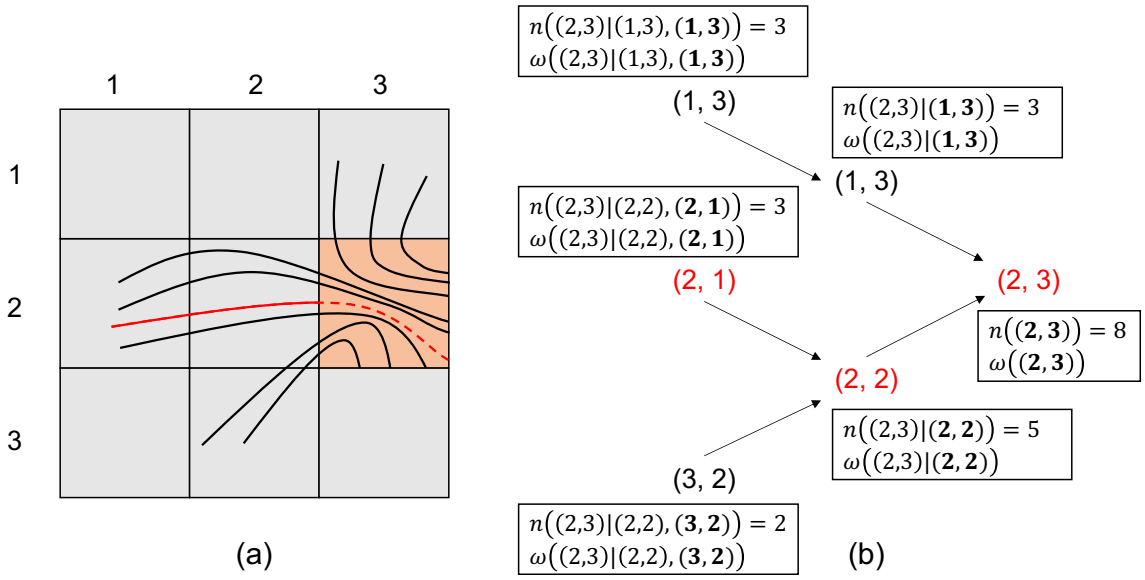


Figure 5.4: An illustration of high-order advection workload estimation. (a) A 2D example. Eight particles, whose trajectories are colored in black, have been traced previously within the block $(2, 3)$ with actual numbers of advection steps recorded; a new particle, colored by red, is a newly incoming particle. The zeroth-order estimation model uses the average of the numbers of advection steps of the eight particles traced to estimate the workload of the incoming particle. While, the high-order estimation model uses the numbers of the advection steps of the three particles close to the incoming particle and also passed through block $(2, 1)$, $(2, 2)$, and $(2, 3)$ for the estimation. (b) The corresponding trajectories tree. We represent the high-order (second-order in this example) workload estimation model by abstracting the particles' accessed blocks using a tree structure with a depth of two rooted at $(2, 3)$.

5.3.1 Data Structure of Trajectories Tree

Our model's data structure is a tree, called trajectories tree, which group particles based on where they come from. Trajectories trees are created from the block sequences accessed by the previously advected particles; the depth of the tree is r. Every block has a corresponding tree structure. For example, in Fig. 5.4b, the tree is formed by the access sequences of blocks by the eight black-colored particles in Fig. 5.4a and is stored in the process that owns block $(2, 3)$.

Each tree node has two attributes: $n(\cdot)$ and $\omega(\cdot)$, which are the statistics computed from the particles that have their data access sequence passing from the blocks of the tree nodes to the root. $n(i_r|i_{r-1}, ..., i_0)$ denotes the number of previoulsy advected particles that travel through the blocks of i_0, ..., i_{r-1} to i_r; while $\omega(i_r|i_{r-1}, ..., i_0)$ represents the historical average of the numbers of advection steps per particle in i_r for particles that traveled from blocks i_0 to i_{r-1}. This tree data structure is saved in the process of the root block.

5.3.2 Workload Estimation of Particle Advection

We estimate the workload of particle advection in data block i_r based on the incoming particles' trajectories. For example, for the new particle (red particle in Fig. 5.4a) that is going to access the block $(2, 3)$, we search the trajectories tree in Fig. 5.4b top-down to find a match of the particle's block based trajectory with those of the previously advected particles for workload estimation. Because the incoming particle traveled through blocks $(2, 1)$, $(2, 2)$, and $(2, 3)$ in order, we use the attribute $\omega((2, 3)|(2, 2), (2, 1))$ stored in the node $(2, 1)$ to estimate the workload for the particle;

$\omega((2,3)|(2,2),(2,1))$ is the average advection step of the black-colored three particles traveled through the block $(2,1)$.

Estimation: More formally, we estimate the advection workload, noted by w_{i_r}, of all incoming particles to the block i_r:

$$w_{i_r} = \sum_{i_{r-1} \in \mathscr{N}_b(i_r)} \sum_{i_{r-2} \in \mathscr{N}_b(i_{r-1})} ... \sum_{i_0 \in \mathscr{N}_b(i_1)} \widetilde{n}(i_r|i_{r-1},...,i_0) \cdot d_a \cdot \omega(i_r|i_{r-1},...,i_0), \tag{5.18}$$

where $\mathscr{N}_b(\cdot)$ gives neighboring data blocks, d_a is the historical time cost per advection step, and $\widetilde{n}(i_r|i_{r-1},...,i_0)$ denotes the number of the particles that have their block-based trajectories as i_0, ..., i_{r-1} and are now going to enter the block i_r. $\widetilde{n}(\cdot)$ is computed and exchanged among the parallel processes before the workload estimation.

Boundary case: If the trajectories tree data structure has no records of $\omega(i_r|i_{r-1},...,i_0)$, we will use an existing record $\omega(i_r|i_{r-1},...,i_k)$ with the smallest k to approximate $\omega(i_r|i_{r-1},...,i_0)$. For example, Fig. 5.4b, if a new particle accesses blocks $(1,2)$, $(1,3)$, and $(2,3)$, we will use the attributes stored on the tree node $(1,3)$ to estimate the workload.

5.3.3 Online Update of Estimation Model

We update the attributes, $n(i_r|\cdot)$ and $\omega(i_r|\cdot)$, stored in the trajectories tree nodes every time after particles finish advection within block i_r.

Update of attribute $\mathbf{n(i_r|\cdot)}$: $n(i_r|i_{r-1},...,i_0)$ is updated directly based on the blocks that the advected particles have passed through; if particles have accessed less than r blocks, we use their seeding block ID to pad the trajectory of the block sequence until the length becomes r.

Then, we update the aggregate statistics $n(i_r|i_{r-1},...,i_k)$, which indicates the sum of all particles that appear in the block i_r and previously pass through the blocks

from i_k to i_{r-1} using the following formula:

$$n(i_r|i_{r-1},...,i_k) = \sum_{i_{k-1}\in\mathscr{N}_b(i_k)} n(i_r|i_{r-1},...,i_k,i_{k-1}), \tag{5.19}$$

k is looped starting from 1 to $r-1$ for the update from the bottom to the top of the trajectories tree. Finally, we update the tree root

$$n(i_r) = \sum_{i_{r-1}\in\mathscr{N}_b(i_r)} n(i_r|i_{r-1}). \tag{5.20}$$

Update of blockwise per-particle workload: The estimated workload per particle, $\omega(i_r|i_{r-1},...,i_0)$, represents the estimated number of advection steps for the particles while advecting inside block i_r that have their block-based trajectory sequence as i_0, ..., i_r. $\omega(i_r|i_{r-1},...,i_0)$ is updated directly after each particle finishes its advection within the data block; if the number of a particle's trajectory sequence in blocks is less than r, we pad the block sequence using its initial seeding block ID until the sequence length becomes r.

We update $\omega(i_r|i_{r-1},...,i_k)$; $\omega(i_r|i_{r-1},...,i_k)$ is computed by averaging the numbers of advection steps of those particles in block i_r and previously pass through blocks from i_k to i_{r-1}. $\omega(i_r|i_{r-1},...,i_k)$ can be obtained using $\omega(i_r|i_{r-1},...,i_{k-1}), \forall i_{k-1}\in\mathscr{N}_b(i_k)$:

$$\begin{aligned}&\omega(i_r|i_{r-1},...,i_k)\\ =&\frac{\sum_{i_{k-1}\in\mathscr{N}_b(i_k)} n(i_r|i_{r-1},...,i_k,i_{k-1})\cdot\omega(i_r|i_{r-1},...,i_k,i_{k-1})}{n(i_r|i_{r-1},...,i_k)},\end{aligned} \tag{5.21}$$

k is also looped starting from 1 to $r-1$ for the bottom-up update of the trajectories tree. Finally, we update the tree root

$$\omega(i_r) = \frac{\sum_{i_{r-1}\in\mathscr{N}_b(i_r)} n(i_r|i_{r-1})\cdot\omega(i_r|i_{r-1})}{n(i_r)}. \tag{5.22}$$

5.4 Communication Cost Model

We build linear transmission cost models [30,85,158] to estimate the cost of fetching a data block, d_b, and the cost of transferring a particle, d_p. The linear transmission cost model [30, 85, 158] considers that the time cost of transferring a data entity from one process to the other is proportional to the size of the entity plus a fixed starting latency. Because the sizes of data blocks are similar and the sizes of particles are the same, we assume moving each block and particle has a similar cost. Linear models can be fitted efficiently, hence, has low overhead for the distributed and parallel computation.

