Advanced Visualization Techniques and Data Representations for Large Scale Scientific Data

By

JINRONG XIE
B.S. (Shanghai Jiao Tong University) 2007
M.S. (Zhejiang University) 2010

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Chair Kwan-Liu Ma

Bernd Hamann

Nelson Max

Committee in Charge

2016

To my family

For their endless love, support and encouragement

CONTENTS

List of Figures vi

List of Tables ix

Abstract x

Acknowledgments xii

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Statement 3

1.3 Contributions 4

2 Related Works **6**

2.1 In-situ and In-transit Visualization 6

2.2 Volume Rendering of Unstructured Grid Data 8

2.3 Parallel Volume Rendering 10

2.4 Data Reduction for Large Scale Data Visualization 10

3 Interactive Ray Casting of Geodesic Grids **12**

3.1 Background 13

3.1.1 Spherical Geodesic Grids 13

3.2 Visualization of Geodesic Grid Data 15

3.3 Ray Casting Framework 16

3.3.1 Grid Representation 16

3.3.2 Grid Traversal 19

3.3.3 Interpolation 20

3.3.4 Sampling and Integration 25

3.3.5 Gradient Estimation 26

3.4 Results and Discussion 27

3.4.1 Rendering Performance 28

3.4.2 Rendering Quality 28

3.5 Summary . . . 32

4 Visualizing Large 3D Geodesic Grid Data with Massively Distributed GPUs 34

4.1 Characterizing Distributed Geodesic Grid Visualization . . . 36

4.2 Parallel Volume Rendering Framework . . . 38

4.2.1 Spherical Quadtree-based Grid Partitioning . . . 41

4.2.2 Space-filling Curve-based Grid Distribution . . . 43

4.2.3 Ray Casting of Local Geodesic Grids . . . 45

4.2.4 Parallel Image Compositing . . . 46

4.3 Results and Discussion . . . 47

4.4 Summary . . . 53

5 Scalable Parallel Distance Field Construction for Large-scale Applications 54

5.1 Background . . . 56

5.1.1 Spatial and Temporal Coherence . . . 57

5.1.2 Octree-based Distance Field Construction . . . 58

5.2 Parallel Distance Tree . . . 59

5.2.1 Assumptions . . . 60

5.2.2 Data Structures . . . 61

5.2.3 Initial Coarse Global Distance Tree Construction . . . 63

5.2.4 Leaf Octants Assignment . . . 63

5.2.5 Communication Schedule . . . 65

5.2.6 Full-grown Local Distance Tree Construction . . . 66

5.2.7 Distance Field Computing . . . 68

5.2.8 Tree and Distance Field Updates . . . 69

5.2.9 Integration with Simulation . . . 71

5.2.10 Acceleration . . . 71

5.3 Results and Discussion . . . 72

5.3.1 Application Results . . . 73

5.3.2 Performance Evaluation . . . 80

5.3.3 Discussion 85
5.4 Summary 86

6 Fast Uncertainty-driven Large-scale Volume Feature Extraction on Desktop PCs 88
6.1 Methods 90
6.1.1 Overview 91
6.1.2 Supervoxel Generation 91
6.1.3 3D Supervoxel Clustering via SLIC 93
6.1.4 Hierarchical Merging 95
6.1.5 Uncertainty-based Refinement 97
6.2 Results and Discussion 97
6.2.1 Combustion Data 98
6.2.2 Ocean Data 101
6.2.3 Flow Data 102
6.2.4 Performance Results 103
6.2.5 Data Size Limits on the Desktop PC 107
6.2.6 User-driven Extraction v.s. Automatic Extraction 107
6.3 Summary 108

7 Summary 109

A Math Derivation of Equation 3.10 and 3.12 111

LIST OF FIGURES

2.1 Comparison among post-processing, in-situ processing, and in-transit processing 7

3.1 The proposed volume visualization of a global cloud resolving model (GCRM) dataset 12
3.2 The first iteration of subdivision to construct a geodesic grid 13
3.3 Duality of Voronoi polygon and Delaunay triangulation of geodesic grid 14
3.4 An illustration of the connectivity data table of the Voronoi cell mesh 18
3.5 An illustration of the connectivity data table of the dual triangular mesh 19
3.6 An illustration of the grid traversal 20
3.7 An illustration of the barycentric interpolation of the scalar value within a triangular frustum 23
3.8 A plot of a reconstructed scalar value using our analytic function 26
3.9 The result of ray casting of a synthetic spherical scalar field defined on a real geodesic grid with 10242 cells and 61 layers 29
3.10 The result of ray casting with lighting using the analytic ray integration and gradient estimation on the high resolution dataset 29
3.11 The rendering quality comparison of our approach with conventional tetrahedron based method and mean value interpolation (MVI) in a close-up view . . . 30
3.12 Performance measures and comparison with conventional tetrahedron based method and mean value interpolation method 31
3.13 The volume rendering of the whole global atmospheric vorticity variable using different visualization parameters 31

4.1 An example of spherical geodesic grids covering the ocean of the Earth surface 35
4.2 Ray-casting of scalar fields 37
4.3 Regular partitioning scheme v.s. our spherical quadtree based partitioning scheme 38
4.4 The major steps of our parallel volume rendering framework. 38
4.5 Illustration of the Ray-casting of a quadrant 40

4.6 Rendering of a quadtree constructed from a simulation dataset. 41
4.7 Spatial decomposition of a spherical surface and its corresponding quadtree . . 44
4.8 The timing results of the GCRM dataset 49
4.9 Timing results of the MPAS dataset 50
4.10 Ray-casting time for each GPU using the MPAS dataset and different data distribution schemes 51
4.11 Visualization results of the GCRM dataset 52
4.12 Visualization results of the MPAS dataset 52

5.1 Comparison of two isosurfaces between two consecutive time steps 57
5.2 Illustration of elements partition and distribution among five processors 60
5.3 Leaf octants with the computed locational key follow the *Z-order* space filling curve 64
5.4 Illustration of the full-grown local distance tree construction 67
5.5 A illustration of the case when Equation 5.5 is violated and searching the element list of the parent is required 70
5.6 An example of distance-based visual analytics of a combustion dataset 72
5.7 During a combustion simulation, the distance field is constructed based on an isosurface of the T variable. 75
5.8 Distance-based transfer function v.s. conventional value-based transfer fucntion 77
5.9 Volume rendering of the temperature field at different distances from the car . . 78
5.10 Isosurface rendering of the distance field generated from the Boeing 777 model 79
5.11 Scalability study using the second combustion dataset for post-processing on Intrepid 84

6.1 An overview of our system pipeline 90
6.2 SLIC clustering results on a combustion simulation dataset shown on a 2D slice. 92
6.3 The result of the first iteration after our 3D SLIC implementation on a combustion simulation dataset 93
6.4 An example of merging sets of clusters to construct a hierarchical partitioning . 95

6.5 A 2D illustration of running template scheme to detect neighboring clusters for every cluster 95
6.6 A 3D volume rendering of the combustion simulation data using a mixture ratio variable 98
6.7 Examples of applying the clustering scheme to the combustion dataset 99
6.8 Refining supervoxels with high uncertainty reveals the finer structure of the combustion flame 100
6.9 A 3D volume rendering of the clustering results on the ocean simulation dataset based on velocity magnitude 101
6.10 A 3D volume rendering of the clustering results based on the vorticity magnitude of the flow simulation dataset 102
6.11 A breakdown of the performance results for each of the extraction steps vs. supervoxel size for the combustion dataset. 104
6.12 A breakdown of the performance results for each of the extraction steps vs. supervoxel size for the ocean dataset. 104
6.13 In-situ timing results for parallel supervoxel generation vs. supervoxel size. . . 106
6.14 A breakdown of the performance results for each of the extraction steps vs. supervoxel size for the flow dataset. 106

LIST OF TABLES

3.1 The GCRM datasets for performance testing in the study 27

3.2 Performance comparison between CPU, single GPU and dual GPUs rendering of one frame 28

4.1 The GCRM and MPAS datasets used in our evaluation 47

5.1 The datasets used in our evaluation. 73

5.2 Timing breakdown for in-situ distance field construction with the combustion simulation on Hopper 81

5.3 Major execution time components and their corresponding operations. 83

5.4 Timing breakdown for computing the distance field of the Boeing 777 model on Hopper 85

6.1 Strong scaling performance testing 105

ABSTRACT

Advanced Visualization Techniques and Data Representations for Large Scale Scientific Data

Scientific simulations provide a critical means for understanding and predicting important natural phenomena, often having significant impact on policy-making and the environment's well-being on the regional and global scales. The output of a typical leading-edge simulation is so voluminous and complex that advanced visualization techniques are urgently needed to explore and interpret the computed results. The new challenges of visualizing large simulation data are mainly imposed by the fact that data are too massive for transferring, storing, and processing. The gap between data generation and scientific discovery is getting wider. A viable solution to bridge the disparity is based on the concept of in-situ processing that can greatly reduce data movement and storage requirements by coupling visualization with simulation. It thus requires designing and deploying new parallel visualization techniques on cutting-edge high performance systems characterized by heterogeneous processors, a high level of concurrency, and deep memory hierarchies.

This dissertation makes contributions to the design of new visualization and data representation techniques to facilitate large-scale visualization on highly parallel distributed systems. We carefully study novel data representations of large and complex simulation data, and explore corresponding data partitioning and distribution schemes to ensure the stability of a visualization system in a large heterogeneous computing environment. Another task of this research is to exploit intra-node and inter-node parallelism at a high level of concurrency to improve parallel efficiency of visualization algorithms. We also study the communication patterns and data access patterns of parallel visualization process, and evaluate and enhance our new data representations to minimize inter-node data exchange. Lastly, we pair these techniques with multi-resolution advantage of data abstraction guided by an uncertainty-driven approach to make it possible to realize scalable visualization solutions for large simulations.

We carry out the experimental study based on selected, representative simulations and corresponding applications, such as high-performance and high-quality visualization of climate

models and efficient data representations for the analysis of large-scale flow simulations. We demonstrate that well-designed visualization techniques and data representations for simulation data can facilitate more responsive and intuitive studies of visualization at large scale, and hence enhance scientists' potential to discover complex patterns and understand numerical simulations.

ACKNOWLEDGMENTS

This dissertation would not have been possible without the help of so many people in so many ways. First and foremost, I would like to express my sincere gratitude to my research advisor Dr. Kwan-Liu Ma, for his continuous support, patient guidance, and constructive advice of my PhD study. Over the last five years, I have been privileged to be a member of the Visualization and Interface Design Innovation (VIDI) group he assembled at UC Davis. It is a vibrant and stimulating environment where creativity, academic freedom, and collaboration keep continuing to thrive.

I'm truly fortunate to have a great mentor as well as my collaborator Dr. Hongfeng Yu, a former PhD student of the VIDI group and a postdoctoral researcher at Sandia National Laboratories, now an assistant professor at the University of Nebraska-Lincoln, whose pearls of wisdom and work ethic have always motivated me to conquer a series of challenges during the course of my research.

In addition, two committee members deserve special thanks: Professor Bernd Hamann and Professor Nelson Max. Bernd's engaging class, ECS 175 in Fall 2010, gave me the first glance of computer graphics, while Nelson's volume rendering class helped me lay a strong foundation for my subsequent research in a way that the mysterious math behind rendering fuzzy effects never looks fuzzy to me anymore. Overall, I'm immensely grateful to their insightful and constructive feedback that greatly improved this manuscript.

My sincere thanks also goes to Dr. Jishang Wei and Dr. Zeqian Shen, who provided me with opportunities to join their team as intern during the summer of 2014 and 2015, respectively, and who have introduced me to real-world problems in industry.

I feel grateful to have worked with a group of colleagues at UC Davis. Especially I thank my fellow lab mates Yubo Zhang for his stimulating discussion, and Franz Sauer for many sleepless nights we were working together before paper and contest deadlines, and for all the fun we have had in the last five years.

Last but not least, I wish to thank my dear family who have missed me during many years. Their enduring love, patience, and support give me endless courage for achieving my dreams.

Chapter 1

Introduction

1.1 Motivation

On November 14, 2008, Oak Ridge officials announced the world's first petaflop supercomputing system, Jaguar, available for open research. It marks for the first time that people have the tools to address some of the most challenging problems in science. Over the years, the developments of supercomputers around the world have transformed scientific research. Its rapid growth in computing power and storage capacity has revolutionized the simulation-based studies with extreme complexity and many new problems can now be addressed for the first time. However, the size of the output data resulting from such large-scale computation presents a great challenge to the subsequent data visualization and analysis tasks, thus leading to a growing gap between researcher's ability to simulate complex phenomena at high resolutions and their ability of knowledge discovery from the massive output datasets.

Visualization is a process for generating images, diagrams, or animations that convey structures, relationships, and trends hidden within data [91]. Through visual graphical representations, it is possible to effectively communicate information or knowledge by taking advantage of human's unparalleled visual perception of patterns and cognitive abilities. Nowadays, visualization has become an indispensable tool in science and engineering. In particular, it is an extremely important component of the scientific effort on facilitating the design of the next-generation climate models, and on advancing the understanding of complex natural phenomena. Advanced visualization is also essential for scientists to gain insight and understanding of phe-

nomena that are simulated based on large and complex numerical models involving multiple variables with different data types such as scalar, vector, and tensor. The ability to simultaneously visualize multiple variables is essential since, for instance, climate is intrinsically a multivariate problem domain, concerned with intricate interaction among the atmosphere, the biosphere, the cryosphere, the geosphere, and the oceans.

Even though visualization has proven to be an effective tool to deliver succinct summaries of large scientific data, it also faces great challenges. As numerical simulations and experimental studies keep producing enormous quantities of data, one of the main challenges for visualization researchers is to discover graphical models and algorithms that scale with the growth in data size and complexity [62]. For example, large scientific applications, such as climate simulations, earthquake modeling, and supernova thermonuclear reactions, can produce terascale or even petascale data that are beyond the capabilities of current visualization tools. Only with appropriate data analysis and visualization support can these scientific studies lead to revolutionary advances in science and engineering [41]. To bridge the gap between the ability to produce numerous data and the capability to analyze and visualize the data, many pioneers in the visualization community have been devoted to the study of high-performance in-situ visualization and data reduction techniques. For instance, as future computing moves to peta and exascale, the conventional workflow of visualization methods that relies on moving simulation output to a visualization machine for post-processing has become infeasible for interactive exploration and online monitoring. To address this challenge, in-situ visualization solutions integrate the analysis and visualization into the simulation pipeline and makes them run on the same supercomputer. This solution aims to capitalize the power of the supercomputer driving the simulation and also keep a minimum amount of simulation data being dumped to persistent storage for post-processing analysis and visualization. Scientists can visually explore their data as the simulation runs. However, this option is seldom adopted because most scientists were reluctant to spend their precious supercomputer time for visualization calculations. Moreover, the parallel simulation code and visualization code are not always compatible in terms of domain decomposition. It is very difficult to couple the two codes. Another promising direction to increase the interactivity of large scale visualization is using approaches such as multi-resolution

data representation, quantization, and transform-based compression to better manage the sheer data size. But these methods defeat the original purpose of adopting high-resolution simulations.