Historical record collecting: We first collect the records of the data-block transfer time. After the initialization stage, each process fetches both necessary vector-field data blocks and the corresponding trajectories trees from other related processes, which can benefit from the fact that we already maintain a block-to-process mapping in the work donation model; the distributed block-to-process mapping can assist each process to identify which processes have the data of blocks efficiently. We record the time of the data migration every time, and fit a linear model to acquire the migration time per data block d_b. Similarly, to acquire the transfer cost of particles, the historical particle transfer time is recorded.

Generally, each historical record of these two types of time is a tuple where the first item is how many involved entities, noted as x, for an event and the second item is the total time, noted as y, of an event.

Fitting for estimation: We build a linear model

$$y = d \cdot x + e, \tag{5.23}$$

where d is a constant time cost of each event and e is an additional latency for each additional entity, to fit each of the two types of historical records by using the least-squares method. d_b and d_p are given by the fitted d based on the two types of historical records, respectively.

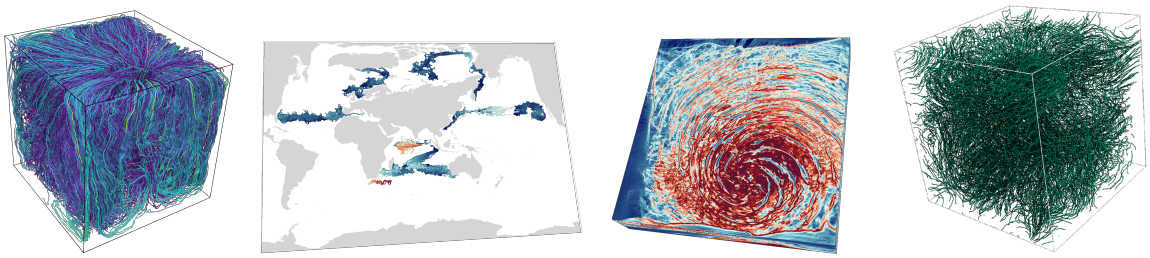
5.5 Performance Evaluation

We studied our method's performance using four particle-tracing tasks, including static flow analysis on a Nek5000 dataset via streamlines, source-destination analysis on a Ocean dataset via pathlines, unsteady flow analysis on a Isabel dataset via FTLEs, and static turbulence analysis on a Turbulence dataset via streamlines.

Implementation details: We prototyped our methods based on `Python 3` [163]. The advection integral was implemented using `C` [139] programs for fast computation and then imported into `Python` with the support of `Cython` [14].

Processes' communications and I/O are supported by `mpi4py` [38] and `DIY` [111,113, 130] libraries with asynchronous parallelism and nonblocking communications. `mpi4py` library [38] offers python-based Message-Passing Interface (MPI) implementations and is used for the communications amongst processes. The efficiency of the block-structured parallel I/O is improved by using the block-IO layer [83] supported by `DIY` library [111, 113, 130]. To store the block assignment $\mathscr{B}_t$ in memory, a block-to-process mapping records which blocks belong to which processes, and is maintained in distributed memory using `MPI` one-sided communications with low overhead, following the dynamic assignment algorithm of DIY library [111, 113, 130].

`PyTorch` library [128] supports the training of RL agents. The `autograd` [127] module supports automatic differentiation for gradient and derivative computation,



(a) (b) (c) (d)

Figure 5.5: Examples of rendering results. (a) We generated 4,096 streamlines on Nek5000 dataset for static flow analysis. (b) We used 2,592 pathlines for the Ocean dataset, which are seeded near Eurasia for the source-destination query. (c) We tested the Isabel dataset by using an FTLE field at timestep 0 within a time range of 16. (d) We generated 4,096 streamlines using the Turbulence dataset for flow turbulence analysis.

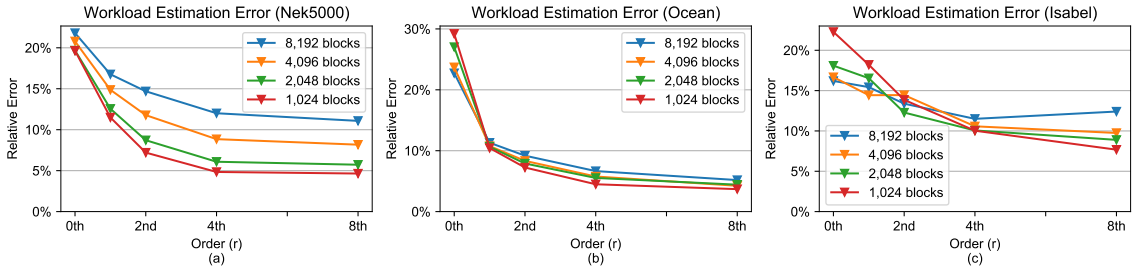


Figure 5.6: Advection workload estimation errors (defined in Section 5.5.1) under different processes and order settings. Three subfigures are corresponding to the three datasets: Nek5000, Ocean, and Isabel, respectively.

which is then used to update the parameters of agents through backpropagation [140] and RMSProp [156] optimizer. RMSProp can adapt the learning rate (i.e., α in Equation 5.17) automatically and was reported [86] to handle non-stationary environments well. Hence, RMSProp is commonly used in reinforcement learning studies, such as [107–109].

Eight blocks are assigned to each process initially, where the initial block assignment is then adjusted by our work donation algorithm dynamically during the runtime.

Table 5.1: Specifications of the three datasets used in this paper. We seed uniformly on Nek5000 and Turbulence to trace streamlines, seed locally on Ocean data to produce pathlines, and seed at all grid points of Isabel data to generate a FTLE field.

Dataset	Domain	Time Step	Size	Visualization Application	Seed Count	Maximum Advection Steps
Nek5000	$512 \times 512 \times 512$	1	1.5 GiB	Streamlines	2M	$1,024$
Ocean	$3,600 \times 2,400 \times 1$	36	2.3 GiB	Pathlines	2.7M	$1,024$
Isabel	$500 \times 500 \times 100$	48	13.4 GiB	FTLE	25M	48
Turbulence	$4,096 \times 4,096 \times 4,096$	1	768 GiB	Streamlines	2.6M, 16.8M, 134.2M	$1,024$

Baseline: We implement the most recently published lifeline based work stealing/requesting approach [17]. The lifeline based approach [17] was shown to improve performance compared with previous work stealing/requesting methods [97, 117] and thus is the current state-of-the-art in dynamic load balancing for parallel particle tracing.

Datasets: We evaluate our methods on four datasets. Detailed data specifications are included in Table 5.1.

The *Nek5000* dataset is a thermal-hydraulics dataset produced by the Nek5000 solver of a large-eddy Navier-stokes simulation [50]. We took one timestep of the simulation output to analyze the flow patterns of the fluid dynamics statically using streamlines (shown in Fig. 5.5a).

The *Ocean* dataset was produced by an eddy-resolving simulation with $1\backslash 10^{\circ}$ horizontal spacing [101]. The dataset consists of monthly averaged time-varying vector fields from February 2001 to January 2004. Source-destination query analysis is performed with local dense seeding by using pathline visualizations (shown in Fig. 5.5b), where a source-destination query is related to seed particles in local regions and query the destination of those seeded particles.

The hurricane *Isabel* data was produced from an atmospheric simulation developed by the National Center for Atmospheric Research. We perform an unsteady flow analysis using FTLEs (shown in Fig. 5.5c).

The isotropic *Turbulence* dataset is a direct numerical simulation of turbulent fluid flow on a 4096^3 grid [176] and is maintained by the Johns Hopkins Turbulence Databases [94]. When the simulation achieved a statistically stationary state, one snapshot of data was output and analyzed statically using streamlines (shown in Fig. 5.5d).

We have different seeding settings for the analysis of the four datasets. For the static flow analysis on Nek5000 and Turbulence data, we uniformly seeded particles to generate streamlines. For the source-destination query on Ocean, we locally seeded particles near Eurasia, consisting of Europe and Asia. For the unsteady flow analysis Isabel, we generated an FTLE field by tracing from all grid points.

Computing platforms specifications: We evaluate Nek5000, Ocean, and Isabel data on Bebop high-performance computing (HPC) cluster and test Turbulence data on Theta supercomputer.

Bebop HPC cluster has 664 compute nodes and uses IBM General Parallel File System. Every compute node has 32 Intel Xeon E5-2695v4 CPU cores with 4 GB memory per core. The compute nodes are interconnected by an Intel Omni-Path network. Message passing uses the Intel MPI library. We used up to $1,024$ cores for the following studies.

Theta supercomputer has $4,392$ compute nodes and uses Lustre Parallel File System. Every compute node has 64 Intel Xeon Phi 7230 processors and 3 GiB memory per processor. The compute nodes are interconnected by Cray's Aries technology and

integrated by the Dragonfly network topology. We used up to $16,384$ processors on Theta.

5.5.1 Advection Workload Estimation Study

We evaluate the accuracy of our high-order workload estimation model. Fig. 5.6 shows the estimation errors when the workload estimation model uses varying orders, where the order represents the length of the recorded accessed blocks for each particle and indicates the complexity of the model. The *relative error* is computed by the sum of the absolute difference between estimated advection time and actual advection time in each block divided by the entire advection time.

The estimation error generally decreases as the order increases with different total block counts for domain decomposition, where higher block counts correspond to smaller block sizes. The result of zeroth-order has errors of around 20% for Nek5000 data, $23\% \sim 29\%$ for Ocean data, and $16\% \sim 22\%$ for Isabel data. With the order becomes eighth, the errors become $5\% \sim 11\%$ for Nek5000 data, $4\% \sim 5\%$ for Ocean data, and $8\% \sim 12\%$ for Isabel data.

The error is difficult to decrease further after the model order is high enough because of the turbulence existing in the input data. Turbulence makes the vector field of a data block not continuous and causes particles to enter at similar entry positions into a data block yet still having different numbers of advection steps. We select fourth-order as the order setting for our following performance study because the errors do not decrease much when using orders higher than four.

5.5.2 Performance Study

To evaluate the performance of our method, we conducted the evaluation by using three measurements:

1. **Strong scaling:** By fixing the number of particles, we evaluate how the execution time changes along with increasing processes. The strong scaling measures the efficiency of our approach on a fixed-size problem with different process counts. Optimal strong scaling is achieved when the execution time is inversely proportional to the number of processes.

2. **Advection load imbalance:** The imbalance is measured by the metric "$\frac{\text{MAX}}{\text{AVG}}$", which is the maximal particle tracing time over all processes divided by the average tracing time per process, following previous studies [178, 180]. When the advection workload of processes is highly imbalanced, the value of this metric is large. As the workload of processes becomes similar, the metric is approaching 1.

3. **I/O and communication cost:** The I/O and communication cost measures the average time per process used for both data blocks' loading and interprocess exchange of data blocks and particles. We combine them together because both I/O and communication time is related to the data blocks' fetching during the runtime.

Strong scalability study: In Fig. 5.7 (a-c), we display the total execution time to evaluate our method's strong scaling performance up to using 1,024 processes. Our approach has lower total execution time than the baseline across the three test datasets when $1,024$ processes are used. Also, our method's total running time decreases faster

than the baseline does as the process count grows. Our method's overheads are 1.35, 0.92, and 12.33 seconds for the three datasets when $1,024$ processes are used, which are 6.19%, 2.02%, and 8.03% of the total execution time. Also, compared with the ideal scaling, the baseline attains the strong scaling efficiency of 29.06%, 14.07%, and 14.39% for the three datasets. Our method attains the parallel efficiency of 74.19%, 58.79%, and 29.98% for the three datasets, respectively, with speedup of 2.33x, 3.98x, and 36.93x over the baseline. The speedup comes from the improvement on both the workload imbalance and the costs of I/O and communication, which are demonstrated below.

Advection workload imbalance study: We display the advection workload imbalance for our method and the baseline approach on the second row of Fig. 5.7. Lower values are better, indicating the maximal advection workload over the average workload per process is smaller. The advection workload imbalance is important for parallel programs to minimize the idle time of processes, improving the total parallel efficiency. The workload imbalance for the three datasets evaluates whether our algorithm is general enough to optimize the workload assigned to processes for different tasks dynamically.

As shown in Fig. 5.7 (d-f), our method achieved lower imbalances than the baseline approach across the three datasets, although imbalance for both methods slightly increased as the process count grew. When $1,024$ processes were used, the baseline achieved the load imbalance of 1.33, 2.45, and 8.05 for the three test datasets, respectively. Correspondingly, our method achieves the load imbalance of 1.12, 2.12, and 1.81.

The results demonstrate proactively balancing workloads before processes become idle is effective. Agents in our work donation algorithm dynamically donate work from overloaded processes to underloaded processes, hence, explicitly reduce the maximal workload over processes and improves the workload imbalance $\frac{\text{MAX}}{\text{AVG}}$.

I/O and communication costs study: We show the I/O cost used for data block loading and the communication cost for the processes' exchange of both data blocks and particles in the third row of Fig. 5.7. As shown in Fig. 5.7 (g-i), our method outperformed the baseline across the three datasets. When $1,024$ processes were used, the baseline had the cost of 33.71, 152.59, and 5601.12 seconds for the three test datasets, respectively. Our method had the cost of 3.48, 5.06, and 51.15 seconds, which are 10.33%, 3.32%, and 0.91% of the time spent by the baseline. Compared with the baseline [17] that chooses to load data blocks from disks, our method is designed to fetch data blocks from other processes through network transfer and minimizes the communication cost for data exchange when optimizing the total execution time which reduces the I/O and communication time across the three applications.

Study on per-process performance: To breakdown the performance of our method in detail, we profile activities of each process, including local computation, I/O and communication, and busy waiting in Fig. 5.8. The results in Fig. 5.8 show that each process has a similar aggregated nonidle time, indicating our method can balance workloads among processes effectively.

Specific for Ocean data, Fig. 5.9 illustrates processes' dynamic block assignment. As shown in Fig. 5.9, the initial workloads of processes are not balanced at the early stage due to local seeding patterns, while the initial block assignment presents the round-robin (a.k.a, block-cyclic) assignment pattern. Our method gradually balances

the processes' workloads by transferring blocks among processes in the middle stage. At the late stage, only a few blocks have particles, making just a few processes have advection computations.

Study on $4,096^3$ Turbulence data: We evaluated our method on Turbulence data up to seeding 134.2 million particles and using $16,384$ processes. Turbulence has the highest spatial resolution and the largest data size among tested data in the existing parallel particle tracing [17, 24, 26, 27, 33, 34, 64, 68, 84, 89, 90, 97, 117, 125, 131, 134, 136, 143, 152, 177, 178, 180].

Fig. 5.10 presents the evaluations with varying numbers of particles on Turbulence data. Our method remains high strong-scaling efficiencies for varying numbers of particles, which are 87.59% for 2.6 million particles, 80.88% for 16.8 million particles, and 91.41% for 134.2 million particles. The advection load imbalance curve increases as the number of particles grows. The I/O and communication cost reduces almost linearly with respect to the increment of processes as the number of particles grows.

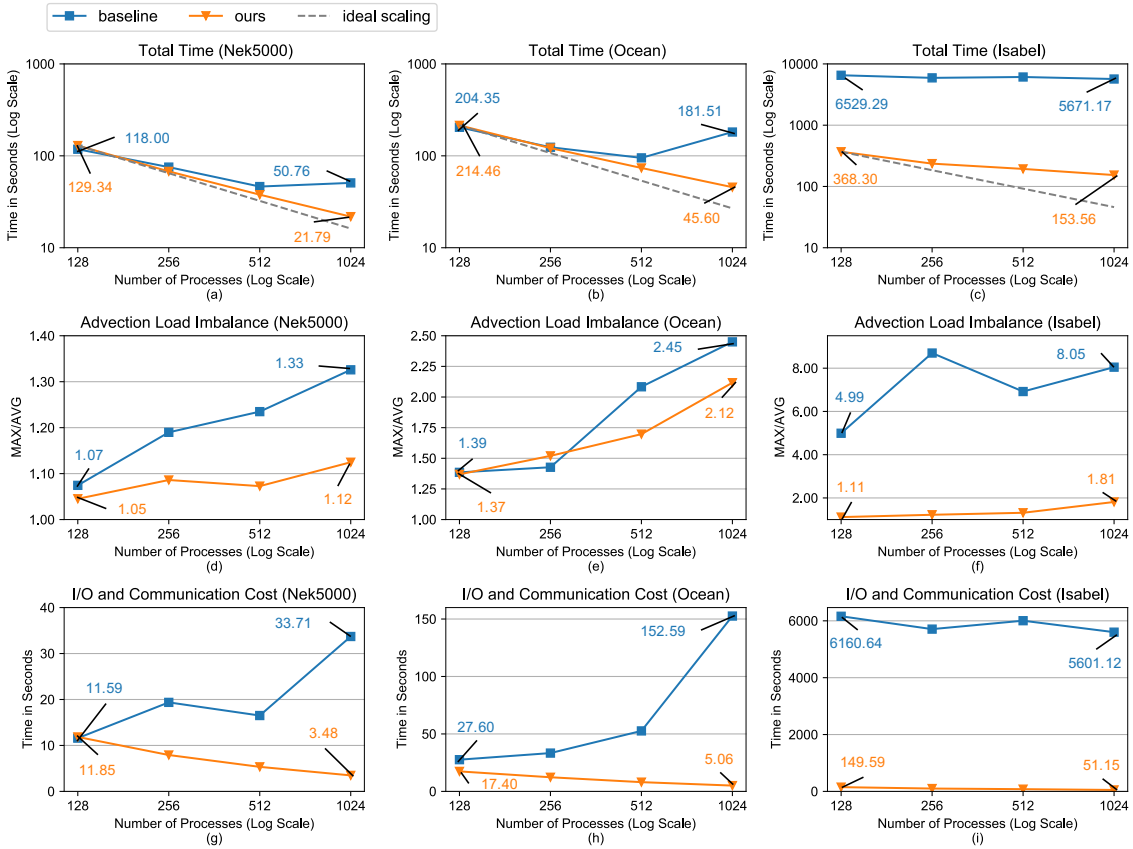


Figure 5.7: We present the total execution time, load balance, and I/O and communication time using the three datasets (Nek5000, Ocean, and Isabel) on the three rows separately, where the initial data loading time is included in both the total execution time and the I/O and communication time. We evaluate our method and compare it with the baseline approach from 128 processes to $1,024$ processes on the Bebop HPC cluster.