A Graphics Processing Unit, or GPU, has proven to be an affordable and effective device not only in graphics rendering, but also in a broader area of general purpose computing. A GPU can deliver enormous computational power while ensuring cost-effectiveness in space, power, and cooling demands. Even though a GPU cluster usually serves as a standalone post-processing machine for visualizing simulation data transferred from a supercomputer system, incorporating GPUs directly into supercomputers has become a growing trend and practice. For instance, since late 2012, Jaguar has been replaced by a new generation of supercomputer, Titan, with a large GPU installation as general purpose computing accelerators. The incorporation of GPUs adds a new level of parallelism to the system and makes Titan 10 times faster with the same space and power consumption as its predecessor [1]. Such an appealing trend is illuminating the future of data generation and analysis. The deployment of GPUs on supercomputers will continue to broaden the gap between the rates of data generation and storage I/O, making in-situ techniques stand out as a prominent solution to bridge the two. Although the GPU has catapulted high-performance computing, it gives rise to a new set of challenges in terms of application development, job scheduling, and resource management. In particular, previous research on in-situ processing was mostly done via CPU-only supercomputers. Fully utilizing the power of a distributed GPU environment requires a thorough study of data decomposition and management, load balancing between simulation and visualization, interplay between multi-core CPUs and GPUs, and inter-node and intra-node communication.

1.2 Problem Statement

We have entered the Big Data era, where the future generation of supercomputing environment is characterized by heterogeneous architecture, deep memory hierarchy, and high level concurrency. As powerful supercomputers keep generating a deluge of data at a fast pace, I/O devices and data processing tools for moving and exploring the data do not keep up with the data growth. To address the large data challenge, advanced visualization techniques and data

representations are required to be adaptive to the underlying architectures and to exploit data locality for efficient memory usage across heterogeneous computing environments. The advancements are expected to be the basis for building scalable software in the next generation computing environment and meeting more sophisticated demands from numerical applications.

These considerations motivate us to develop advanced visualization strategies at extreme scale that aim to enable efficient exploration of large data and optimal data management. Such strategies consider the following aspects:

- Study and develop data representations that are adaptive to the underlying distributed contexts and deliver efficient memory usage by sharing and reusing data among multiple computing environments while reducing communication costs.

- Investigate how and to what extend efficient data reduction, compression, and indexing schemes can be applied to decrease the amount of data for delivery, processing and visualization while preserving to a large extent the essential information in the original data.

- Study and evaluate highly scalable parallel visualization techniques. As commodity technologies can deliver larger systems in a more cost-effective way for computation, rendering, and display, our visualization systems must scale well with these systems to handle increased data loads by using parallel rendering techniques on multi-core CPU and GPU computers.

1.3 Contributions

This dissertation has introduced and discussed several advanced visualization techniques and data management schemes with efforts to bridge the gap between ever increasing computing power and comparably slow I/O speed, providing responsive interactive visualization of large datasets in scientific computing. The study focuses on exploiting communication patterns, data orgnization, data indexing, and multi-resolution schemes to achieve an optimal visualization and analysis performance. The following contributions have been made:

- Chapter 3. A memory efficient ray-casting scheme for visualization of unstructured geodesic grid data using GPU acceleration [97]. It efficiently reuses the original data

representation generated by the simulations and provides an efficient and precise analytic solution for scalar and gradient interpolation, achieving smooth shading and salient depth cues. This step is critical for achieving in-situ visualization of geodesic grid data.

- Chapter 4. A study extending the ray-casting scheme to efficiently perform parallel visualization of geodesic grid (See section 3.1.1 for definition) data on a distributed GPU cluster while keeping the communication overhead and memory footprint at a minimum and avoiding memory resource contention between the simulation and visualization [98].

- Chapter 5. A scalable distributed data structure for massive data indexing on parallel machines [103]. This particular work includes a new method for efficient indexing and its application to a scalable distance transform of time-varying data using parallel machines. A parallel spatial data structure is designed to manage the level sets of the data for facilitating surface tracking over time, hence significantly reducing the computation and communication cost for calculating the distance to the surfaces of interest from any spatial location.

- Chapter 6. A supervoxel based multi-resolution data representation technique for large-scale volume feature extraction [96]. It is helpful to encode the uncertainty information into the introduced data structure. Based on the data representation, a hybrid feature extraction technique is designed with multiple GPUs acceleration for large-scale data exploration.

Chapter 2

Related Works

2.1 In-situ and In-transit Visualization

HPC simulations generate a deluge of data that are conventionally loaded offline by post-analysis and/or visualization tools. Extrapolating current technology trends towards the exascale computing reveals the increasing disparity between I/O speed and compute speed that hinders fast data exploration and knowledge discovery [70]. For example, scientists can only store a small fraction of the data during their detailed modeling and simulation processes, thus possibly missing highly intermittent transient phenomena. Moreover, they also have to wait a significant amount of time before data is transferred to the end machine, processed, visualized and finally displayed on the monitor. There has been a growing trend toward coupling simulation with analysis and visualization. In other words, instead of moving the data to the visualization device, a more viable solution is to move the visualization algorithm back to the data source. This concurrent analysis framework is generally referred to as *in-situ visualization* as illustrated in Figure 2.1. It provides the advantage of bypassing slow I/O and generates results faster. It also enables users to steer their simulations as they run.

Two variants of this framework are *in-situ* and *in-transit* data processing paradigms. In-situ visualization is more strictly considered as a tightly coupled framework, where researchers directly integrate operations, such as visualization [78], statistical compression and queries[30], into simulation routines to operate on in-memory simulation data. In contrast, in-transit approach utilizes dedicated resources, *staging nodes*, to run analysis and visualization codes. The

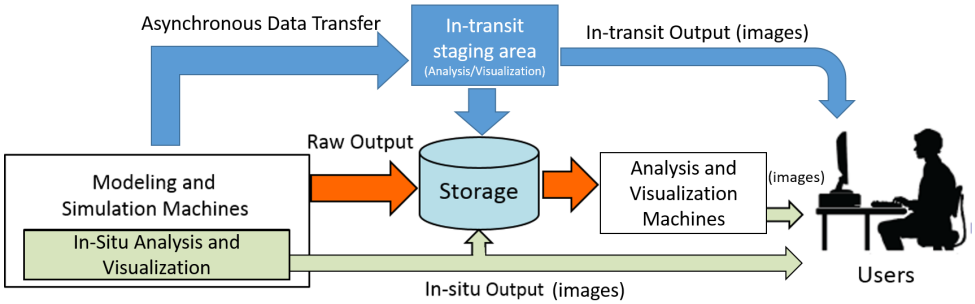


Figure 2.1. Comparing post-processing, in-situ processing, and in-transit processing. Post-processing (red arrows) involves storing and transferring tera- or peta-byte raw data through slow disk I/O, making it unsustainable for analysis and visualization. In-situ processing (green arrows) provides a highly scalable solution for dealing with large-scale simulations data. In-transit processing (blue arrows) allows data being buffered in the staging area and then write to disk. Asynchronous data transfer enables simulations to continue running while data being transferred to staging area for analysis and visualization and then being written to disk if necessary.

staging nodes can be separate compute nodes in the same supercomputer as the simulation, or in a remote site. Extensive research has been done in data staging [3, 104, 17, 2, 16, 80, 53]. Many of the solutions target how to achieve fast data movement asynchronously from simulation site to reduce the impact of expensive I/O operations. However, these approaches are subject to limited data operations within the staging area, such as pre-processing and transformation. Thus many of the staging nodes are under utilized.

Both in-situ and in-transit processing paradigms share the same idea that is to perform the analysis (e.g., visualization) as the data generation is running, and then save the results (e.g., images) that are orders of magnitude smaller than the raw data. Such approaches can significantly minimize the I/O cost and lead to exploration of data in full scale. In-situ and in-transit paradigms differ from where and how the analysis is performed. In-situ analysis typically happens right on the same computing resource as where the simulation runs. On the contrary, in-transit transfers part of the data being generated to different processors. Those processors can be on the same computing facility or on entirely different machines.

It's hard to tell which one is better. In-situ analysis almost entirely removes the necessity for data movement. However, it has to compete for the local computing and memory resources with the simulation code. Since most simulations are memory bound with limited allocation of

computing time, it incurs a significant strain on the resource usage available to in-situ analysis. In-transit analysis, on the contrary, has minimum impact on the simulation because it asynchronously transfers computation to the secondary resource. But in practice, the overhead of moving a small subset of the original data over the network can be expensive, and moreover, the memory and computing resource on secondary machines can easily be used up. Both variants require analysis algorithm to scale as well as simulations when executed with thousands to hundreds of thousands of CPU cores.

Bennett et al. [9] have introduced a hybrid approach that combines in-situ and in-transit processing. The key insight is based on the fact that the most analysis algorithms can be decomposed into two stages: a highly efficient and massively parallel in-situ stage, and a small-scale parallel or serial in-transit stage connected through a transparent data staging framework.

2.2 Volume Rendering of Unstructured Grid Data

To study physical problems, many scientific simulations produce multiple 3D datasets. Without volume visualization [47], it is almost impossible to explore and understand the content of the data in a holistic view. Therefore, an efficient volume rendering technique is the key to support scientific exploration and knowledge discovery and is the basis to scale up to parallel visualization for handling large-scale dataset. Among many different kinds of simulation grid, the unstructured grid is one of the most popular grids and is widely adopted in scientific simulation due to its flexibility and adaptability. Volume rendering of unstructured grids are commonly based on either cell-projection or ray-casting. Shirley et al. [73] introduced one of the first cell-projection algorithms, Projected Tetrahedra, in 1990. Since it needs visibility ordering of the cells, this algorithm can be very expensive for a large amount of cells. A set of optimization techniques has been developed to accelerate sorting [94, 75, 29] or avoid sorting [86]. Ray-casting is another popular method for rendering unstructured grids [22]. It avoids the costly visibility sorting; however, the mesh connectivity information is usually needed for mesh traversal. Various approaches have been presented to accelerate ray-casting of tetrahedral grids using graphics hardware [85, 87, 49], with a particular focus on the representations of connectivity information for memory efficiency.

Cell-projection and ray-casting methods have been extended to render more general polyhedral grids. For example, Bennett et al. [8] presented a parallel implementation of cell-projection for meshes with arbitrary polyhedra. Muigg et al. [55] presented a hybrid scheme for rendering unstructured grids based on importance classification. Muigg et al. [54] further introduced a new unstructured-grid representation which can minimize redundant connectivity information and improve grid traversal performance in graphics hardware.

Substantial efforts have been made to generate high-quality volume rendering of unstructured grids. Max et al. [48] showed that with barycentric interpolation for signal reconstruction inside a tetrahedron, integration using sampling at the cell faces can generate exact rendering results. Röttger et al. [66] applied this idea to Projected Tetrahedra, and developed a pre-integrated volume rendering method to render isosurfaces using 3D texture hardware. Röttger et al. [65] further enhanced the pre-integration technique using 2D texture hardware and increased the performance of volume rendering. Moreland et al. [52] presented partial pre-integration to approximate the volume rendering integral with computational feasibility for tetrahedral grids. Lum et al. [40] improved the pre-integration technique by speeding up lookup table generation and minimizing lighting artifacts. However, Marchesin et al. [45] showed that pre-integration is less suitable for volume rendering of AMR data with hexahedral cells. They derived an analytical function for signal reconstruction inside a hexahedron, and used the gradient of the function for shading. Similar to their idea Xie et al. [97] extended their approach to analytically compute scalar and gradient within a spherical prism which is a constituent cell of a geodesic grid. Such a grid has become increasingly prevalent in the development of large-scale climate models [11, 63]. While it is possible to first convert geodesic grid data into conventional representations (such as tetrahedral grid) and then apply suitable algorithms (such as cell-projection [73] or ray-casting [22]) to visualize 3D scalar fields, this solution requires computation and storage overhead that can be prohibitively high for current tera/petascale and future exascale simulations. The approach by Xie et al. [97] directly visualizes 3D scalar fields defined on raw geodesic grids without data transformation and achieves interactive rendering rates with a single GPU. This technique is described in details in Chapter 3. However, these methods are limited on a single machine where the entire dataset should fit in the GPU memory of a single

desktop PC.

2.3 Parallel Volume Rendering

When data size is larger than the available memory capacity of a single machine, a common strategy is to use multiple machines to perform visualization in parallel. Researchers have developed various parallel visualization algorithms for large data. One of the main focuses is to achieve high-quality rendering while maintaining scalable performance with an increasing number of processors. Ma et al. [42] developed a parallel cell-projection algorithm for 3D unstructured data from aerodynamics applications. They used a round robin distribution of data cells to achieve an effective static load balancing among the processors. Parker et al. [60] conducted ray tracing using shared-memory multiprocessor machines. Their method enables an interactive isosurface visualization of large volume data with high parallel efficiency. Leaf et al. [32] developed a parallel volume rendering algorithm for large-scale Adaptive Mesh Refinement (AMR) data. They partitioned AMR data into convexly-bounded chunks and distributed them among multiple GPUs using static load balancing to perform distributed ray-casting. Xie et al. [98] designed a new data partitioning and distribution scheme that employs a spherical quadtree to decompose and index the multiresolution spherical grids for accurate rendering load estimation and balanced work load partition. This method is presented in Chapter 4. Recent efforts have been made to further improve the scalability of parallel visualization to be on par with simulations, thus enabling in-situ visualization to address future exascale supercomputing challenges [70]. Researchers have directly integrated visualization [100, 19] into simulation routines to operate on in-memory simulation data. Xie et al. [98] also take in-situ visualization into account such that it is possible to directly process the original simulation grid data and achieve high parallel efficiency comparable to simulations.

2.4 Data Reduction for Large Scale Data Visualization

While scalable visualization techniques are tackling the large-scale data problem at one end, data reduction techniques are striking the problem at the other end. Data reduction of volume data is an effective approach to minimize the I/O bandwidth cost and has been studied extensively. Shen et al. [72] has described a differential volume rendering approach to compress the

data and speed up volume animation. The key insight is based on the fact that temporal coherence exists between consecutive time steps. Shen et al. [71] has further proposed a solution to managing large scale time-varying data using a time-space partitioning (TSP) tree that keeps track of data's spatial and temporal coherence. Both approaches treat the spatial and temporal information separately. In contrast, Wilhelms et al. [93] treat spatial and temporal dimensions uniformly by encoding time-varying dataset using 4D tree.