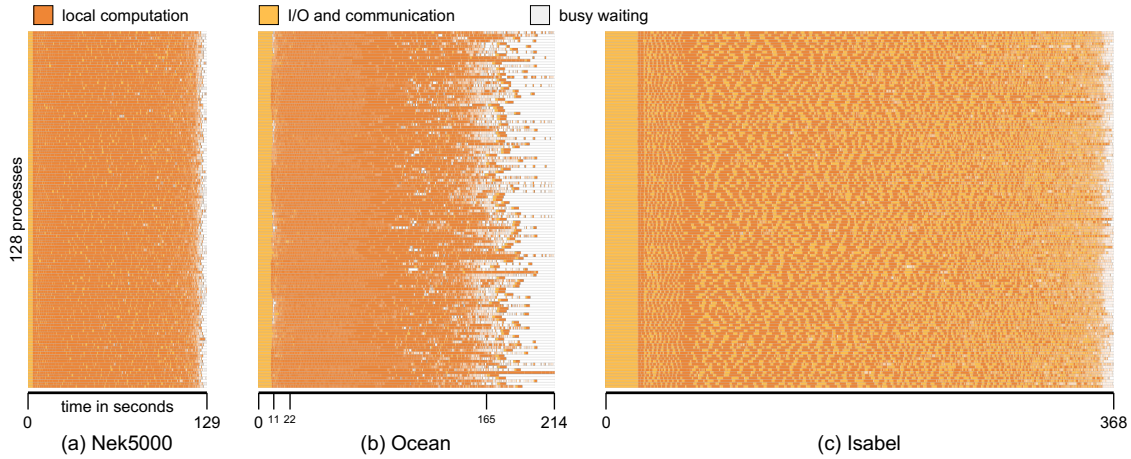


Figure 5.8: Gantt charts for our method using 128 processes on (a) Nek5000, (b) Ocean, and (c) Isabel data. Each row of the vertical axis corresponds to a process. The horizontal axis encodes the execution time.

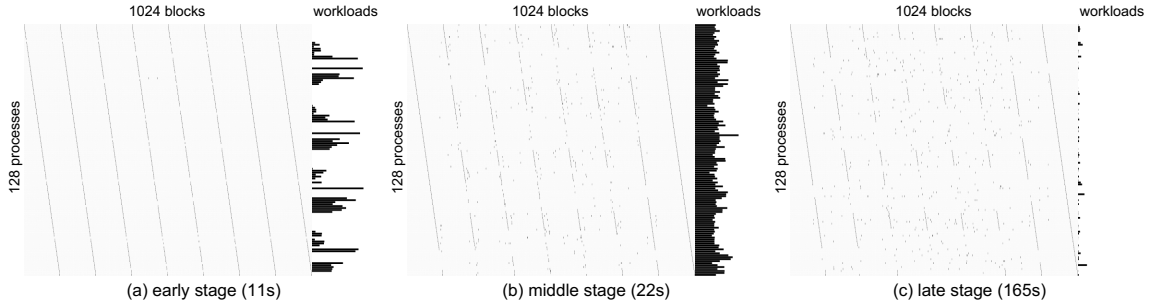


Figure 5.9: For Ocean data, we present time-varying block assignment when using our method with 128 processes. We extract three snapshots at 11, 22, and 165 seconds to illustrate the change of block assignment change at (a) early, (b) middle, and (c) late stages, respectively, which are also labeled in Fig. 5.8b. Each snapshot has two views. First, the left view illustrates the block-to-process assignment, where each row of the vertical axis corresponds to a process, and each column of the horizontal axis corresponds to a block. A dark dot represents a block is assigned to a process at the specified execution time. Second, the right view, a bar chart, encodes the workload of each process using bar length.

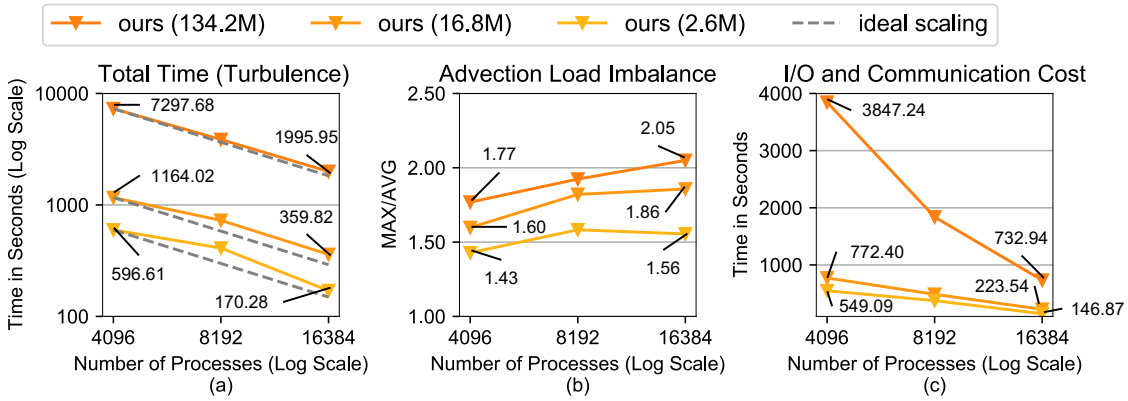


Figure 5.10: For $4,096^3$ Turbulence data, we present the total execution time, load balance, and I/O and communication time on the three columns separately with 2.6, 16.8, and 134.2 million particles. We evaluate our method from $4,096$ processes to $16,384$ processes on Theta supercomputer. The baseline approach is not presented here due to its execution time exceeding Theta's three-hour execution time constraint across different settings.

5.6 Discussions

Optimality limitations: We used the estimates of workloads to compute reward signals for agents. The quality of discovered strategy depends on the estimation. If the estimation errors are high, even if the agents can learn an optimal policy, the workload balance may still be suboptimal. To remedy this, in the future, we are going to train agents to calibrate and adjust the estimates of workloads by computing loss using the difference between the estimates and the real computation time, given the entry point of a particle and the vector field block, which may further decrease the estimation errors.

Out-of-core support: If the distributed memory of participating processes is not enough to hold the newly generated data (e.g., for in situ computation), we use the least-recently-used (LRU) rule to evict data blocks having no particles to save space for new data, following Pugmire et al. [134]. Because evicted data blocks may be reloaded in the future, we also write constructed trajectories trees of the blocks to disk for possible future reuse.

Trade off between I/O and interprocess communications: Our algorithm currently is designed to favor migrating data blocks among processes rather than loading data blocks from disks because most of HPC clusters and supercomputers have network bandwidth higher than I/O, which may not be beneficial for a computing platform with low network bandwidth and high-speed I/O. In the future, we will incorporate the estimation of I/O cost into our cost model to allow agents to fetch data blocks from disks if the I/O cost is lower.

Chapter 6: Conclusion and Future Work

We give a summary of the work presented in this dissertation and outline our future plans.

6.1 Conclusion

We conclude our solutions, including (1) a geometry-driven tracking and visualization of viscous and gravitational fingers, (2) a distributed asynchronous and load-balanced union-find algorithm, and (3) a reinforcement-learning based parallel particle tracing approach.

First, to detect viscous and gravitational fingering, we present a novel geometry-driven approach with a ridge voxel detection guided finger core extraction, a finger skeletonization and pruning method, and a spanning tree based finger branch extraction. An interactive visual-analytics system with a novel geometric-glyph augmented tracking graph is established for scientists to track the geometric growth, merging, and splitting of fingers over time in detail. The earth scientists recognized the value of our work and thought that this could significantly reduce their workload for the analysis of fingers both in space and time.

Second, we present a novel distributed union-find algorithm that (1) overlaps communications with computations using asynchronous parallelism to reduce synchronization costs and (2) redistributes set elements using a distributed k-d tree decomposition to balance processes' workloads for scalable scientific visualization and analysis. Our algorithm demonstrated improved scaling characteristics than existing distributed union-find methods in the scientific applications of critical point tracking and super-level set extraction. Benchmark datasets included synthetic data, exploding wire experimental data, and fusion plasma simulation data.

Third, we exploit the benefits of using RL based optimization approach to address a challenging workload balance and communication cost reduction problem for parallel flow visualization and analysis. The dynamic nature of reinforcement learning takes advantage of different cost functions to balance workload and minimize the communication cost. An optimized workload balance is achieved by considering the cost of communications for data transfer, enhancing the parallel particle tracing performance. We evaluate our approach with fluid dynamics, ocean simulation, and weather simulation data to analyze our method's workload balance and scaling efficiency. Our results demonstrate that our RL-based method can dynamically make block assignment decisions to minimize the total execution time.

6.2 Future Work

We describe possible extensions and improvements to our proposed solutions and give future plans.

Future work of viscous and gravitational fingering analysis: Our collaboration with the earth scientists continues to use the visualization and analysis

tools developed in this work to study the scaling behavior associated with pattern formation in flow instabilities. Specifically, the topological measurements offered by our methods, including the number of branches and the number of critical points, can be studied for discovering new scaling laws.

Feature work of our distributed union-find algorithm: In the future, first, we will evaluate our algorithm's performance in 4D (3D in space and 1D in time) scientific data. Distributed union-find algorithms can be extended to 4D seamlessly because the union-find algorithms accept elements and edges between elements as input, independent of the dimensionality of elements. Also, the element redistribution using distributed k-d trees can deal with elements with higher dimensions. Second, we will integrate our algorithm into other visualization and analysis applications, such as in situ visualization and graph/network data analysis.

Future works of RL-based parallel particle tracing: We identify four potential future works. First, the work donation and work stealing schemes can complement each other. In the future, we can allow agents to both donate and steal works from other processes with allowing duplicated data blocks, which can further improve the performance when just a few blocks have particles, for example, at the late stage in Fig. 5.9. Second, we may constrain the estimated advection time of a particle not larger than its remaining lifetime to reduce the workload estimation error further. For efficient processes' communications, we may compute the lifetime distribution as a compact representation, which is then exchanged among processes. Third, we can use deep neural networks to formulate the latent function instead of the linear model, which may exploit additional learnability for agents. Fourth, we will evaluate our algorithm on unsteady flow for in situ computation.

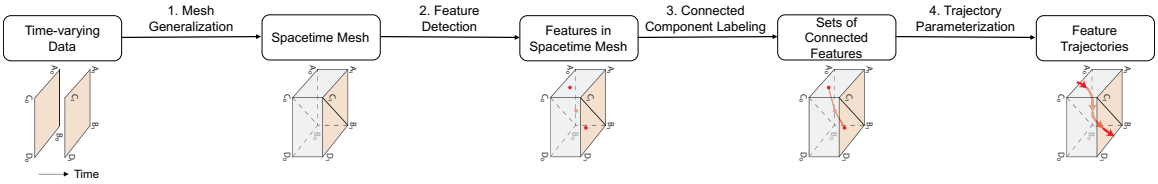


Figure 6.1: A possible pipeline for a general framework of feature tracking and visualization. Given the input time-varying data, first, we generalize it into spacetime mesh. Second, we detect features on the spacetime mesh. Third, we perform connected component labeling (CCL) to group features on the same trajectory into disjoint sets. Fourth, we parameterize the sets of features into geometric trajectories for visualization.

Future plans for scientific feature extraction and visualization : In the future, we will integrate our separate innovations for feature extraction and visualization into a general framework, which is illustrated in Figure 6.1 consisting of four stages. With the idea, we are collaborating to develop a toolkit called Feature Tracking Kit (FTK) [63, 65], which can facilitate scientists to track features, visualize feature trajectories, and derive scientific insights efficiently. Guided by the framework pipeline, challenges exist in different stages.

In the feature detection and tracking stage, the remaining difficulties include how to handle noise and how to preserve temporal continuity. First, detection robustness to noise is usually challenging. Researchers can choose to use Gaussian blurring to smooth the data to remedy the noise, however, which may remove sharp features. Recent advancements in convolutional neural networks show exceptional performance for robust feature detection in noisy imaging data [60] and bring the opportunities to detect scientific features robustly in an end-to-end manner. The other possible solution is to incorporate scientists' soft domain knowledge to help feature detection.

Active learning consists of a learning methodology to include humans in the learning process and improve the detection models based on human knowledge. Active learning has got success in feature detection in imaging data [4, 96], and has the potential to be applied to scientific feature detection. Second, to prevent losing tracks of features, it is important to predict features' movement to preserve the temporal continuity when the temporal resolution is not high enough. However, features' movement is usually non-linear and hard to predict. Given collected data of feature trajectories, we may use that to train neural networks to predict the movement of unseen features.

When visualizing the time-varying features, the scalability of visualizations usually is restricted by the screen, which may only display a limited number of features at a time. For example, our tracking graph for viscous and gravitational fingers can display four consecutive timesteps and show around one hundred fingers and hundreds of branches for each timestep when we use a screen resolution of 2880×1800. If more than one hundred fingers are at a timestep, the visual clutter becomes a bottleneck to arrange all the fingers in the same row. To remedy this scalability issue, we may further design a multi-level visualization for the tracked features and show more details or different facets when users interactively drill down into parts of features of interest.

Bibliography

[1] Tecplot, March 2018. `https://www.tecplot.com/`.

[2] Alekh Agarwal, Sham M Kakade, Jason D Lee, and Gaurav Mahajan. On the theory of policy gradient methods: Optimality, approximation, and distribution shift. *arXiv preprint arXiv:1908.00261*, 2019.

[3] Alekh Agarwal, Sham M Kakade, Jason D Lee, and Gaurav Mahajan. Optimality and approximation with policy gradient methods in markov decision processes. In *Proc. Conference on Learning Theory*, pages 64–66, 2020.

[4] Hamed H Aghdam, Abel Gonzalez-Garcia, Joost van de Weijer, and Antonio M López. Active learning for deep detection neural networks. In *Proc. IEEE International Conference on Computer Vision*, pages 3672–3680, 2019.

[5] Garrett Aldrich, Jonas Lukasczyk, Michael Steptoe, Ross Maciejewski, Heike Leitte, and Bernd Hamann. Viscous fingers: A topological visual analytics approach. *IEEE Scientific Visualization Contest*, 1(2):4, 2016.

[6] Mohammad Amin Amooie, Mohamad Reza Soltanian, and Joachim Moortgat. Hydrothermodynamic mixing of fluids across phases in porous media. *Geophysical Research Letters*, 44(8):3624–3634, 2017.

[7] Mohammad Amin Amooie, Mohamad Reza Soltanian, and Joachim Moortgat. Solutal convection in porous media: Comparison between boundary conditions of constant concentration and constant flux. *Physical Review E*, 98(3):033118, 2018.

[8] Mohammad Amin Amooie, Mohamad Reza Soltanian, Fengyang Xiong, Zhenxue Dai, and Joachim Moortgat. Mixing and spreading of multiphase fluids in heterogeneous bimodal porous media. *Geomechanics and Geophysics for Geo-Energy and Geo-Resources*, 3(3):225–244, 2017.

[9] Richard J Anderson and Heather Woll. Wait-free parallel algorithms for the union-find problem. In *Proc. of the twenty-third annual ACM symposium on Theory of computing*, pages 370–380, 1991.

[10] Utkarsh Ayachit. The paraview guide: a parallel visualization application. 2015.

[11] Michael Balzer and Oliver Deussen. Voronoi treemaps. In *Proc. IEEE Symposium on Information Visualization*, pages 49–56, 2005.

[12] Samer S Barakat and Xavier Tricoche. Adaptive refinement of the flow map using sparse samples. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2753–2762, 2013.

[13] Ulrich Bauer, Xiaoyin Ge, and Yusu Wang. Measuring distance between reeb graphs. In *Proc. Annual Symposium on Computational Geometry*, page 464, 2014.

[14] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2010.

[15] Marsha J Berger and Shahid H Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, (5):570–580, 1987.

[16] Silvia Biasotti, Daniela Giorgi, Michela Spagnuolo, and Bianca Falcidieno. Reeb graphs for shape analysis and applications. *Theoretical Computer Science*, 392(1-3):5–22, 2008.

[17] Roba Binyahib, David Pugmire, Boyana Norris, and Hank Childs. A lifeline-based approach for work requesting and parallel particle advection. In *Proc. Symposium on Large Data Analysis and Visualization*, pages 52–61. IEEE, 2019.

[18] Robert D Blumofe and Charles E Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[19] Cynthia A Brewer. Colorbrewer: Color advice for maps, September 2018. `http://www.ColorBrewer.org`.

[20] Mark Bruls, Kees Huizing, and Jarke J Van Wijk. Squarified treemaps. In *Data visualization*, pages 33–42, 2000.

[21] A Buka and P Palffy-Muhoray. Stability of viscous fingering patterns in liquid crystals. *Physical Review A*, 36(3):1527, 1987.

[22] A Buka, P Palffy-Muhoray, and Z Racz. Viscous fingering in liquid crystals. *Physical Review A*, 36(8):3984, 1987.

[23] Brian Cabral and Leith Casey Leedom. Imaging vector fields using line integral convolution. In *Proc. Annual Conference on Computer Graphics and Interactive Techniques*, pages 263–270, 1993.

[24] Brian Cabral and Leith Casey Leedom. Highly parallel vector visualization using line integral convolution. In *Proc. SIAM Conference on Parallel Processing for Scientific Computing, (PPSC)*, pages 802–807, 1995.

[25] Olca A Çakıroğlu, Cesim Erten, Ömer Karataş, and Melih Sözdinler. Crossing minimization in weighted bipartite graphs. In *International Workshop on Experimental and Efficient Algorithms*, pages 122–135, 2007.

[26] David Camp, Christoph Garth, Hank Childs, David Pugmire, and Kenneth Joy. Streamline integration using MPI-hybrid parallelism on a large multicore architecture. *IEEE Transactions on Visualization and Computer Graphics*, 17(11):1702–1713, 2010.

[27] David Camp, Hari Krishnan, David Pugmire, Christoph Garth, Ian Johnson, E. Wes Bethel, Kenneth I. Joy, and Hank Childs. GPU Acceleration of Particle Advection Workloads in a Parallel, Distributed Memory Setting. In Fabio Marton and Kenneth Moreland, editors, *Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2013.

[28] Hamish Carr, Jack Snoeyink, and Ulrike Axen. Computing contour trees in all dimensions. *Computational Geometry*, 24(2):75–94, 2003.

[29] Hamish Carr, Jack Snoeyink, and Michiel van de Panne. Flexible isosurfaces: Simplifying and displaying scalar topology using the contour tree. *Computational Geometry*, 43(1):42–58, 2010.

[30] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert Van De Geijn. Collective communication: Theory, practice, and experience. *Concurrency and Computation: Practice and Experience*, 19(13):1749–1783, 2007.

[31] Chun-Ming Chen, Boonthanome Nouanesengsy, Teng-Yok Lee, and Han-Wei Shen. Flow-guided file layout for out-of-core pathline computation. In *Proc. IEEE Symposium on Large Data Analysis and Visualization*, pages 109–112, 2012.

[32] Chun-Ming Chen, Lijie Xu, Teng-Yok Lee, and Han-Wei Shen. A flow-guided file layout for out-of-core streamline computation. In *Proc. IEEE Symposium on Large Data Analysis and Visualization*, pages 115–116, 2011.

[33] Li Chen and Issei Fujishiro. Optimizing parallel performance of streamline visualization for large distributed flow datasets. In *Proc. Pacific Visualization Symposium*, pages 87–94, 2008.