Transform-based compression and rendering scheme points to another direction. Lum et al. [39] has proposed an approach that uses the discrete cosine transform to encode an individual voxel in the temporal domain. A color table animation technique is employed in their approach for volume rendering using hardware acceleration. Guthe et al. [24] introduce an algorithm that uses the wavelet transforms to encode each spatial volume. Then they apply a motion compensation strategy to match the volume blocks in adjacent time steps. Sohn et al. [74] use wavelet transforms in their compression scheme to generate intracoded volumes and use difference encoding for time sequence compression. A vector quantization approach was used by both Schneider et al. [69] and Fout et al. [20] to compress time-varying volumetric data. The distance transform, or distance field, can also serve for indexing and compression to manage and explore data at extreme resolutions. Laney et al. [31] utilize a multiresolution distance volume for progressive surface compression. Wang et al. [81] employ a distance volume for encoding the importance value of original voxels. Unlike the TSP tree approach that uses a binary tree for storing temporal information, they merge the octree nodes that share temporal coherence in consecutive time steps into space-time blocks. Recently, Yu et al. [103] designed a distributed spatial hierarchical data structure, named a *parallel distance tree*, to support scalable parallel distance transform of time-varying data using distributed parallel machines. The detail of this method is presented in chapter 5. In parallel with Yu, Xie et al. [96] introduced a supervoxel based technique for compact volume representation in support of large-scale volume feature extraction in situ. The method encodes the uncertainty information in the supervoxel, based on which a hybrid feature extraction technique is designed with multiple GPUs acceleration. The technique is covered in chapter 6.

Chapter 3

Interactive Ray Casting of Geodesic Grids

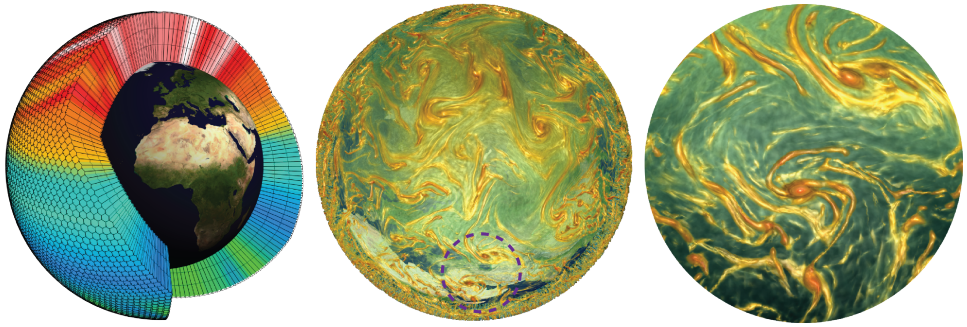


Figure 3.1. The proposed volume visualization of a global cloud resolving model (GCRM) dataset. The left image shows the geodesic grid mesh. The middle image shows the volume rendering of the global atmospheric vorticity variable defined on the geodesic grid. The right image is a close-up view of a selected region of interest in the middle image, succinctly revealing the fine features of eddies with strong vortical structures.

Climatological and environmental studies are critical for understanding and predicting important natural phenomena, such as glacier shrinkage and sea-level rise. Climate research has been internationally recognized as one of the most crucial missions with the largest investment for computational and mathematical analysis. For example, the U.S. Department of Energy aims to enable exascale computing capabilities to model climate at a resolution of roughly 10 kilometers over the entire Earth [36]. By leveraging the unprecedented power of the supercomputer, scientists can simulate the highest-resolution climate/Earth system models, leading to more reliable numerical weather prediction.

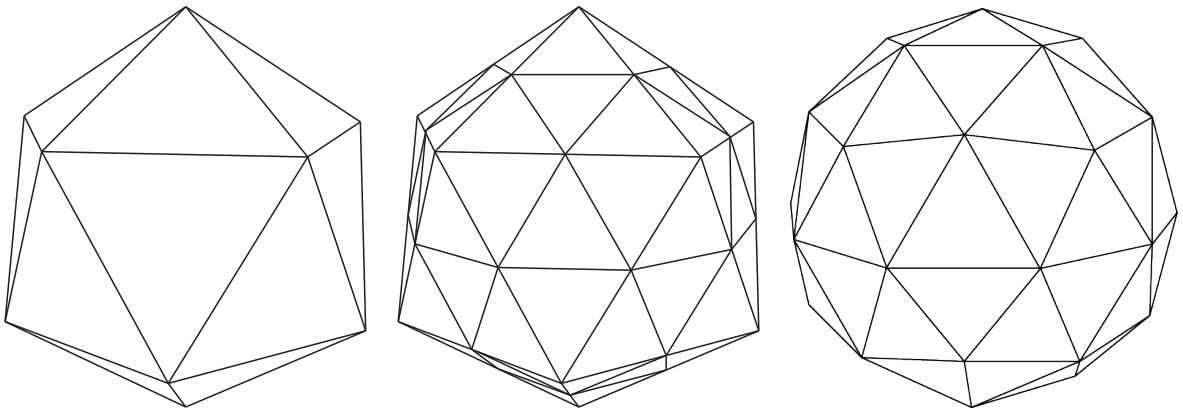


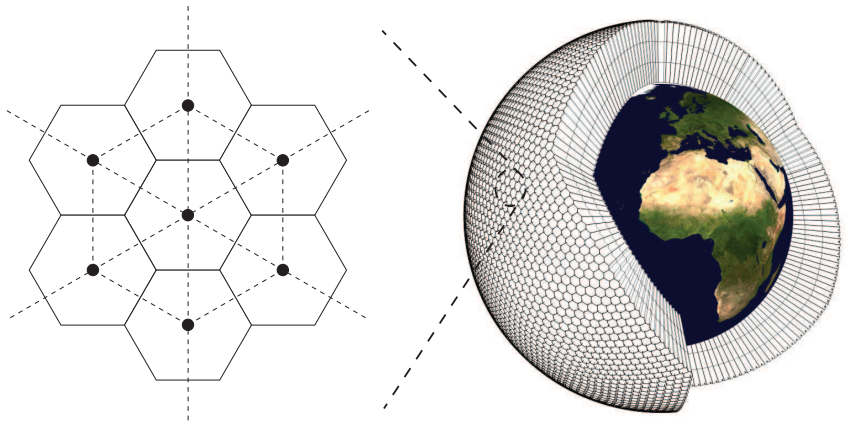
Figure 3.2. The first iteration of subdivision to construct a geodesic grid. (a) shows the initial icosahedron with 12 vertices, 30 edges, and 20 equal triangles. (b) shows each triangle is subdivided through bisection. (c) shows the new vertices are popped out onto the sphere.

3.1 Background

3.1.1 Spherical Geodesic Grids

Geodesic grids, first introduced by Williamson [95] and Sadourny et al. [67] for meteorological applications in 1968, have been rediscovered widely to model the Earth's surface for various applications of climate modeling in the recent years [63, 5, 88, 11]. The advantage of geodesic grids is particularly evident by their suitability for large-scale numerical simulations [44, 50]. The isotropic structure of geodesic grids can completely avoid the well-known Pole problem and the expensive filtering operations present in conventional latitude-longitude grids (e.g., curvilinear hexahedral grids for spherical shells). The hexagonal-triangular duality of geodesic grids allows scientists to design discrete operators and upgrade scales to higher resolutions in a more flexible manner. In addition, the data structure of geodesic grids is extremely well suited for scientists to parallelize numerical solvers and achieve high efficiency on large distributed memory parallel computers.

The construction of a geodesic grid typically begins with an icosahedron inside a sphere, as shown in Figure 3.2 (a). We can then subdivide the icosahedron to generate the refined grid by iteratively applying a bisection operation on the triangles [63], as shown in Figure 3.2 (b) and (c). This subdivision process can be repeated until the desired resolution is obtained. Each grid



(a) (b)

Figure 3.3. (a) shows the duality of Voronoi polygon and Delaunay triangulation. (b) shows the structure of a 3D spherical geodesic grid.

point has six nearest neighbors, except for the original twelve icosahedral points where each point only has five neighbors. These twelve special points are referred to as *pentagonal points*. The triangles define a Delaunay grid, and we can further join the centroids of neighboring triangles to construct Voronoi polygons. As shown in Figure 3.3 (a), the dashed lines represent the Delaunay triangles, and the solid lines represent the Voronoi polygons. The Voronoi polygons are pentagons with respect to the pentagonal points, and are hexagons with respect to the other points. For a triangular grid, each grid cell has some neighbors across the cell walls (referred to as *wall neighbors*) whereas others across the cell vertices (referred to as *vertex neighbors*). The different types of neighbors incur unavoidable asymmetries for the finite-difference operators on triangular grids. Fortunately, for the grid constructed from Voronoi polygons, each cell's neighbors are all wall neighbors, thus allowing the finite-difference operators to treat all neighboring cells in the same way and become as symmetrical and isotropic as possible. Therefore, a geodesic grid defined on a spherical surface is composed of Voronoi polygons, which contains the information of each polygon's vertices, edges, and center point. The data values are stored at the center points of polygonal cells. To model ocean or atmosphere in 3D, the spherical polygon mesh is scaled and duplicated to construct a set of spherical layers parallel to the Earth's surface, and each layer corresponds to a different height above the Earth's surface, as shown in Figure 3.3 (b).

3.2 Visualization of Geodesic Grid Data

Although a variety of sophisticated visualization algorithms have been designed for structured and unstructured grids, comprehensive work on visualization of geodesic grids is surprisingly poorly covered in the literature, possibly due to the relatively short history of large-scale applications of this grid type in practice. With the fast growth of geodesic grid applications, new techniques and tools are highly desirable to enable effective analysis and visualization of high-resolution data generated from large climate simulations. However, visualization of large geodesic grid data imposes some unique challenges.

First, the data structure of geodesic grids is constructed using a recursive refinement procedure on a spherical surface, which presents very different geometry properties from other existing unstructured grids. Therefore, it is usually not applicable to use conventional tools to visualize geodesic grid data directly without any remeshing operations. Second, even though it is possible to transform geodesic grids into more generally supported grids, such as tetrahedral grids, for visualization, this approach often incurs significant computing and storage overhead, and it becomes infeasible to process large data from current tera/petascale and future exascale simulations. This chapter introduces a viable solution to generate high-quality visualization of geodesic grid data in full extent at interactive rates. This is achieved by minimizing the data movement and by preprocessing before carrying out the visualization operation. This solution has the potential to support in-situ visualization with simulations on geodesic grids. It will enable scientists to monitor simulations and possibly capture important features during the simulation runs. Moreover, exascale computers are expected to be mostly heterogeneous systems with specialized processors, deep memory hierarchies, and high levels of concurrency [70]. Care must be taken to consider the easy applicability of the introduced solution to such platforms.

To the best of our knowledge, this is the first design and implementation of interactive ray casting of geodesic grid volume data with almost no overhead. The following highlights a list of the contributions in this chapter:

- A novel scheme is presented to traverse rays in an efficient way without reconstructing full connectivity information in 3D, and thus avoid memory and computing overheads.

- An analytic solution is derived for the interpolation of scalar values along a ray within a geodesic grid cell, which achieves high quality rendering with adaptive sampling (see Figure 3.1).

- The geodesic grid is represented in GPU memory in a form that best matches the original storage format, hence minimizes the data transformation overhead.

Two datasets of real-world climate simulations are used to evaluate the approach. The proposed method can generate high-quality visualization of full resolution geodesic grid data, and allows scientists to navigate in great detail at an interactive or nearly interactive rate. Moreover, it directly takes original data as input, allowing for sharing data structures with simulations and for performing visualization during simulation time on supercomputers.

3.3 Ray Casting Framework

A simple approach to visualizing geodesic grids is to first decompose a cell into a set of tetrahedra, and then to render the tetrahedral grids using conventional techniques and tools like the ones discussed in section 2.2. However, this simple approach often incurs significant computing and storage overhead for large data. While it is also possible to treat geodesic grids as polyhedral grids, the scheme either requires visibility ordering of all cells or constructing 3D connectivity information for mesh traversal, and incurs considerable overhead as well. Alternatively, we develop an approach that directly takes geodesic grids as input, performs mesh traversal without constructing full 3D connectivity information, and thus can significantly reduce memory footprint and improve computing efficiency. Similar to Marchesin et al. [45], we also developed an analytic solution to compute the ray integral within grid cells and generate high-quality renderings of geodesic grid data.

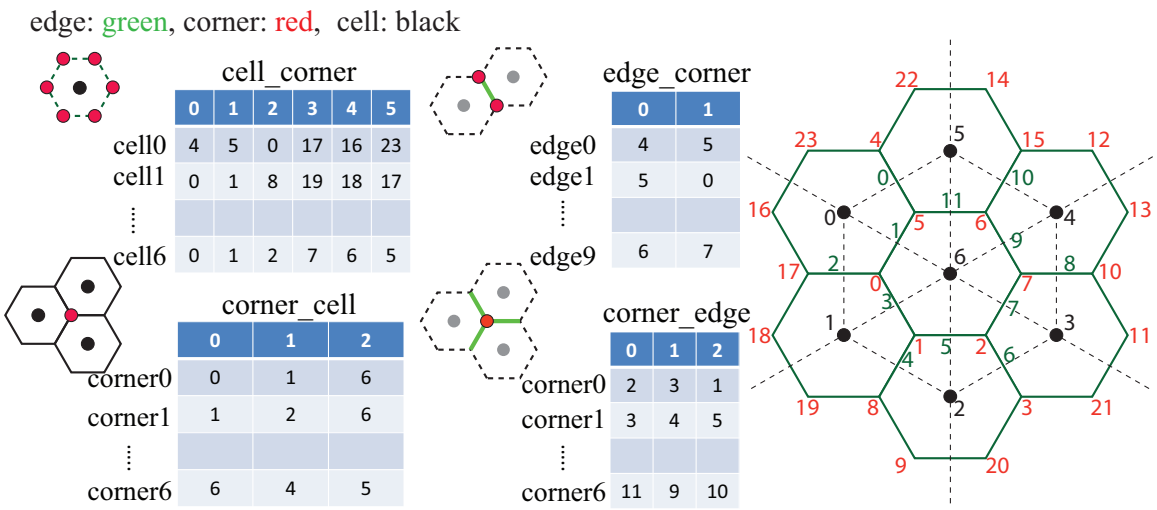
3.3.1 Grid Representation

A viable grid representation is essential for efficient mesh traversal. A typical representation organizes mesh connectivity information with a focus on memory efficiency, which is the key for the overall ray-casting performance. For a general geodesic grid dataset, because each

layer has the same Voronoi polygonal mesh structure as the outer spherical surface, scientists only save the 2D connectivity information of the outer surface in data. Intuitively, we can simply construct the 3D cells of hexagonal or pentagonal frusta by connecting the corresponding vertices on every two neighboring spherical layers. However, this approach needs considerable computing and memory consumption.

Our design is based on two important properties of geodesic grids. First, a geodesic grid has the duality of Voronoi-Delaunay, and the data values are stored at the center points of hexagonal or pentagonal cells. To compute the data value at any other spatial location, first we need to identify the triangular frustum that contains the sample location, and use interpolation to compute the needed value according to the values stored at the frustum vertices. We observe that although triangular grid connectivity information is not directly provided, the Voronoi-Delaunay duality allows us to construct and identify triangular frusta from the Voronoi polygonal grids without explicitly reconstructing the full triangular grids. Second, a geodesic grid is also radially symmetric. This means if the side edges of an arbitrary frustum cell are extended along the direction of radius, they all converge on the sphere center. Given this property, we can implicitly construct an intermediate 3D frustum cell during mesh traversal according to the 2D mesh structure of the outer surface and the depth values of layers. We utilize these tables:

- **center** table: Each entry of the table contains the longitude and latitude values of a cell center.
- **corner** table: Each entry of the table contains the longitude and latitude values of a corner.
- **cell_corner** table: Each entry of the table contains six indices of cell corners in the **corner** table with respect to a cell. If two indices share a same value, this entry indicates a pentagonal cell.
- **corner_cell** table: Each entry of the table contains three indices of the cells in the **cell_corner** table with respect to a corner, and these three cells share the corner.
- **edge_corner** table: Each entry of the table contains two indices of the corners in the **corner_cell** table with respect to an edge, and these two corners share the edge.

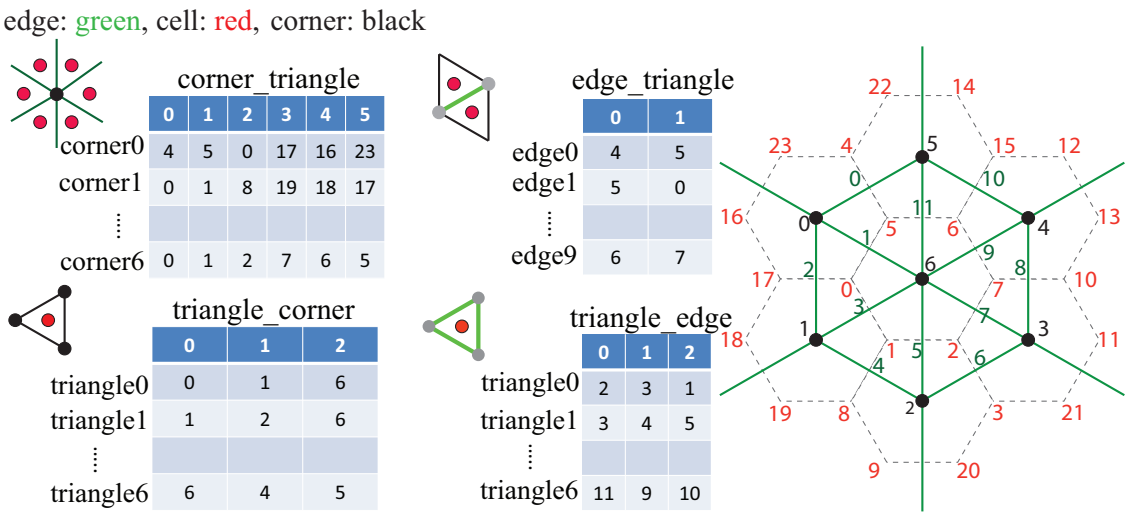


(a)

Figure 3.4. The *cell_corner*, *corner_edge*, *corner_cell*, and *edge_corner* tables used to represent a 2D Voronoi polygonal grid of climate simulations.

- **corner_edge** table: Each entry of the table contains three indices of the edges in the **edge_corner** table with respect to a corner, and these three edges share the corner.

Figure 3.4 illustrates the tables of several cells. This simple representation matches the storage pattern of raw geodesic grid data [63, 59], and thus facilitates fetching and organizing a geodesic grid into GPU memory with minimal computing, memory and implementation overhead. With this representation, we can implicitly obtain the dual Delaunay triangulation to meet the data interpolation requirement and perform ray traversal in an efficient and effective manner. First, as shown in Figure 3.5, we note that each triangle is uniquely identified by a hexagonal cell corner, which means the ith hexagonal cell corner uniquely corresponds to the ith triangle. Second, each triangle's vertices are uniquely identified by the centers of three neighboring hexagonal cells, and these three cells share the corner identifying the triangle. For example, the three indices in **corner_cell**[i] entry can be treated as three vertex indices of the ith triangle. Third, querying **corner_edge**[i] gives the three edge indices of the ith triangle. Then from these three edge indices, we can quickly identify the neighboring triangles by querying the **edge_corner** table.



(b)

Figure 3.5. The representation of dual triangular mesh of Figure 3.4. We note that the names of tables and table entries are modified for an illustration purpose, and they are not changed in the real implementation. Thus, from Figure 3.4 to 3.5, the content of the four tables keeps intact, which implies that no explicit grid transformation is required.

3.3.2 Grid Traversal

Algorithm 1 illustrates the overall framework of traversing the grid and performing ray integration frustum by frustum. Note that our current ray-casting scheme does not consider the case where the ray origin is inside the grid. As shown in Figure 3.6 (a), given a ray r, we first find its first intersection point, $\mathbf{p}_1$, with the grid. We use the conventional image-based technique [85] to find the first intersected triangle $\triangle_i$ on the spherical surface. By taking advantage of the radially symmetric property, we can easily construct a 3D triangular frustum on the fly, as the yellow frustum c_1 shown in Figure 3.6 (a). Next we compute the intersection point using the function *RayIntersectFace* that is a trivial ray-triangle intersection test. For a quadrilateral of a frustum, it is trivial to decompose it into two triangles. We assume that the intersection point is $\mathbf{p}_2$, where the ray leaves the frustum c_1, on the bottom face of c_1. In this case, the ray enters into the green frustum c_2 in the next layer, and c_2 is still constructed on the fly based on the triangle $\triangle_i$. We compute the next intersection point $\mathbf{p}_3$, and assume $\mathbf{p}_3$ is on a side face f_1 of c_2. A side face of any frustum uniquely corresponds to an edge of the triangle that is used to construct this frustum. We assume that in this case, f_1 corresponds to the edge e_j of the triangle

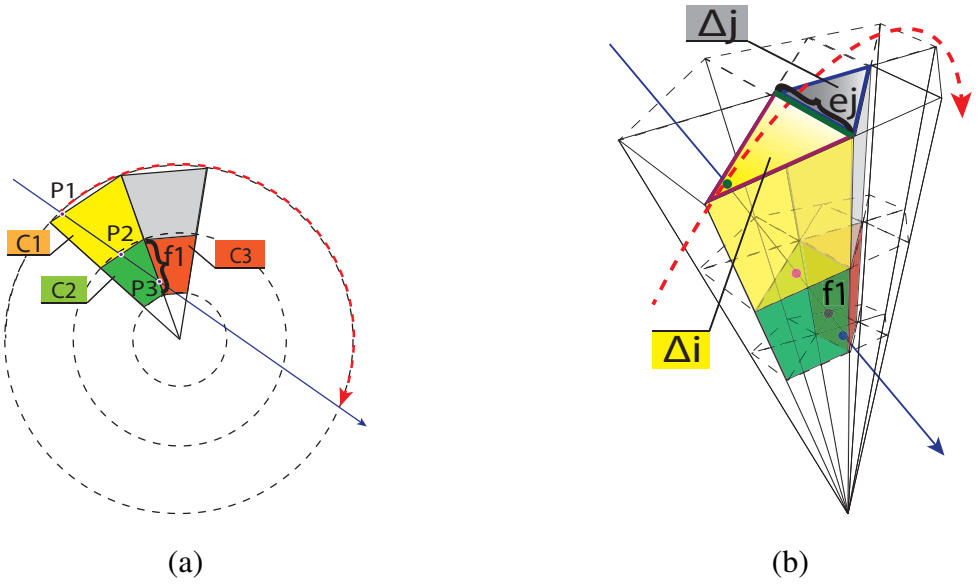


Figure 3.6. Grid traversal. (a) Side View. The yellow, green, and red frusta are constructed on the fly with marching the ray through the grid. The intersection of the ray and a temporary frustum is used to identify the next neighboring triangle on the spherical surface. (b) Perspective View. The red dashed curve is the projection of the ray on the spherical surface. The triangles $\triangle_i$ and $\triangle_j$ are identified based on the surface connectivity information, and are used to construct the corresponding frusta at the different layers.

$\triangle_i$, as shown in Figure 3.6 (b). By querying the **edge_corner** table, we can quickly find the neighboring triangle $\triangle_j$ on the spherical surface, and construct the next frustum c_3, as the red one shown in Figure 3.6. Algorithm 2 provides the details for finding the next neighboring triangle. We repeat this traversal process until the ray leaves the domain. We note that this process only uses the 2D connectivity information on the outer spherical surface, and constructs the 3D triangular frustum intersected with a ray on the fly. Each ray corresponds to a projected curve on the spherical surface, as the red dashed curve shown in Figure 3.6 (b). A temporary frustum is essentially used to find the intersection edges of surface triangles with the projected ray on the spherical surface, thus enabling grid traversal without explicit 3D connectivity information.

3.3.3 Interpolation

The geodesic grid in our study is composed of polygonal cells, and the scalar values are cell centered. The signal reconstruction for such grids can be handled using conventional piecewise-constant sampling techniques. However, if smoothed results are preferred, the scalar value at

Algorithm 1 Ray-casting geodesic grids

for each pixel on the projected visible surface **parallel do**
  Spawn a ray r from the camera
  Obtain the current triangle index $\triangle$ hit by r
  $cl = 0$ //Initialize current layer index cl with the outer layer index 0
  $ef = -1$ //Initialize r's exit face index ef of cell c with -1
  $\lambda_{min} = \infty$, $\lambda_{max} = -1$ //Entry and exit position parameters on the ray
  while r is within the domain of the grid **do**
    Construct a frustum c according to $\triangle$ and cl
    for each face j of c **do**
      //Loop through all faces of the frustum
      $\lambda = RayIntersectFace(r, c, j, el)$
      if $\lambda_{min} > \lambda$ **then**
        $\lambda_{min} = \lambda$
      end if
      if $\lambda_{max} < \lambda$ **then**
        $\lambda_{max} = \lambda$, $ef = j$
      end if
    end for
    $VolumeIntegration(\lambda_{min}, \lambda_{max}, c)$ //See section 3.3.4
    $FindNextTriangle(c, ef, cl, \triangle)$
  end while
end for

any spatial location inside a cell can be interpolated with the vertex-centered data [54]. In our study, given the Voronoi-Delaunay duality, we have the dual Delaunay triangulation where the scalar values are defined on the dual grid vertices. Therefore, the scalar and gradient values can be computed within triangular frustum cells.

We use barycentric interpolation for computing scalar values within a frustum as shown in Figure 3.7. Let the point $\mathbf{m}_3 = (x, y, z)$ be the sample point whose scalar value needs to be

Algorithm 2 $FindNextTriangle(frustum, exitFaceID, currentLayerID, currentTriangleID)$

```
maxLayerID := the maximum number of layers in the grid
if exitFaceID is the bottom face of frustum then
    if currentLayerID ≥ maxLayerID then
        // hit the deepest layer
        terminate the traversal of the current ray
    else
        currentLayerID++
    end if
else if exitFaceID is the top face of frustum then
    if currentLayerID ≤ 0 then
        // hit the outer layer
        Terminate the traversal of the current ray
    else
        currentLayerID--
    end if
else
    // Ray hits a side face
    Find the edge index edgeID according to exitFaceID in frustum
    if currentTriangleID ≠ edge_corner[edgeID][0] then
        currentTriangleID = edge_corner[edgeID][0]
    else
        currentTriangleID = edge_corner[edgeID][1]
    end if
end if
```

interpolated. In the frustum containing $\mathbf{m}_3$, we denote the vertices of top triangle by $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, and the vertices of bottom triangle by $\mathbf{p}_4$, $\mathbf{p}_5$, $\mathbf{p}_6$. Given the radially symmetric property of geodesic grids, we generate an interpolation ray shooting from the sphere center $\mathbf{o}$ through $\mathbf{m}_3$. This interpolation ray intersects with the top and bottom triangles at the points $\mathbf{m}_1$ and $\mathbf{m}_2$,

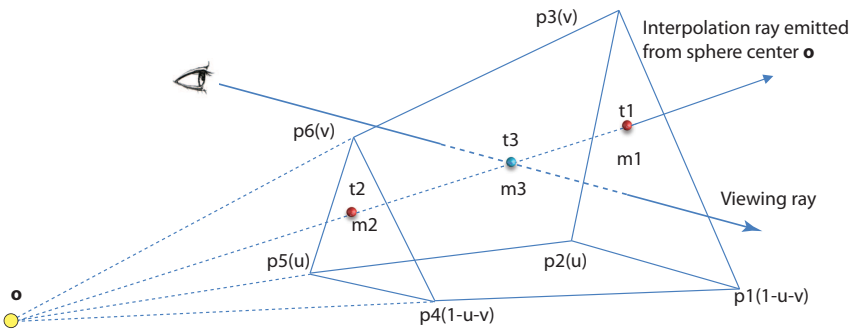


Figure 3.7. Barycentric interpolation of the scalar value within a triangular frustum. We first interpolate the scalar values at the points $\mathbf{m}_1$ and $\mathbf{m}_2$ on the top and bottom triangles, respectively, and then use linear interpolation to compute the scalar value at the point $\mathbf{m}_3$ on the line segment $\mathbf{m}_1\mathbf{m}_2$.

respectively.