[34] Hank Childs, Scott Biersdorff, David Poliakoff, David Camp, and Allen D Malony. Particle advection performance over varied architectures and workloads.

In *Proc. International Conference on High Performance Computing*, pages 1–10, 2014.

[35] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Cyrus Harrison, Gunther H. Weber, Hari Krishnan, Thomas Fogal, Allen Sanderson, Christoph Garth, E. Wes Bethel, David Camp, Oliver Rübel, Marc Durant, Jean M. Favre, and Paul Navrátil. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pages 357–372. Oct 2012.

[36] Nicu D Cornea, Deborah Silver, and Patrick Min. Curve-skeleton properties, applications, and algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 13(3):0530–548, 2007.

[37] George Cybenko, Tom G Allen, and JE Polito. Practical parallel union-find algorithms for transitive closure and clustering. *International journal of parallel programming*, 17(5):403–423, 1988.

[38] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. MPI for Python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115, 2005.

[39] James Damon. Properties of ridges and cores for two-dimensional images. *Journal of Mathematical Imaging and Vision*, 10(2):163–174, 1999.

[40] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 752–768, 2018.

[41] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Vishwesh Jatala, Keshav Pingali, V Krishna Nandivada, Hoang-Vu Dang, and Marc Snir. Gluon-async: A bulk-asynchronous system for distributed and heterogeneous graph analytics. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 15–28, 2019.

[42] Pietro De Anna, Marco Dentz, Alexandre Tartakovsky, and Tanguy Le Borgne. The filamentary structure of mixing fronts and its control on reaction kinetics in porous media flows. *Geophysical Research Letters*, 41(13):4586–4593, 2014.

[43] M de Saint-Venant. Surfaces à plus grande pente constituées sur des lignes courbes. *Bulletin de la Société Philomathématique de Paris*, pages 24–30, 1852.

[44] James Dinan, D Brian Larkins, Ponnuswamy Sadayappan, Sriram Krishnamoorthy, and Jarek Nieplocha. Scalable work stealing. In *Proc. Conference on High Performance Computing Networking, Storage and Analysis*, pages 53:1–11, 2009.

[45] Harish Doraiswamy and Vijay Natarajan. Output-sensitive construction of reeb graphs. *IEEE transactions on visualization and computer graphics*, 18(1):146–159, 2012.

[46] B.D. Dudson, M.V. Umansky, X.Q. Xu, P.B. Snyder, and H.R. Wilson. BOUT++: A framework for parallel plasma fluid simulations. *Computer Physics Communications*, 180(9):1467–1480, 2009.

[47] Soumya Dutta and Han-Wei Shen. Distribution driven extraction and tracking of features for time-varying data analysis. *IEEE transactions on visualization and computer graphics*, 22(1):837–846, 2015.

[48] David Eberly. *Ridges in image and data analysis*, volume 7. Springer Science & Business Media, 2012.

[49] Guillaume Favelier, Charles Gueunet, and Julien Tierny. Visualizing ensembles of viscous fingers. In *IEEE Scientific Visualization Contest*, 2016.

[50] Paul Fischer, James Lottes, David Pointer, and Andrew Siegel. Petascale algorithms for reactor hydrodynamics. In *Proc. Journal of Physics: Conference Series*, volume 125, page 012076, 2008.

[51] Anke Friederici, Wiebke Köpp, Marco Atzori, Ricardo Vinuesa, Philipp Schlatter, and Tino Weinkauf. Distributed percolation analysis for turbulent flows. In *Proc. IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 42–51, 2019.

[52] Xiaojing Fu, Luis Cueto-Felgueroso, and Ruben Juanes. Pattern formation and coarsening dynamics in three-dimensional convective mixing in porous media. *Phil. Trans. R. Soc. A*, 371(2004):20120355, 2013.

[53] Jacob D Furst and Stephen M Pizer. Marching ridges. In *SIP*, pages 22–26, 2001.

[54] Zvi Galil and Giuseppe F Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys (CSUR)*, 23(3):319–344, 1991.

[55] Bernard A Galler and Michael J Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964.

[56] Michael R Garey, David S Johnson, and Larry Stockmeyer. Some simplified NP-complete problems. In *Proc. ACM symposium on Theory of computing*, pages 47–63, 1974.

[57] Christoph Garth, Xavier Tricoche, and Gerik Scheuermann. Tracking of vector field singularities in unstructured 3D time-dependent datasets. In *Proc. IEEE Visualization*, pages 329–336, 2004.

[58] Michael R Gary and David S Johnson. Computers and intractability: A guide to the theory of NP-completeness, 1979.

[59] Jan Gläscher, Nathaniel Daw, Peter Dayan, and John P O'Doherty. States versus rewards: dissociable neural prediction error signals underlying model-based and model-free reinforcement learning. *Neuron*, 66(4):585–595, 2010.

[60] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[61] William Gropp, Torsten Hoefler, Rajeev Thakur, and Ewing Lusk. *Using advanced MPI: Modern features of the message-passing interface*. MIT Press, 2014.

[62] Charles Gueunet, Pierre Fortin, Julien Jomier, and Julien Tierny. Task-based Augmented Reeb Graphs with Dynamic ST-Trees. In Hank Childs and Steffen Frey, editors, *Proc. Eurographics Symposium on Parallel Graphics and Visualization*. The Eurographics Association, 2019.

[63] Hanqi Guo. FTK: Feature Tracking Kit, October 2019. `https://github.com/hguo/ftk`.

[64] Hanqi Guo, Fan Hong, Qingya Shu, Jiang Zhang, Jian Huang, and Xiaoru Yuan. Scalable Lagrangian-based attribute space projection for multivariate unsteady flow data. In *Proc. IEEE Pacific Visualization Symposium*, pages 33–40, 2014.

[65] Hanqi Guo, David Lenz, Jiayi Xu, Xin Liang, Wenbin He, Iulian R Grindeanu, Han-Wei Shen, Tom Peterka, Todd Munson, and Ian Foster. FTK: A high-dimensional simplicial meshing framework for robust and scalable feature tracking. *arXiv preprint arXiv:2011.08697*, 2020.

[66] Hanqi Guo, Tom Peterka, and Andreas Glatz. In situ magnetic flux vortex visualization in time-dependent Ginzburg-Landau superconductor simulations. In *Proc. IEEE Pacific Visualization Symposium (PacificVis)*, pages 71–80, 2017.

[67] Hanqi Guo, Carolyn L Phillips, Tom Peterka, Dmitry Karpeyev, and Andreas Glatz. Extracting, tracking, and visualizing magnetic flux vortices in 3D complex-valued superconductor simulation data. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):827–836, 2015.

[68] Hanqi Guo, Xiaoru Yuan, Jian Huang, and Xiaomin Zhu. Coupled ensemble flow line advection and analysis. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2733–2742, 2013.

[69] Hanqi Guo, Jiang Zhang, Richen Liu, Lu Liu, Xiaoru Yuan, Jian Huang, Xiangfei Meng, and Jingshan Pan. Advection-based sparse data management for visualizing unsteady flow. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2555–2564, 2014.

[70] George Haller. Distinguished material surfaces and coherent structures in three-dimensional fluid flows. *Physica D: Nonlinear Phenomena*, 149(4):248–277, 2001.

[71] Robert M Haralick. Ridges and valleys on digital images. *Computer Vision, Graphics, and Image Processing*, 22(1):28–38, 1983.

[72] Cyrus Harrison, Jordan Weiler, Ryan Bleile, Kelly Gaither, and Hank Childs. A distributed-memory algorithm for connected components labeling of simulation data. In *Topological and Statistical Methods for Complex Data*, pages 3–19. 2015.

[73] Lifeng He, Xiwei Ren, Qihang Gao, Xiao Zhao, Bin Yao, and Yuyan Chao. The connected-component labeling problem: A review of state-of-the-art algorithms. *Pattern Recognition*, 70:25–43, 2017.

[74] Christian Heine, Dominic Schneider, Hamish Carr, and Gerik Scheuermann. Drawing contour trees in the plane. *IEEE Transactions on Visualization and Computer Graphics*, 17(11):1599–1611, 2011.

[75] Einar L Hinrichsen, KJ Maloy, Jens Feder, and T Jossang. Self-similarity and structure of dla and viscous fingering clusters. *Journal of Physics A: Mathematical and General*, 22(7):L271, 1989.

[76] Marcel Hlawatsch, Filip Sadlo, and Daniel Weiskopf. Hierarchical line integration. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1148–1163, 2010.

[77] George M Homsy. Viscous fingering in porous media. *Annual review of fluid mechanics*, 19(1):271–311, 1987.

[78] Fan Hong, Jiang Zhang, and Xiaoru Yuan. Access pattern learning with long short-term memory for parallel particle tracing. In *Proc. IEEE Pacific Visualization Symposium*, pages 76–85, 2018.

[79] John E. Hopcroft and Jeffrey D. Ullman. Set merging algorithms. *SIAM Journal on Computing*, 2(4):294–303, 1973.

[80] Jeremy Iverson, Chandrika Kamath, and George Karypis. Evaluation of connected-component labeling algorithms for distributed-memory systems. *Parallel Computing*, 44:53–68, 2015.

[81] Guangfeng Ji, Han-Wei Shen, and Rephael Wenger. Volume tracking using higher dimensional isosurfacing. In *Proc. IEEE Visualization*, pages 209–216, 2003.

[82] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proc. Visualization*, pages 284–291, 1991.

[83] Wesley Kendall, Jian Huang, Tom Peterka, Robert Latham, and Robert Ross. Toward a general I/O layer for parallel-visualization applications. *IEEE Computer Graphics and Applications*, 31(6):6–10, 2011.