In the first step, we use barycentric interpolation to compute the scalar value at $\mathbf{m}_1$ on the top triangle $\triangle p_1 p_2 p_3$. The coordinates of $\mathbf{m}_1$ can be expressed in barycentric coordinates:

$$\mathbf{m}_1 = (1-u_1-v_1)\mathbf{p}_1 + u_1\mathbf{p}_2 + v_1\mathbf{p}_3 \tag{3.1}$$

where u_1 and v_1 are the barycentric coordinates of $\mathbf{m}_1$. Let us denote the direction vector

$$\mathbf{d} = \frac{\mathbf{m}_3 - \mathbf{o}}{\| \mathbf{m}_3 - \mathbf{o} \|} \tag{3.2}$$

Therefore, $\mathbf{m}_1 = \mathbf{o} + t_1\mathbf{d}$, following Equation 3.1 we obtain:

$$\mathbf{o} + t_1\mathbf{d} = (1-u_1-v_1)\mathbf{p}_1 + u_1\mathbf{p}_2 + v_1\mathbf{p}_3 \tag{3.3}$$

where u_1, v_1, t_1 are unknown. Reformulate Equation 3.3 in matrix form:

$$\begin{bmatrix} \mathbf{p}_2 - \mathbf{p}_1, & \mathbf{p}_3 - \mathbf{p}_1, & -\mathbf{d} \end{bmatrix} \begin{bmatrix} u_1 \\ v_1 \\ t_1 \end{bmatrix} = \mathbf{o} - \mathbf{p}_1 \tag{3.4}$$

Let $\mathbf{E}_{21} = \mathbf{p}_2 - \mathbf{p}_1$, $\mathbf{E}_{31} = \mathbf{p}_3 - \mathbf{p}_1$, $\mathbf{D} = -\mathbf{d}$, and $\mathbf{T}_1 = \mathbf{o} - \mathbf{p}_1$. Solving the system of equation

3.4:

$$\begin{bmatrix} u_1 \\ v_1 \\ t_1 \end{bmatrix} = \frac{1}{|\mathbf{E}_{21}, \mathbf{E}_{31}, \mathbf{D}|} \begin{bmatrix} \begin{vmatrix} \mathbf{T}_1, & \mathbf{E}_{31}, & \mathbf{D} \end{vmatrix} \\ \begin{vmatrix} \mathbf{E}_{21}, & \mathbf{T}_1, & \mathbf{D} \end{vmatrix} \\ \begin{vmatrix} \mathbf{E}_{21}, & \mathbf{E}_{31}, & \mathbf{T}_1 \end{vmatrix} \end{bmatrix} \tag{3.5}$$

Similarly, to interpolate the scalar value at $\mathbf{m}_2$ on the bottom triangle $\triangle p_4 p_5 p_6$, we obtain:

$$\begin{bmatrix} u_2 \\ v_2 \\ t_2 \end{bmatrix} = \frac{1}{|\mathbf{E}_{54}, \mathbf{E}_{64}, \mathbf{D}|} \begin{bmatrix} \begin{vmatrix} \mathbf{T}_2, & \mathbf{E}_{64}, & \mathbf{D} \end{vmatrix} \\ \begin{vmatrix} \mathbf{E}_{54}, & \mathbf{T}_2, & \mathbf{D} \end{vmatrix} \\ \begin{vmatrix} \mathbf{E}_{54}, & \mathbf{E}_{64}, & \mathbf{T}_2 \end{vmatrix} \end{bmatrix} \tag{3.6}$$

where $\mathbf{E}_{54} = \mathbf{p}_5 - \mathbf{p}_4$, $\mathbf{E}_{64} = \mathbf{p}_6 - \mathbf{p}_4$, $\mathbf{D} = -\mathbf{d}$, and $\mathbf{T}_2 = \mathbf{o} - \mathbf{p}_4$.

The radially symmetric property implies that the top triangle $\triangle p_1 p_2 p_3$ is linearly proportional to the bottom triangle $\triangle p_4 p_5 p_6$. Therefore, $u_1 = u_2$ and $v_1 = v_2$. The actual implementation only needs to evaluate Equation 3.5. In the following formulas we use u to denote u_1 and u_2, and use v to denote v_1 and v_2.

In the next step, we perform linear interpolation along the line segment $\mathbf{m}_1\mathbf{m}_2$. We first denote the scalar value of the point $\mathbf{m}_i$ by sm_i, and the scalar value of the vertex $\mathbf{p}_i$ by sp_i. Then we have:

$$\begin{aligned} m_1 &= (1-u-v)p_1 + up_2 + vp_3 \\ m_2 &= (1-u-v)p_4 + up_5 + vp_6 \\ sm_1 &= (1-u-v)sp_1 + usp_2 + vsp_3 \\ sm_2 &= (1-u-v)sp_4 + usp_5 + vsp_6 \end{aligned} \tag{3.7}$$

The t value of $\mathbf{m}_3$ on the line segment $\mathbf{m}_1\mathbf{m}_2$ can be calculated as the following:

$$t = (t_3 - t_2)/(t_1 - t_2) \tag{3.8}$$

where $t_1 = \| \mathbf{m}_1\mathbf{o} \|$, which is already computed in Equation 3.5, $t_2 = t_1 * \| \mathbf{p}_4\mathbf{o} \| / \| \mathbf{p}_1\mathbf{o} \|$, and $t_3 = \| \mathbf{m}_3\mathbf{o} \|$. Therefore, we can apply linear interpolation for the value sm_3 on the point $\mathbf{m}_3$ based on the scalar values on the points $\mathbf{m}_1$ and $\mathbf{m}_2$:

$$sm_3 = sm_2 + t(sm_1 - sm_2) \tag{3.9}$$

By substituting Equations 3.5, 3.7 and 3.8 into Equation 3.9, we can express the scalar value sm_3 in terms of (x_p, y_p, z_p) as:

$$sm_3 = \frac{a_4x + a_5y + a_6z + a_7x^2 + a_8y^2 + a_9z^2 + a_{10}xy + a_{11}xz + a_{12}yz}{a_1x + a_2y + a_3z} \tag{3.10}$$

where a_i, $i \in \{0, 1, ..., 11\}$ are the coefficients depending on the coordinates of the neighborhood vertices and their scalar values. The Mathematica software is used to help derive and simplify the Equation 3.10. See Appendix for details. In addition, let $\mathbf{e}$ denote the camera position and $\mathbf{w}$ the viewing ray direction, the point $\mathbf{m}_3$ can be expressed as:

$$\mathbf{m}_3 = \mathbf{e} + \lambda\mathbf{w} \tag{3.11}$$

We substitute (x, y, z) in Equation 3.10 using Equation 3.11, and obtain:

$$sm_3 = (b_2 + b_1\lambda + b_0\lambda^2)/(b_4 + b_3\lambda) \tag{3.12}$$

where b_i, $i \in \{0, 1, ..., 4\}$ are the coefficients depending on the coordinates of the neighborhood vertices and their scalar values. See Appendix for details. This concludes the analytic interpolation of scalar value in a frustum cell.

3.3.4 Sampling and Integration

A conventional way to compute ray integration is by sampling over a viewing ray with certain step sizes. Usually a homogeneous sampling can miss small structures within a high frequency

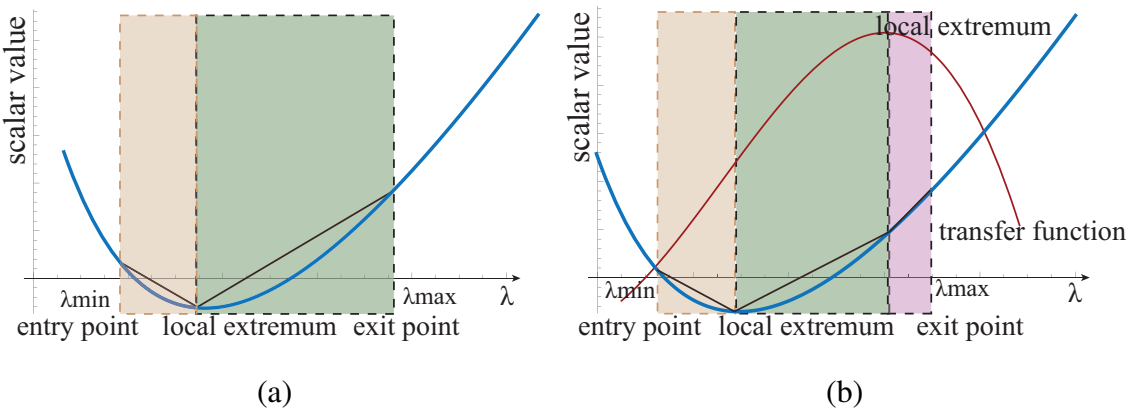


Figure 3.8. The plot of a reconstructed scalar value using our analytic function. (a) The analytic function is shown in blue, and has one local minimum. The cell along a ray is split into two intervals colored in light orange and green. The analytic function is monotonic within each interval, and its piecewise linear approximation is shown in black. (b) A transfer function is shown in red, and introduces an additional local extremum point. The cell is further split into three monotonic intervals accordingly.

scalar field. An alternative solution is to build a pre-integration table to capture possible small features. However, given the number of scalar values and the degrees of our analytic function in Equation 3.12, it is infeasible to build a pre-integration table to cover the entire interval of large data. Similar to Marchesin et al. [45], we develop a scheme for the integration of a reconstructed scalar value. For our geodesic grid rendering, the scalar function might have several local extrema depending on the specific value of b_i. Let's take one extremum for instance, as shown in Figure 3.8 (a). We can split a ray within a frustum by the ray entry point, local minimum point, and the ray exit point, to obtain a set of monotonic intervals. For each interval, we use piecewise linear approximation for integration [45]. In this way we can cover the entire interval of scalar value and capture all small structures. In addition, more extreme points may be presented over a ray if we take the transfer function into consideration, as shown in Figure 3.8 (b). In this case, we still can capture fine features by further splitting a ray at the additional extreme points, and apply the same scheme to generate high-quality volume rendering.

3.3.5 Gradient Estimation

It is possible to compute the gradient at a sampling point by taking the partial derivative of Equation 3.10 in terms of x, y and z, and obtain the gradient expression $(\frac{\partial sm_3}{\partial x}, \frac{\partial sm_3}{\partial y}, \frac{\partial sm_3}{\partial z})$. However, if the gradient estimation only relies on the six vertices of a frustum, the gradient

	Low-resolution GCRM	High-resolution GCRM
resolution	220km	28km
# cells	10242	655362
# corners	20480	1310720
# edges	30720	1966080
# layers	61	99

Table 3.1. The GCRM datasets for performance testing in the study. Both datasets contain multiple variables including vorticity, temperature, water vapor, heat flux, rain water tendency, etc.

values are not necessarily continuous at the frustum boundaries.

To smooth the gradients across frusta, a common strategy is to use the information of the neighboring frusta [14]. Because each vertex is shared by twelve frusta (except for the outer most and inner most surface points and pentagonal points), any given vertex must have twelve different gradient values with respect to the frusta incident to this vertex. First, twelve per-vertex gradients are evaluated independently using the derived analytic gradient estimation within each neighboring frustum. Second, the average of these twelve gradients is assigned to the given vertex. The averaged gradients are used to interpolate the gradient at any sampling point along the ray within the frustum. This way can significantly improve the shading effect and enhance the perception of shape and depth. Given that computing the averaged gradient is computationally intensive, an optimized approach is to pre-compute the per-vertex averaged gradients and query them during ray casting.

3.4 Results and Discussion

The approach is evaluated from several aspects, including the performance and the effectiveness of the analytic interpolation and gradient estimation. In addition, it is also compared with conventional tetrahedron-based rendering results.

variable	CPU	1 GPU	2 GPUs
heat flux	37.199s	0.494s	0.272s
temperature	45.591s	0.783s	0.468s
rain water tendency	56.247s	0.755s	0.384s
vorticity magnitude	59.397s	0.689s	0.460s

Table 3.2. Performance (in millisecond) comparison between CPU, single GPU and dual GPUs rendering of one frame. The grid has 655362 cells and 99 layers. The image resolution is 512x512. Four different variables are chosen for rendering. Note that although the grid and camera setting are the same, the variation of the amount of time across these variables in a column are due to the early ray termination.

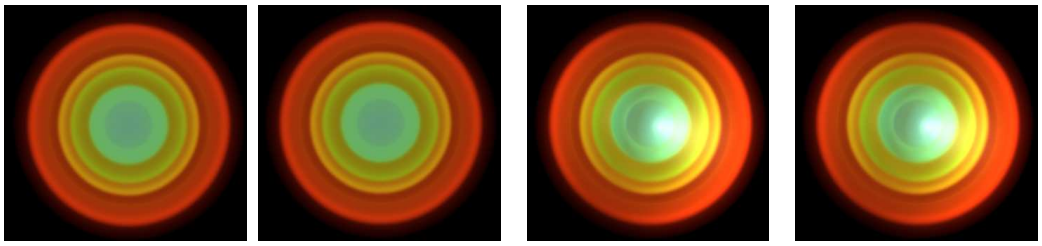
3.4.1 Rendering Performance

The experimental study has been conducted on a Intel(R) Core(TM)2 Quad CPU at 2.4 GHz with 24 GB memory and dual NVIDIA Geforce 580 GTX GPUs with 1.5 GB memory per GPU. Both CPU and GPU implementations are tested. The performance evaluation is conducted on two global cloud resolving model (GCRM) datasets with low and high resolutions. The datasets were produced by the Pacific Northwest National Laboratory. The detailed information of the two GCRM datasets is listed in Table 3.1.

Table 3.2 shows the performance difference between CPU, single GPU and dual GPUs rendering on the high-resolution geodesic grid. The CPU rendering uses only one CPU core and performs rendering in a sequential fashion. The dual GPUs test duplicates the grid data on each GPU and partitions workload in image space. It is shown that the single GPU accelerated volume rendering achieves approximately a 100-fold speedup vs. CPU rendering, and the dual GPUs rendering achieves roughly a two-fold speedup vs. the single GPU version.

3.4.2 Rendering Quality

To verify the effectiveness of the analytic scheme for ray integration and gradient estimation, we first use a synthetic spherical scalar field generated on a real geodesic grid. The ground truth images are also generated in a way that at a sampling point along a ray the scalar value is determined as the Euclidean distance between the sphere center and the point, and the gradient is determined by the vector from the sphere center to the point, as shown in Figure 3.9 (d).



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

Figure 3.9. Ray casting of a synthetic spherical scalar field defined on a real geodesic grid with 10242 cells and 61 layers. (a) and (c) show the results using the analytic approach with and without lighting respectively. (b) and (d) show the ground truth images with and without lighting respectively.

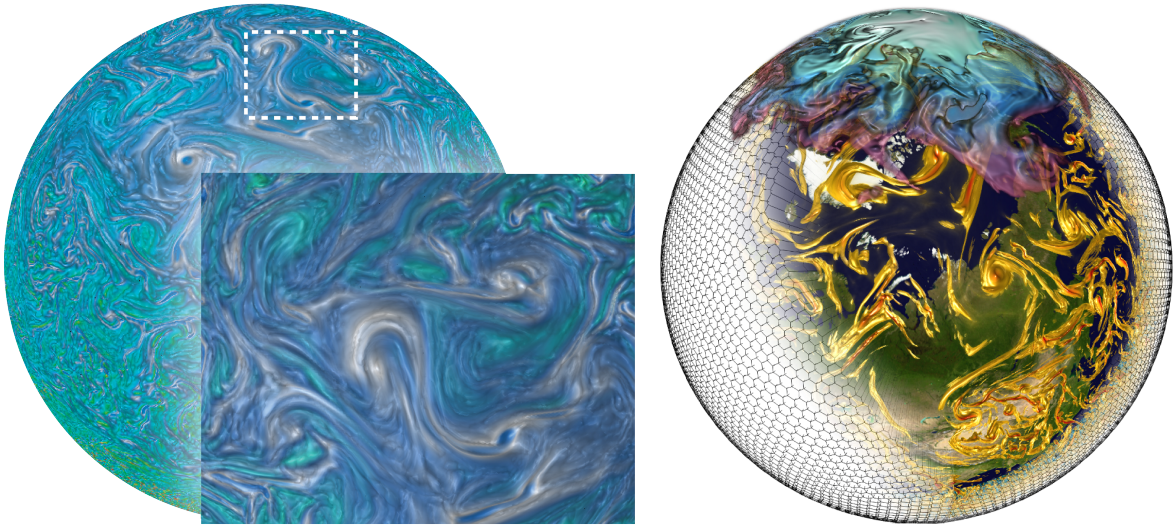


Figure 3.10. Ray casting with lighting using the analytic ray integration and gradient estimation on the high resolution dataset. The left images shows the vorticity variable and the close-up view of a selected region. The right image shows the combination of grid illustration, volume rendering of vorticity (yellow tubes) and temperature (north cap). Note that for the purpose of clear illustration in the right image, the grid is coarser than the actual one used to generate the vorticity and temperature. The introduced approach can reveal important details from large data with enhanced depth and shape cues.

It is noted that the analytic scheme can generate high-quality rendering and provide nearly indistinguishable results from the ground truth. The difference only becomes perceivable when the grid resolution is extremely low, which is normally impractical for simulations.

Figure 3.10 demonstrates the quality of ray integration and lighting effects using the analytic scheme on the real climate dataset. It is shown that our approach can achieve high-quality

rendering with smooth shading. In particular, the fine turbulent features embedded in the high-resolution data are succinctly revealed with the enhanced depth and shape cues provided by our gradient estimation method.

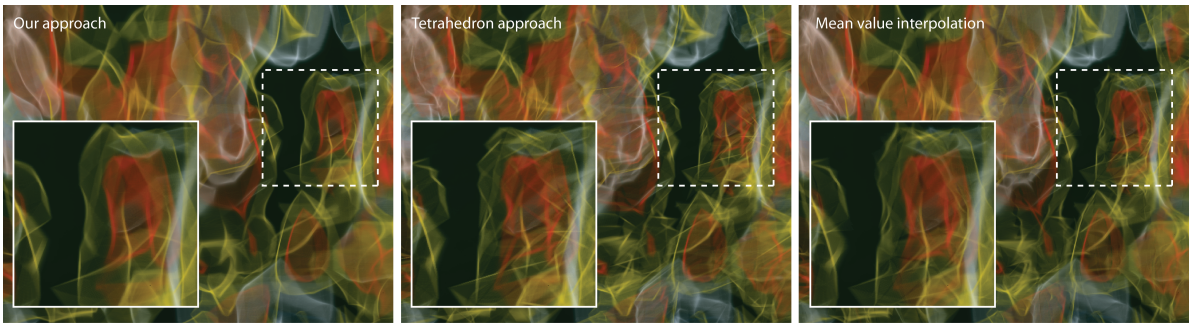


Figure 3.11. Rendering quality comparison of our approach (left) with conventional tetrahedron based method (center) and mean value interpolation (MVI) (right) in a close-up view. We render the vorticity variable in the high-resolution grid. The ratio of sample step size to the globe radius is set as 10^{-4} to make sure there are enough sampling points within each cell. The boundary artifacts at the high frequency regions are quite salient in the rendering of tetrahedral mesh due to less vertex information for interpolation. Meanwhile, even if the rendering quality of mean value interpolation method is slightly better than the tetrahedron method, certain jaggy effect can still be perceived. This is due to the fact that mean value interpolation has proved to be smooth in the interior of a cell except at the vertices [26]. Therefore, in terms of image quality, our approach outperforms both methods.

We also provide both qualitative and quantitative comparisons of our approach with two state-of-the-art approaches. To have a fair comparison, all three approaches in this experiment sample the ray at the same uniform interval. The first approach implicitly tetrahedralizes each frustum on the fly and uses the ray-casting method on tetrahedral mesh [85] to render the transformed data. The second approach employs mean value interpolation [54] [55] to compute scalar values at sampling points within a frustum. Figure 3.11 demonstrates the rendering quality comparison of our ray-casting approach with these two methods in a close-up view. We observe that our approach can generate high-quality rendering superior to both methods, especially at the regions with high frequency. Our ray-casting also features a lower complexity in terms of overall cell count, and can achieve a higher frame rate than the conventional methods, as denoted by the performance comparison in Figure 3.12. Our approach achieves an average of 98.6% performance gain over the tetrahedron method. This is mainly because the tetrahedron-based method requires visibility sorting of three tetrahedrons within a frustum. Our approach

also achieves an average of 172% improvement over the mean value interpolation method. This is maily because it requires an inverse trigonometric function and several vector operations to compute the angles over eight triangular faces of a frustum for mean value coordinates [55].

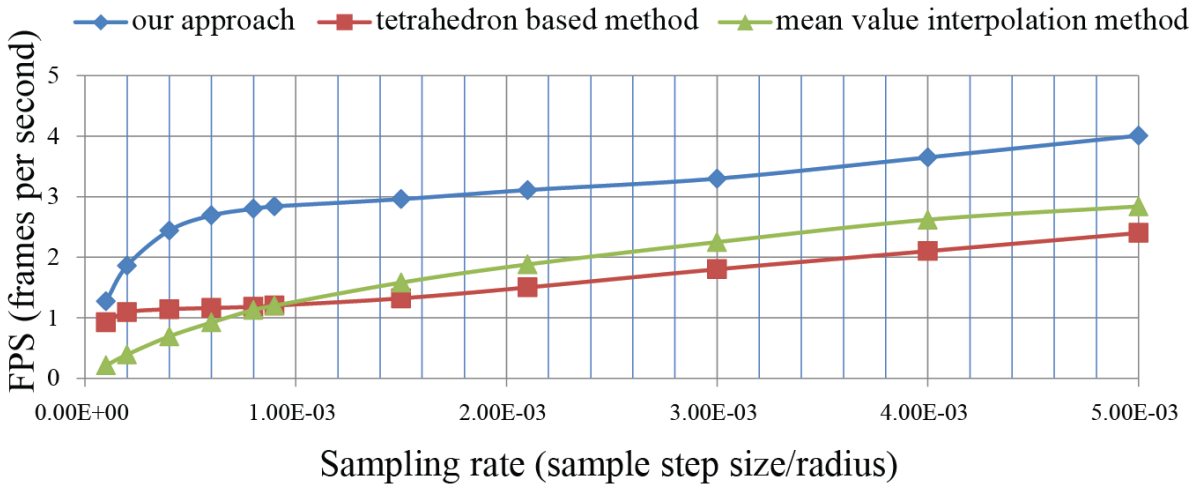


Figure 3.12. Performance measures and comparison with conventional tetrahedron based method and mean value interpolation method. The performance data is obtained by rendering images of Figure 3.11 with different sampling rate defined as the ratio of sample step size to the globe radius. Our approach outperforms both methods.

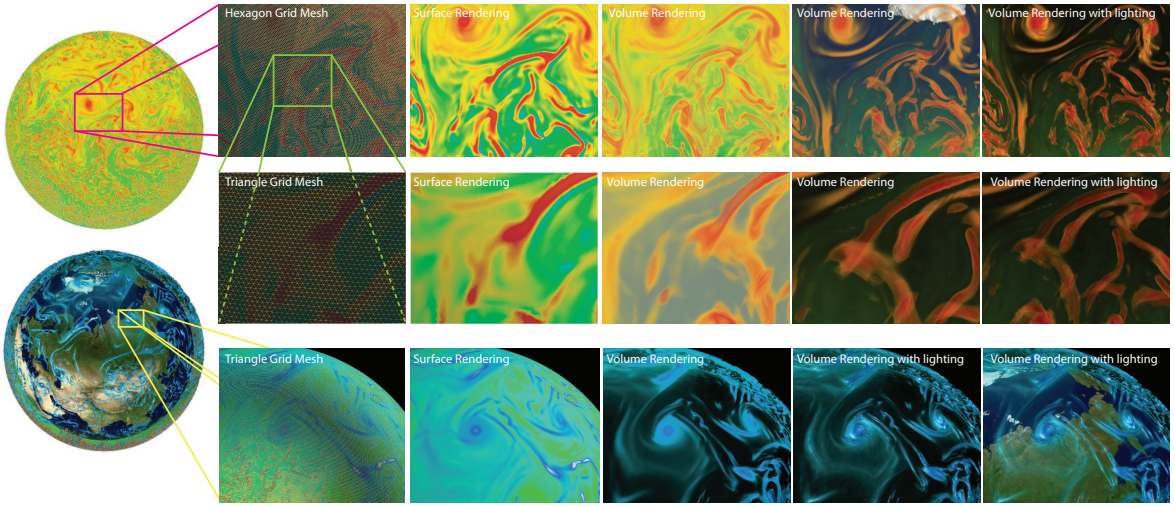


Figure 3.13. The left most two images show the volume rendering of the whole global atmospheric vorticity variable using different visualization parameters. The top two rows of images incrementally show the close-up views of a particular region of interest. From left to right, the mesh grid, the surface rendering, the volume rendering with different transfer functions, and the images with lighting disabled and enabled. The bottom row shows a close-up view from a different geolocation, superimposed with geophysical information.

Figure 3.13 shows a set of high-quality and high-resolution rendering of full-resolution data by our approach, which allows scientists to capture the fine details from their large climate data on geodesic grids.

3.5 Summary

This chapter presents a direct volume rendering approach to geodesic grid volume data visualization. Without lose of generality, the approach is applied to hexagonal grids, which can be directly fetched and rendered without any intermediate grid transformation. As a result, it can significantly reduce computing and memory cost. For the higher-resolution grid in the current study, the approach directly takes the grid of outer most surface with 1.3 million grid points as input. However, explicitly converting the geodesic grid to the tetrahedral or polygonal grids over 99 layers would result in an input of 128.7 million grid corners and thus makes the memory and preprocessing cost unaffordable for large datasets. The data structure in the approach has the same complexity as the original simulation grids, and is potentially scalable with the increasing scale of simulations. Furthermore, the current design matches the simulation data's storage patterns, which facilitates direct integration with simulation codes and offers possibly viable in-situ visualization solutions for large climate simulations.

A new analytic scheme for interpolation, gradient calculation, and ray integration has also been derived. The accuracy of the analytic scalar and gradient interpolation is comparable to those using central difference numerical computations in simulations. The rendered results feature better image quality, less memory overhead, and high performance compared with conventional methods.

There are several promising directions for future work. First, there is room to improve the gradient calculation using higher order analytic functions. Care must be taken as such improvements may incur more memory access and computing cycles. It is an interesting topic on how to make a trade off between image quality and computation complexity for large data.

Secondly, there are still opportunities to optimize the performance of multiple GPUs with the ray-casting scheme, and address the bottleneck of data transfer between GPU memory and

main memory. The current grid is generated after seven recursive subdivisions of an icosahedron, such that each mesh cell covers an Earth surface area of 27 km in diameter. Currently, climate scientists are generating higher-resolution grids using finer subdivision. For example, in the near future, a grid will have a resolution of 4 km in diameter and 256 layers. Given this scale, the size of each variable will reach to 4.5 terabytes. No single machine can presently handle such a large scale dataset with ease. The next chapter will seek a multi-resolution solution on GPU clusters and in-situ solution on HPC supercomputers to effectively process the large datasets. This enables scientists to study climate data at full scale and deliver more accurate global weather or climate forecasts. This chapter presents an important foundation work for carrying out these further research tasks.

Chapter 4

Visualizing Large 3D Geodesic Grid Data with Massively Distributed GPUs

Advanced supercomputing techniques and systems allow scientists to conduct simulations with detailed numerical weather and climate models. Geodesic grids have become increasingly prevalent in the development of models [11, 63]. This type of grid can facilitate the modeling of the Earth's surface with higher resolutions and higher numerical stability, leading to a simulation of an unprecedented scale. Such a simulation can possibly generate tera- or petabytes of data that are typically time-varying and multivariate. The size of data requires scalable and interactive visualization techniques for agile exploration and timely weather and climate prediction. Parallel visualization provides a solution to address the vast amount of simulation data, in that data are partitioned and distributed among multiple processing units and visualization calculations are conducted in a divide-and-conquer manner. A parallel visualization algorithm with a balanced workload assignment can achieve scalable performance over a large number of processors, and make it possible to interactively explore massive data.

Although extensive research has been carried out on parallel visualization, several fundamental challenges have prevented a direct application of current solutions on geodesic grids. First, most parallel visualization solutions rely on data partitioning and distribution schemes that are designed in Cartesian coordinates, while geodesic grids are constructed in the spherical coordinate system. The data representations used in geodesic grids are fundamentally different from grids in Cartesian coordinates. Existing data partitioning and distribution schemes cannot be directly applied to handle geodesic grids. Second, the spherical structure introduces unique

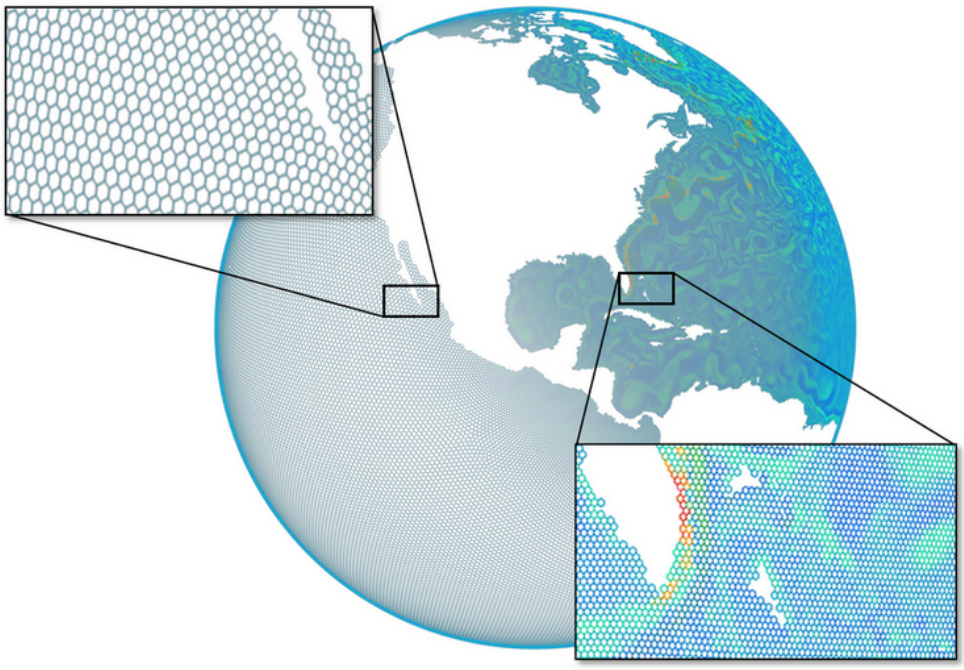


Figure 4.1. An example of spherical geodesic grids covering the ocean of the Earth surface. The resolutions of Voronoi polygons vary across different regions: higher-resolution grids are used in the regional areas of interest, while lower-resolution grids are placed for the remainder of the surface. Note that this example only displays the mesh of the top surface. For a real simulation, this spherical mesh is scaled and duplicated to construct a set of layers along the direction perpendicular to the Earth surface, resulting in a 3D mesh.

visualization requirements in that only the scalar fields in front are desired to be visualized. Given the multiresolution nature of geodesic grids, the data density of visible regions can vary significantly across different view angles. Thus it is difficult to estimate the rendering cost using conventional methods for scientific volume data.