[84] Wesley Kendall, Jingyuan Wang, Melissa Allen, Tom Peterka, Jian Huang, and David Erickson. Simplified parallel domain traversal. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 10:1–11, 2011.

[85] Thilo Kielmann, Henri E Bal, Sergei Gorlatch, Kees Verstoep, and Rutger FH Hofman. Network performance-aware collective communication for clustered wide-area systems. *Parallel Computing*, 27(11):1431–1456, 2001.

[86] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv*, abs/1412.6980, 2014.

[87] Sergei I Krasheninnikov. On scrape off layer plasma transport. *Physics Letters A*, 283(5-6):368–370, 2001.

[88] Aaditya G Landge, Valerio Pascucci, Attila Gyulassy, Janine C Bennett, Hemanth Kolla, Jacqueline Chen, and Peer-Timo Bremer. In-situ feature extraction of large scale combustion simulations using segmented merge trees. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1020–1031. IEEE, 2014.

[89] David A Lane. UFAT-a particle tracer for time-dependent flow fields. In *Proc. Visualization*, pages 257–264, 1994.

[90] David A Lane. Parallelizing a particle tracer for flow visualization. Technical report, Society for Industrial and Applied Mathematics, Philadelphia, PA (United States), 1995.

[91] Robert S. Laramee, Helwig Hauser, Helmut Doleisch, Benjamin Vrolijk, Frits H. Post, and Daniel Weiskopf. The state of the art in flow visualization: Dense and texture-based techniques. *Computer Graphics Forum*, 23(2):203–221, 2004.

[92] Francis Lazarus and Anne Verroust. Level set diagrams of polyhedral objects. In *Proc. ACM Symposium on Solid Modeling and Applications*, pages 130–140, 1999.

[93] Shuwang Li, John S Lowengrub, Jake Fontana, and Peter Palffy-Muhoray. Control of viscous fingering patterns in a radial hele-shaw cell. *Physical review letters*, 102(17):174501, 2009.

[94] Yi Li, Eric Perlman, Minping Wan, Yunke Yang, Charles Meneveau, Randal Burns, Shiyi Chen, Alexander Szalay, and Gregory Eyink. A public turbulence database cluster and applications to study lagrangian evolution of velocity increments in turbulence. *Journal of Turbulence*, (9):N31, 2008.

[95] Tony Lindeberg. Edge detection and ridge detection with automatic scale selection. *International Journal of Computer Vision*, 30(2):117–156, 1998.

[96] Zimo Liu, Jingya Wang, Shaogang Gong, Huchuan Lu, and Dacheng Tao. Deep reinforcement active learning for human-in-the-loop person re-identification. In *Proc. IEEE International Conference on Computer Vision*, pages 6122–6131, 2019.

[97] Kewei Lu, Han-Wei Shen, and Tom Peterka. Scalable computation of stream surfaces on large scale vector fields. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1008–1019, 2014.

[98] Timothy Luciani, Andrew Burks, Cassiano Sugiyama, Jonathan Komperda, and G Elisabeta Marai. Details-first, show context, overview last: Supporting exploration of viscous fingers in large-scale ensemble simulations. *IEEE Transactions on Visualization and Computer Graphics*, 2018.

[99] Jonas Lukasczyk, Garrett Aldrich, Michael Steptoe, Guillaume Favelier, Charles Gueunet, Julien Tierny, Ross Maciejewski, Bernd Hamann, and Heike Leitte. Viscous fingering: A topological visual analytic approach. In *Applied Mechanics and Materials*, volume 869, pages 9–19, 2017.

[100] Jonas Lukasczyk, Gunther Weber, Ross Maciejewski, Christoph Garth, and Heike Leitte. Nested tracking graphs. *Computer Graphics Forum*, 36(3):12–22, 2017.

[101] Mathew E Maltrud and Julie L McClean. An eddy resolving global 1/10 ocean simulation. *Ocean Modelling*, 8(1-2):31–54, 2005.

[102] Fredrik Manne and Md Mostofa Ali Patwary. A scalable parallel union-find algorithm for distributed memory computers. In *Proc. International Conference on Parallel Processing and Applied Mathematics*, pages 186–195, 2009.

[103] Joseph Marino and Arie Kaufman. Planar visualization of treelike structures. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):906–915, 2016.

[104] James E McClure, Mark A Berrill, Jan F Prins, and Cass T Miller. Asynchronous in situ connected-components analysis for complex fluid flows. In *Proc. Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 12–17, 2016.

[105] Tony McLoughlin, Robert S. Laramee, Ronald Peikert, Frits H. Post, and Min Chen. Over two decades of integration-based, geometric flow visualization. *Computer Graphics Forum*, 29(6):1807–1829, 2010.

[106] Jincheng Mei, Chenjun Xiao, Csaba Szepesvari, and Dale Schuurmans. On the global convergence rates of softmax policy gradient methods. In *Proc. International Conference on Machine Learning*, pages 6820–6829, 2020.

[107] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. *arXiv*, abs/1602.01783, 2016.

[108] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *arXiv*, abs/1312.5602, 2013.

[109] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[110] Joachim Moortgat. Viscous and gravitational fingering in multiphase compositional and compressible flow. *Advances in Water Resources*, 89:53–66, 2016.

[111] Dmitriy Morozov and Tom Peterka. Block-parallel data analysis with DIY2. In *Proc. IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, pages 29–36. IEEE, 2016.

[112] Dmitriy Morozov and Tom Peterka. Efficient delaunay tessellation through kd tree decomposition. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 728–738, 2016.

[113] Dmitriy Morozov and Tom Peterka. *DIY: data-parallel out-of-core library*, accessed January 2021. `https://github.com/diatomic/diy`.

[114] Dmitriy Morozov, Tom Peterka, Hanqi Guo, Mukund Raj, Jiayi Xu, and Han-Wei Shen. IExchange: Asynchronous communication and termination detection for iterative algorithms. In *Proc. IEEE Symposium on Large Data Analysis and Visualization (LDAV)*, 2021.

[115] Dmitriy Morozov and Gunther Weber. Distributed merge trees. In *Proc. ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 93–102, 2013.

[116] Dmitriy Morozov and Gunther H Weber. Distributed contour trees. In Peer-Timo Bremer, Ingrid Hotz, Valerio Pascucci, and Ronald Peikert, editors, *Topological Methods in Data Analysis and Visualization III*, pages 89–102. Springer, 2014.

[117] Cornelius Müller, David Camp, Bernd Hentschel, and Christoph Garth. Distributed parallel particle advection using work requesting. In *Proc. IEEE Symposium on Large-Scale Data Analysis and Visualization*, pages 1–6, 2013.

[118] Shigeru Muraki, Eric B Lum, K-L Ma, Masato Ogata, and Xuezhen Liu. A PC cluster system for simultaneous interactive volumetric modeling and visualization. In *Proc. IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 95–102, 2003.

[119] F Nespoli, I Furno, B Labit, P Ricci, F Avino, F D Halpern, F Musil, and F Riva. Blob properties in full-turbulence simulations of the TCV scrape-off layer. *Plasma Physics and Controlled Fusion*, 59(5):055009, mar 2017.

[120] Federico Nespoli, Patrick Tamain, Nicolas Fedorczak, Guido Ciraolo, Davide Galassi, Raffaele Tatali, Eric Serre, Yannick Marandet, Hugo Bufferand, and Philippe Ghendrih. 3D structure and dynamics of filaments in turbulence simulations of west diverted plasmas. *Nuclear Fusion*, 2019.

[121] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 456–471, 2013.

[122] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. *Distributed implementation of connected components in Galois system*, accessed July 2020. `https://iss.oden.utexas.edu/?p=projects/galois/analytics/dist-cc`.

[123] Arnur Nigmetov and Dmitriy Morozov. Local-global merge tree computation with local exchanges. In *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 60:1–13, 2019.

[124] Boonthanome Nouanesengsy, Teng-Yok Lee, Kewei Lu, Han-Wei Shen, and Tom Peterka. Parallel particle advection and FTLE computation for time-varying flow fields. In *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 61:1–11, 2012.

[125] Boonthanome Nouanesengsy, Teng-Yok Lee, and Han-Wei Shen. Load-balanced parallel streamline generation on large scale vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1785–1794, 2011.

[126] Valerio Pascucci and Kree Cole-McLaughlin. Parallel computation of the topology of level sets. *Algorithmica*, 38(1):249–268, 2004.

[127] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *Proc. NIPS Autodiff Workshop*, 2017.

[128] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. 32:8026–8037, 2019.

[129] Ronald Peikert and Filip Sadlo. Height ridge computation and filtering for visualization. In *Proc. IEEE Pacific Symposium on Visualization*, pages 119–126, 2008.

[130] Tom Peterka, Robert Ross, Attila Gyulassy, Valerio Pascucci, Wesley Kendall, Han-Wei Shen, Teng-Yok Lee, and Abon Chaudhuri. Scalable parallel building blocks for custom data analysis. In *Proc. IEEE Symposium on Large Data Analysis and Visualization*, pages 105–112, 2011.

[131] Tom Peterka, Robert Ross, Boonthanome Nouanesengsy, Teng-Yok Lee, Han-Wei Shen, Wesley Kendall, and Jian Huang. A study of parallel particle tracing for steady-state and time-varying flow fields. In *Proc. International Parallel & Distributed Processing Symposium*, pages 580–591, 2011.

[132] Carolyn L Phillips, Hanqi Guo, Tom Peterka, Dmitry Karpeyev, and Andreas Glatz. Tracking vortices in superconductors: Extracting singularities from a

discretized complex scalar field evolving in time. *Physical Review E*, 93(2):023305, 2016.