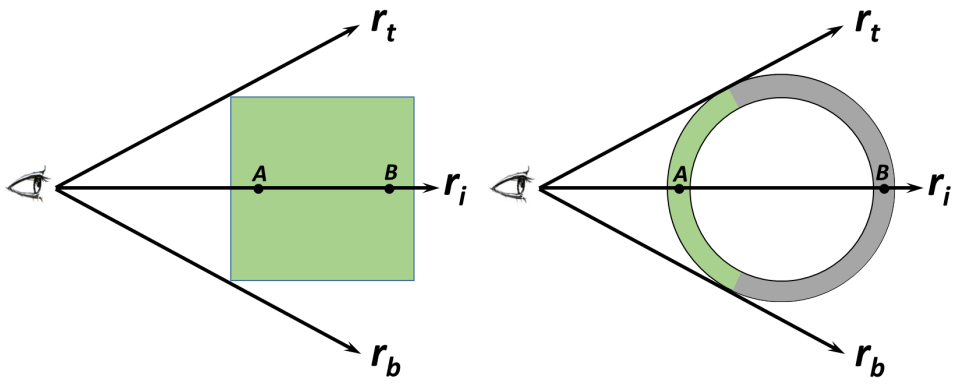
In this chapter, we introduce a scalable parallel solution to interactively visualize large-scale geodesic grid data using multiple GPUs. Unlike the grid we dealt with in the last chapter, the one we use in this chapter features various resolutions at different regions of interest. This grid has been widely adopted by the Model for Prediction Across Scales (MPAS) project for developing atmosphere, ocean and other earth-system simulation components [64]. In particular, the grid for the simulation of the ocean system only has the ocean water be gridded and no grid is designated in the continental area, as shown in Figure 4.1. The finer grids have only finer Voronoi grid cells and do not have more corresponding depth layers. The varying depth

of each ocean layer can be accessed through a separate data table. For the details on the multi-resolution tesselation of the grid, we refer readers to the work by Ringler et al. [64]. Based on a careful characterization of geodesic grid visualization, we design a new data partitioning and distribution scheme that employs a spherical quadtree to decompose and index the multiresolution spherical grids. This spatial data structure enables us to accurately estimate the rendering load for regions with different resolutions. Moreover, we use the quadtree structure to scatter the regions among the processors and ensure that each processor can be assigned an approximately equal amount of workloads from any viewing direction. Therefore, no processors are idle during the rendering, and the maximum parallelism can be achieved. Our implementation of the entire rendering pipeline is based on the MPI and CUDA architecture and can be directly executed on state-of-the-art supercomputers. The scalability and effectiveness of our solution have been demonstrated using the full extent of geodesic grid data from real-world simulations. Great 3D details can be delivered at an interactive rate to scientists for exploration. In addition, our approach directly takes the original simulation data as input, and it can be readily extended to support in-situ visualization.

4.1 Characterizing Distributed Geodesic Grid Visualization

A balanced workload assignment is the key to the scalability of parallel visualization algorithms. Given the unique visualization requirements from geoscience applications, current methods to estimate the workload associated with conventional mesh structures cannot be applied to geodesic grids.

For conventional mesh structures, the entire domain of a scalar field is often counted in workload estimation regardless of view angles. This is because during volume rendering of the field users can adjust transfer functions and assign different opacities to different regions, which implies that the entire domain can be involved in the rendering calculation along an arbitrary ray. As shown in Figure 4.2(a), along a ray r_i for rendering a general scalar field, both the sampling points A and B can be perceived with a certain configuration of transfer function. Therefore, when designing a parallel visualization algorithm, we may assign the regions containing A and



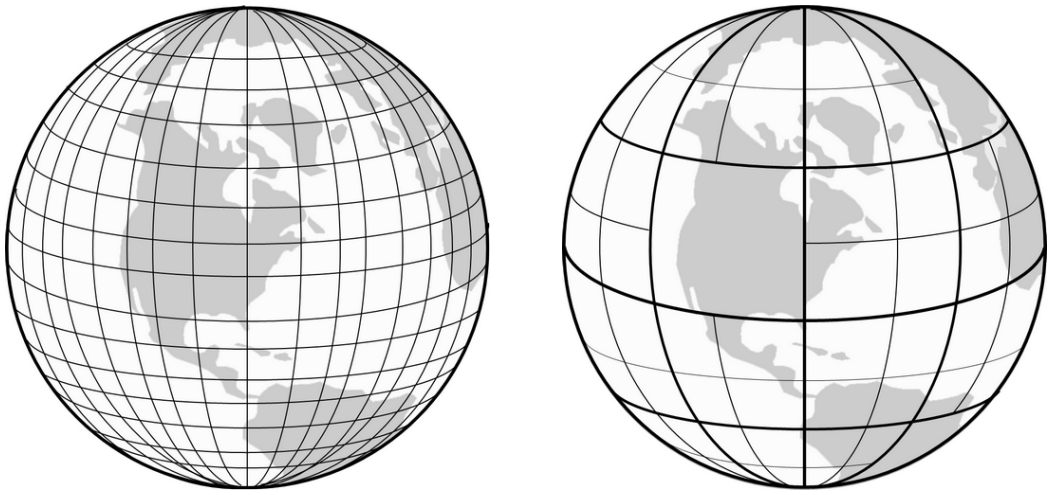
(a) (b)

Figure 4.2. Ray-casting of scalar fields. r_t and r_b correspond to the boundary of the view frustum. A and B are the sampling points along a ray r_i. (a): For a general scalar field, both A and B can be visible, and the entire domain (in green) is involved in the rendering calculation. (b): For a scalar field defined on a spherical mesh covering the Earth, we only render A in the front region (in green), while excludes B in the back region (in gray) that is invisible.

B to different processors, and all processors can be involved in rendering from any view angle.

However, in a geoscience application, scientists typically focus merely on the scalar field of the front Earth surface towards the viewers. For the phenomena over the other region, they can rotate the sphere and bring that region to the front for observation. As shown in Figure 4.2(b), for spherical geodesic grids covering the Earth surface, we just render the sample points in the front visible region along a ray r_i. Although it is technically feasible to make A and B visible, a display of both points can preclude a clear observation and thus is rarely applied in practice. In this case, if we still simply assign the regions of A and B to different processors, some processors can be idle during the rendering if the corresponding regions have been rotated to the back, which may lead to severely unbalanced workloads among the processors.

In addition, as shown in Figure 4.1, the mesh existence and density can vary greatly in a model of the Earth surface. Thus, rendering workloads can be dramatically different with respect to different view angles. For example, showing the Pacific Ocean would incur a larger amount of rendering calculations compared to a display of the Americas due to a lack of grids in the continental regions. Such use of unstructured and variable-density grids exacerbates the issue of workload assignment, in that the amount of grids assigned to different processors



(a) (b)

Figure 4.3. (a) shows a regular partitioning scheme along the latitude and longitude. (b) shows our spherical quadtree based partitioning scheme, which can achieve load balancing with a smaller number of regions and a lower cost of parallel image compositing.

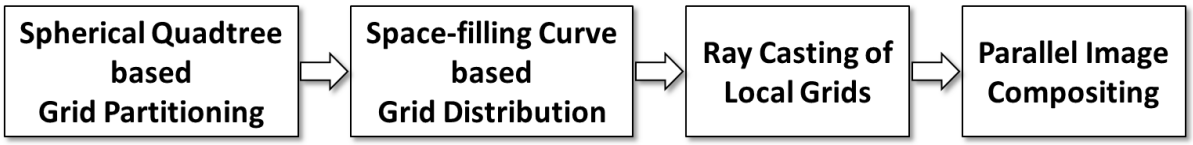


Figure 4.4. The major steps of our parallel volume rendering framework.

must be carefully balanced with a holistic consideration of view angles and grid resolution distributions.

4.2 Parallel Volume Rendering Framework

A parallel rendering framework is typically comprised of the stages of data partition, distribution, rendering, and image compositing. Given the characterization of geodesic grid visualization, we have taken several design considerations into account for the development of a framework.

First, there are three basic parallel rendering approaches, namely, sort-first, sort-middle, and sort-last [51]. Although the rendering cost of geodesic grids is largely view-dependent, an image-based workload partition scheme may constantly require communication between processors to exchange simulation data when changing views. This communication cost can be

prohibitively high for processing large climate simulation data. To this end, we choose sort-last parallel rendering because of its simple workload decomposition for achieving load balancing and no communication overhead involved in the rendering stage.

Second, data partition and distribution are conducted before the rendering stage in sort-last parallel rendering. In order to maximize the parallelism, it is imperative to keep all the processors busy in rendering visible regions. However, if a processor is only assigned one or a few partitions, it will be idle when its regions are facing away from the viewer. As illustrated in Figure 4.2(b), one intuitive idea is to assign the opposite regions along a ray, such as the regions containing both A and B, to one processor. That is, the processor is always busy whenever a user is facing towards either A or B. It is possible for the processor to become idle again when the viewing direction is perpendicular to AB. Thus, we need to design a more sophisticated partition and distribution strategy to allow each processor to have visible regions to render from any viewing direction.

Third, besides visible regions, each processor would also be ideally assigned a roughly equal amount of rendering workload from any viewing direction. The data size of each region depends on its local mesh resolution. However, the raw mesh of a simulation data mainly contains the connectivity information rather than the mesh densities. It requires us to find a way to index the data according to the variation of grid resolutions, such that we can quickly quantify the mesh density and then compute the data size for any given region. This functionality is vital for us to accurately estimate the regional rendering cost for load balancing.

These considerations can lead to a design in which we can regularly decompose the spherical surface into a set of patches along the latitude and longitude lines, as shown in Figure 4.3(a). The number of the patches is sufficiently larger than the number of processors, and then we randomly distribute the patches among the processors. Hence, each processor can be assigned the regions scattered over the surface, and a portion of regions are visible from a viewing direction. In addition, by randomization, the amount of data assigned to each processor can be roughly equal. This design is based on fine-grained partitioning and randomization, which is commonly used in distributed computing for load balancing. However, a large number of decompositions can also increase the number of the partial images generated by each processor. These partial images

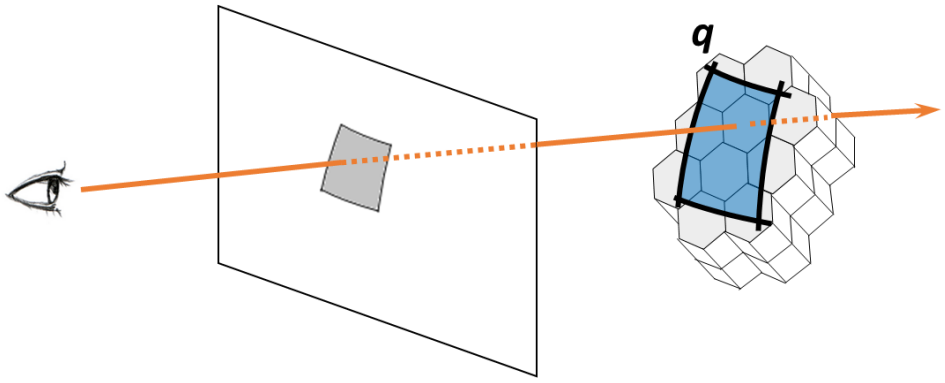


Figure 4.5. Ray-casting of a quadrant. Given a quadrant q (in blue), we can project it onto a 2D screen space. The gray area corresponds to the pixels of projection. A ray is casted from each pixel to penetrate the grid cells overlapped by q, and the color and opacity values of sample points are accumulated along the ray to generate the final color of the pixel.

of each processor typically cannot be composited locally, because they correspond to a set of scattered regions whose projections may not be continuous in depth. A significant overhead will be introduced in the parallel image compositing stage for merging a large number of partial images into a final image, and becomes the main performance bottleneck of the entire pipeline.

We design our sort-last parallel volume rendering framework for geodesic grids to address these issues. We first construct a spherical quadtree to cover the surface of a geodesic mesh, as shown in Figure 4.3(b). The quadtree is refined adaptively according to the count of regional grid cells. In this way, we can quantify the mesh density within the region of any quadtree node. We then partition the grids into the regions corresponding to the leaf nodes of the quadtree, and distribute them among the processors according to the traversal order of the leaf nodes. Hence we can not only control the total number of regions, but also ensure that each processor is assigned a number of regions scattered across the spherical surface. No processors are idle for any viewing direction, and the rendering load is balanced among the processors. After each processor renders its assigned regions, we use parallel image compositing to generate the final image. Our approach can lower the number of partitions and significantly reduce the overhead of parallel image compositing compared to the conventional fine-grained partitioning scheme. Figure 4.4 shows the major steps of our approach.

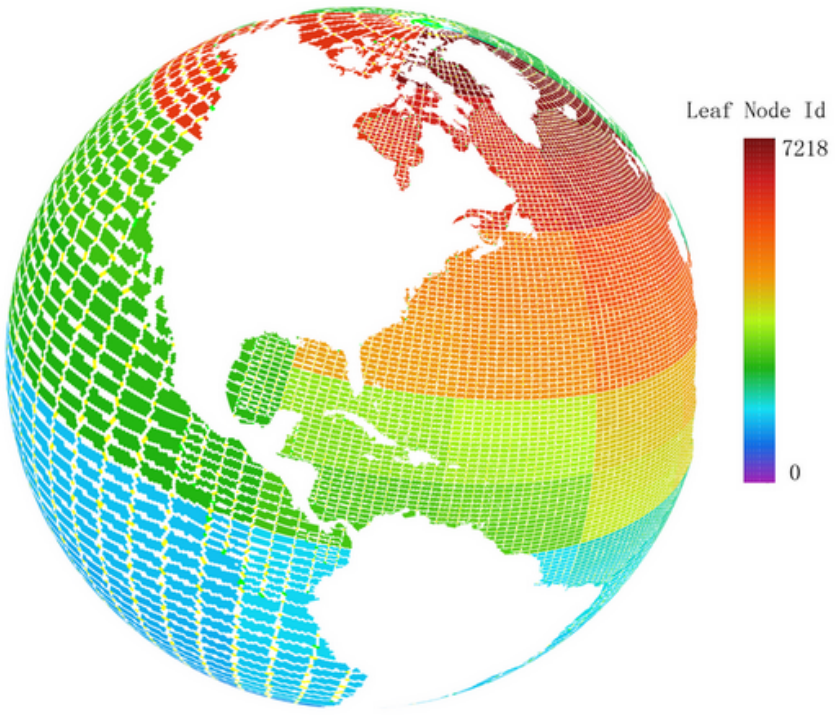


Figure 4.6. Rendering of a quadtree constructed from a simulation dataset.