[133] Frits H. Post, Benjamin Vrolijk, Helwig Hauser, Robert S. Laramee, and Helmut Doleisch. The state of the art in flow visualisation: Feature extraction and tracking. *Computer Graphics Forum*, 22(4):775–792, 2003.

[134] Dave Pugmire, Hank Childs, Christoph Garth, Sean Ahern, and Gunther H Weber. Scalable computation of streamlines on very large datasets. In *Proc. of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 16:1–12, 2009.

[135] David Pugmire, Tom Peterka, and Christoph Garth. Parallel integral curves. In Charles Hansen E. Wes Bethel, Hank Childs, editor, *High Performance Visualization: Enabling Extreme Scale Scientific Insight*, pages 13–30. CRC Press, 2012.

[136] David Pugmire, Abhishek Yenpure, Mark Kim, James Kress, Robert Maynard, Hank Childs, and Bernd Hentschel. Performance-Portable Particle Advection with VTK-m. In *Proc. Eurographics Symposium on Parallel Graphics and Visualization*, 2018.

[137] Adrian E Raftery. A model for high-order markov chains. *Journal of the Royal Statistical Society: Series B (Methodological)*, 47(3):528–539, 1985.

[138] Paul Reverdy and Naomi Ehrich Leonard. Parameter estimation in softmax decision-making models with linear objective functions. *IEEE Transactions on Automation Science and Engineering*, 13(1):54–67, 2015.

[139] Dennis M Ritchie, Brian W Kernighan, and Michael E Lesk. *The C programming language*. Prentice Hall Englewood Cliffs, 1988.

[140] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*, pages 318–362. MIT Press, 1987.

[141] D. A. Russell, D. A. D'Ippolito, J. R. Myra, W. M. Nevins, and X. Q. Xu. Blob dynamics in 3D BOUT simulations of Tokamak edge turbulence. *Phys. Rev. Lett.*, 93:265001, Dec 2004.

[142] Vijay A Saraswat, Prabhanjan Kambadur, Sreedhar Kodali, David Grove, and Sriram Krishnamoorthy. Lifeline-based global load balancing. *ACM SIGPLAN Notices*, 46(8):201–212, 2011.

[143] Samuel D Schwartz, Hank Childs, and David Pugmire. Improving parallel particle advection performance with machine learning. In *Proc. Eurographics Symposium on Parallel Graphics and Visualization*, 2021.

[144] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *Proc. International Conference on High Performance Computing for Computational Science*, pages 1–25, 2010.

[145] Yossi Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[146] Julian Shun and Guy E Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proc. ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 135–146, 2013.

[147] Deborah Silver and Xin Wang. Volume tracking. In *Proc. Visualization*, pages 157–ff, 1996.

[148] Arne Skauge, Per Arne Ormehaug, Tiril Gurholt, Bartek Vik, Igor Bondino, Gerald Hamon, et al. 2-d visualisation of unstable waterflood and polymer flood for displacement of heavy oil. In *Proc. SPE Improved Oil Recovery Symposium*, 2012.

[149] Mohamad Reza Soltanian, Mohammad Amin Amooie, Zhenxue Dai, David Cole, and Joachim Moortgat. Critical dynamics of gravito-convective mixing in geological carbon sequestration. *Scientific Reports*, 6:35921, 2016.

[150] Mohamad Reza Soltanian, Mohammad Amin Amooie, Naum Gershenzon, Zhenxue Dai, Robert Ritzi, Fengyang Xiong, David Cole, and Joachim Moortgat. Dissolution trapping of carbon dioxide in heterogeneous aquifers. *Environmental Science & Technology*, 51(13):7732–7741, 2017.

[151] Carsten Steger. An unbiased detector of curvilinear structures. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(2):113–125, 1998.

[152] David Sujudi and Robert Haimes. Integration of particle paths and streamlines in a spatially-decomposed computation. In A. Ecer, J. Periaux, N. Satdfuka, and S. Taylor, editors, *Parallel Computational Fluid Dynamics 1995*, pages 315–322. Elsevier, 1996.

[153] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[154] Jay Takle, Deborah Silver, Eve Kovacs, and Katrin Heitmann. Visualization of multivariate dark matter halos in cosmology simulations. In *2013 IEEE*

Symposium on Large-Scale Data Analysis and Visualization (LDAV), pages 131–132. IEEE, 2013.

[155] Robert E Tarjan and Jan Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)*, 31(2):245–281, 1984.

[156] T. Tieleman and G. Hinton. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012. `https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`.

[157] Julien Tierny, Guillaume Favelier, Joshua A Levine, Charles Gueunet, and Michael Michaux. The topology toolkit. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):832–842, 2018.

[158] Jesper Larsson Träff. On optimal trees for irregular gather and scatter collectives. *IEEE Transactions on Parallel and Distributed Systems*, 30(9):2060–2074, 2019.

[159] Xavier Tricoche, Thomas Wischgoll, Gerik Scheuermann, and Hans Hagen. Topology tracking for the visualization of time-dependent two-dimensional flows. *Computers & Graphics*, 26(2):249–257, 2002.

[160] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[161] Marc Van Kreveld, René van Oostrum, Chandrajit Bajaj, Valerio Pascucci, and Dan Schikore. Contour trees and small seed sets for isosurface traversal. In *Proc. Annual Symposium on Computational Geometry*, pages 212–220, 1997.

[162] Jan van Leeuwen and R van der Weide. *Alternative path compression techniques*, volume 77. Unknown Publisher, 1977.

[163] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[164] Luc Vincent and Pierre Soille. Watersheds in digital spaces: an efficient algorithm based on immersion simulations. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (6):583–598, 1991.

[165] Zhehui Wang, Qiuguang Liu, W Waganaar, John Fontanese, David James, and Tobin Munsat. Four-dimensional (4D) tracking of high-temperature microparticles. *Review of Scientific Instruments*, 87(11):11D601, 2016.

[166] Zhehui Wang, Jiayi Xu, Yao E Kovach, Bradley T Wolfe, Edward Thomas Jr, Hanqi Guo, John E Foster, and Han-Wei Shen. Microparticle cloud imaging and tracking for data-driven plasma science. *Physics of Plasmas*, 27(3):033703, 2020.

[167] Theodorus Petrus Weide. *Datastructures: An Axiomatic Approach and the Use of Binomial Trees in Developing and Analyzing Algorithms*. PhD thesis, Leiden University, 1980.

[168] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[169] Jo Wood and Jason Dykes. Spatially ordered treemaps. *IEEE Transactions on Visualization and Computer Graphics*, 14(6), 2008.

[170] Jiayi Xu, Soumya Dutta, Wenbin He, Joachim Moortgat, and Han-Wei Shen. Geometry-driven detection, tracking and visual analysis of viscous and gravitational fingers. *IEEE Transactions on Visualization and Computer Graphics*, 2020.

[171] Jiayi Xu, Hanqi Guo, Han-Wei Shen, Mukund Raj, Xueyun Wang, Xueqiao Xu, Zhehui Wang, and Tom Peterka. Asynchronous and load-balanced union-find for distributed and parallel scientific data visualization and analysis. *IEEE Transactions on Visualization and Computer Graphics*, 27(6):2808–2820, 2021.

[172] Jiayi Xu, Hanqi Guo, Han-Wei Shen, Mukund Raj, Skylar Wolfgang Wurster, and Tom Peterka. Reinforcement learning for load-balanced parallel particle tracing. *arXiv preprint arXiv:2109.05679*, 2021.

[173] Lijie Xu and Han-Wei Shen. Flow Web: a graph based user interface for 3D flow field exploration. In *Visualization and Data Analysis*, volume 7530, page 75300F, 2010.

[174] X. Q. Xu, R. H. Cohen, T. D. Rognlien, and J. R. Myra. Low-to-high confinement transition simulations in divertor geometry. *Physics of Plasmas*, 7(5):1951–1958, 2000.

[175] Atsuko Yamaguchi and Akihiro Sugimoto. An approximation algorithm for the two-layered graph drawing problem. In *International Computing and Combinatorics Conference*, pages 81–91, 1999.

[176] PK Yeung, DA Donzis, and KR Sreenivasan. Dissipation, enstrophy and pressure statistics in turbulence simulations at high reynolds numbers. *Journal of Fluid Mechanics*, 700:5–15, 2012.

[177] Hongfeng Yu, Chaoli Wang, and Kwan-Liu Ma. Parallel hierarchical visualization of large time-varying 3D vector fields. In *Proc. ACM/IEEE Conference on Supercomputing*, pages 24:1–12, 2007.

[178] Jiang Zhang, Hanqi Guo, Fan Hong, Xiaoru Yuan, and Tom Peterka. Dynamic load balancing based on constrained kd tree decomposition for parallel particle tracing. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):954–963, 2017.

[179] Jiang Zhang, Hanqi Guo, and Xiaoru Yuan. Efficient unsteady flow visualization with high-order access dependencies. In *Proc. IEEE Pacific Visualization Symposium (PacificVis)*, pages 80–87, 2016.

[180] Jiang Zhang, Hanqi Guo, Xiaoru Yuan, and Tom Peterka. Dynamic data repartitioning for load-balanced parallel particle tracing. In *Proc. IEEE Pacific Visualization Symposium (PacificVis)*, pages 86–95, 2018.

[181] Jiang Zhang and Xiaoru Yuan. A survey of parallel particle tracing algorithms in flow visualization. *Journal of Visualization*, 21(3):351–368, 2018.