4.2.1 Spherical Quadtree-based Grid Partitioning

A quadtree is a commonly used data structure to partition a 2D space. A typical quadtree is constructed by recursively subdividing the space into four quadrants or regions until certain criteria are reached. The concept of quadtrees can be naturally extended to decompose spherical surfaces [37, 83, 77] in that the decomposition can be conducted along the latitude and longitude lines instead of X- and Y-axes in a 2D square space.

We use the quadtree to partition spherical geodesic grids such that each quadrant or region will be associated with an approximately equal amount of rendering load. To achieve this goal, we first need to estimate the rendering cost of a quadrant q. We use ray-casting to volume render geodesic grids. As shown in Figure 4.5, we first project q onto a 2D screen space, and then cast a ray from each pixel to penetrate the grid cells overlapped by q. For each ray, we sample the scalar field and accumulate the color and opacity values of sampling points to compute the final color of the pixel. Thus, the rendering cost is proportional to the number of pixels and the number of cells. We note that the number of pixels depends on the projection and the area

of the region. For a static view, the number of pixels projected from different regions can be different. But if we allow users to interactively view the spherical surface from any direction, the amortized number of pixels projected from a region is proportional to the area of the region, because each region on a sphere has an equal probability to be viewed. Thus, for a quadrant q, the rendering cost C_q, is estimated as a linear function of its area S_q and its number of cells G_q:

$$C_q = kS_qG_q, \tag{4.1}$$

where k is a constant, and S_q is computed according to the latitudes lat_1 and lat_2 and the longitudes lon_1 and lon_2 that bound q[1].

This cost model guides us in constructing a quadtree. Given a geodesic mesh, we first use Equation 4.1 to estimate the total rendering cost C_t using the total spherical surface area and the total cell number. Assume that the number of processors is N and each processor will be assigned m quadrants. The average rendering cost C_{avg} of each quadrant is computed as

$$C_{avg} = \frac{C_t}{mN}. \tag{4.2}$$

We start to recursively subdivide the spherical surface to construct the quadtree. We stop subdividing a quadrant if its estimated rendering cost is smaller than C_{avg} or the total quadrant number is larger than mN. In this way, we can construct a quadtree where each quadrant is associated with a similar rendering cost.

According to the Earth satellite constellation design, three satellites spaced equally around the equator can cover most of the Earth [13]. Imagine that a user's view point is a satellite around the spherical mesh; this implies that each processor needs to be assigned at least three quadrants scattered on the surface. Thus from any viewing direction, a processor can have at least one quadrant that is visible, which can prevent the processor from becoming idle. In practice, we find that $m = 5$ leads to a good performance. Figure 4.6 shows a quadtree constructed from a real simulation data. We can clearly see that the finer-grained quadrants are generated to cover the higher-resolution regional areas of interest, while the coarser-grained quadrants cover the remains of ocean area, which matches the distribution of grid resolutions in Figure 4.1.

[1]Our implement is similar to the `areaquad` function of MATLAB to compute surface area of latitude-longitude quadrangle [46].

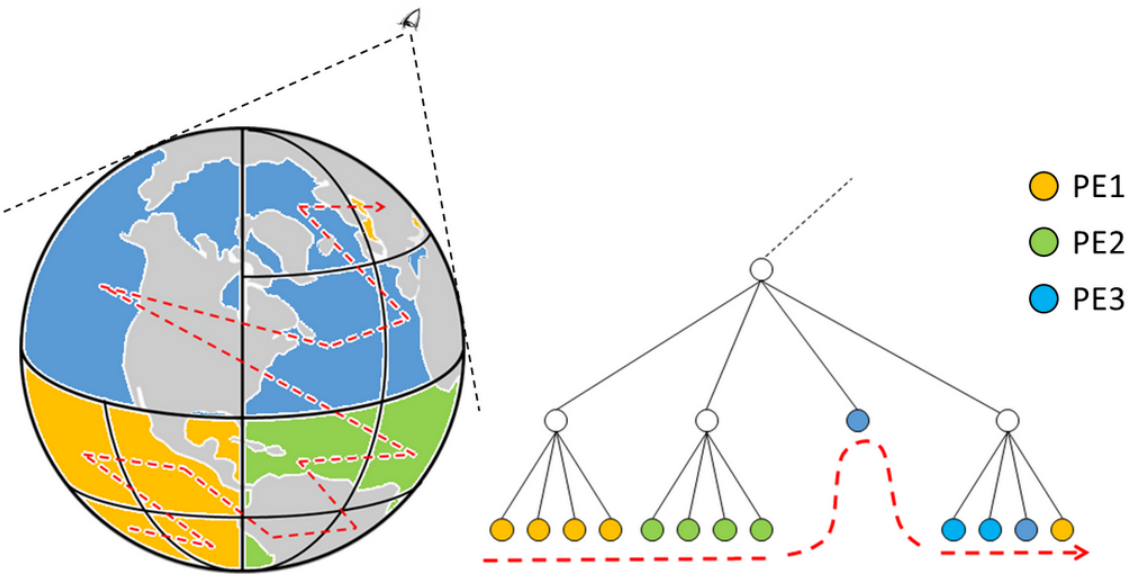
With our spherical quadtree based partitioning, the shape of the quadrants can be different across the sphere: they are close to be rectilinear for regions around the equator, but are triangular for regions around the poles, as shown in Figure 4.3(b). Our cost model considers the surface area of quadrants, and the workload estimation is independent of the location and the shape of a quadrant.

4.2.2 Space-filling Curve-based Grid Distribution

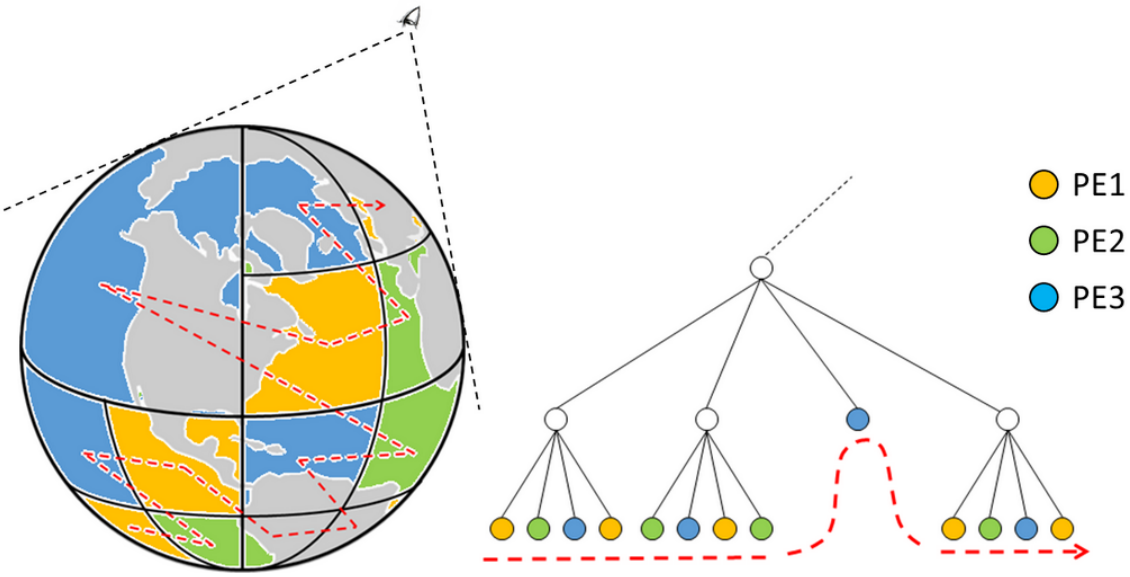
It is desirable to assign each processor with a set of quadrants that are as scattered as possible. A straight-forward approach is to randomly assign the quadrants among the processors. However, we can achieve a more appropriate assignment by leveraging the spatial locality encoded in a quadtree. If we use the linear quadtree technique [21] to encode and distinguish quadrants, a pre-order traversal of quadrants will lead to the well-known space-filling curve which groups the spatial nearby quadrants together on the spherical surface.

Figure 4.7 shows an example of spherical surface decomposition and the corresponding quadtree. The traversal of leaf quadrants from left to right is equivalent to the zigzag on the spherical surface. Therefore, a strong spatial locality can be clearly identified for these quadrants. This property of quadtree has been widely used in the optimization of data layout. For example, the grid cells can be linearized and saved in persistent storage according to the space-filling curve obtained by quadtree. This storage pattern can guarantee contiguous reads/writes, and improve I/O performance [15].

If we assign the processors along the space-filling curve for parallel visualization, each processor will be responsible for contiguous regions on the surface. Figure 4.7(a) illustrates such an assignment for three processors distinguished in different colors. However, the regions assigned to a processor can be occluded from certain view points. For example, as shown in Figure 4.7(a), the green regions cannot be seen if the viewer is around the north pole, and the processor PE2 becomes idle.



(a)



(b)

Figure 4.7. The spatial decomposition of a spherical surface and its corresponding quadtree. A pre-order traversal of quadrants is equivalent to the space-filling curve on the spherical surface. (a) shows that we evenly assign the quadrants among three processors from left to right in the quadtree, and the distribution of their regions is contiguous along the space-filling curve. In this case, each processor's regions may not be always visible from different viewing directions. (b) shows that we assign the quadrants among three processors in round robin, and the neighboring regions are largely assigned to different processors. In this case, a portion of a processor's regions can be visible from any viewing direction.

We note that the spatial locality becomes the best along the space-filling curve of the decomposition. On the other hand, we may achieve the *worst* locality using the space-filling curve in the opposite way to favor visualization workload assignment. Instead of a contiguous assignment, we distribute the quadrants among the processors in round robin along the space-filling curve. Given a sufficient number of processors, this distribution can guarantee that the neighboring regions are assigned to different processors. As shown in Figure 4.7(b), the regions of each processor scatter across the spherical surface, and a processor always has a portion of regions that are visible from any viewing direction. Thus, we can keep all processors busy in rendering. In addition, each processor is still assigned an equal number of quadrants, and each quadrant corresponds to a nearly equal amount of rendering cost. This assignment enables our solution to achieve load balancing during interactive exploration.

4.2.3 Ray Casting of Local Geodesic Grids

After we partition and distribute geodesic grids among the processors, each processor starts to render its local regions, where the GPU-based ray-casting algorithm developed in the last chapter is employed in this stage.

The algorithm is tailored to process geodesic grids with minimal overhead. First, we directly use the original Voronoi polygonal mesh of simulations in rendering without any intermediate grid transformation. In particular, a set of table-based representations are deployed to manage the mesh in GPU memory for efficient data access. Then, during the process of ray-casting, we leverage the properties of geodesic grids and march rays using the 2D connectivity information of the outer layer. Hence, we do not need to reconstruct full 3D connectivity information, and can further reduce memory and computing overhead. To achieve high-quality rendering, an analytic solution has been designed to reconstruct the signal within a geodesic grid cell for scalar value interpolation, gradient estimation, and ray integration. The accuracy of the analytic scalar and gradient interpolation is comparable to the results achieved by central difference numerical computations in simulations.

Each processor iterates through its assigned quadrants and renders the grid cells of each quadrant into a partial image using the algorithm mentioned in the last chapter. The rendered results feature high image quality, less memory overhead, and higher computing performance.

In addition, no communication is needed in this stage, and thus local grid rendering can scale well given our partitioning and distribution scheme for load balancing.

4.2.4 Parallel Image Compositing

The partial images rendered by each processor need to be composited (i.e., back-to-front alpha blending) to generate the final image. The parallel image compositing stage requires inter-processor communication, and can become expensive when the number of partial images increases. The most representative parallel image compositing algorithms include direct send [56] and binary swap [43]. Direct send is simple and easy to implement; however in the worst case it needs to exchange $N(N-1)$ messages among N compositing processors, introducing link contention due to its nature of all-to-all communication pattern. Binary swap uses a binary tree style compositing process and reduces the number of messages from $N(N-1)$ to $Nlog(N)$. However, to achieve the best performance, binary swap needs the number of processors to be an exact power-of-two.

Yu et al. [102] presented the 2-3 swap image compositing algorithm that combines the advantages of direct send and binary swap. On one hand, 2-3 swap can be as flexible as direct send in that it can use any number of processors. On the other hand, 2-3 swap involves the number of messages bounded by $O(Nlog(N))$, which is as efficient as binary swap. Peterka [61] presented the Radix-k algorithm that unifies direct-send and binary swap and has the similar complexity as 2-3 swap. However, it is non-trivial to configure Radix-k for achieving optimal performance [82].

We use 2-3 swap in this work for parallel image compositing. As discussed in Section 4.2.1, each processor can be assigned m quadrants, and the total number of partial images is mN for N processors. Because of the spatial discontinuity of the quadrants assigned to each processor, these partial images are not necessarily contiguous in depth on a processor, and cannot be blended locally before being sent to the other processors. Therefore, one message is required for one partial image in the worst case, and the total number of messages is bounded by $O(mNlog(mN))$. Compared to the fine-grained partitioning and distribution scheme discussed in Section 4.2, our scheme can minimize the value of m by leveraging spherical quadtrees, and significantly reduce the image compositing cost.

4.3 Results and Discussion

	GCRM	MPAS
resolution	28km	15∼75km
# cells	655362	253746
# vertices	1310720	517338
# edges	1966080	771377
# layers	60	40
# time steps	40	12

Table 4.1. The GCRM and MPAS datasets used in our evaluation. Both datasets contain multiple variables.

We have evaluated our framework using two datasets, where one is generated from the Global Cloud Resolving Model (GCRM) and the other is from the Model for Prediction Across Scales (MPAS). Both GCRM and MPAS are developed based on geodesic grids where mesh density varies over the Earth according to the distribution of regions of interest. However, GCRM is mainly used to model cloud processes in atmosphere, and a GCRM mesh covers the entire sphere. MPAS is mainly used to model ocean processes such that the continental areas are not covered in a MPAS mesh. In general, compared to a GCRM mesh, a MPAS mesh exhibits more variation in grid resolutions over the sphere. Table 4.1 lists the detailed information of the GCRM and MPAS datasets.

We performed our experimental study on Titan, a Cray XK7 system at Oak Ridge National Laboratory. The system contains 18,688 compute nodes, and each node has a conventional 16-core AMD Opteron CPU and an NVIDIA Tesla K20 GPU accelerator with a total 38GB of memory. The compute nodes are connected through a Cray Gemini interconnect.

We conducted a strong scaling test on our approach with an increasing number of GPUs from 4 to 1024. We rendered the datasets using three different image sizes, including 1024^2, 2048^2, and 4096^2, and from 10 different viewing directions that are evenly distributed across

the sphere. For each viewing direction, we measured the timing results of the major operations, including rasterization, ray-casting, and parallel image composting. The timing results are averaged over the viewing directions.

Figure 4.8 shows the timing results for rendering the GCRM dataset. The rasteraization time is nearly negligible for all three image resolutions. Our CUDA-based implementation of rasterization can render millions of polygons interactively. For the GCRM data, our rasterization approach achieves a rate of 30 frames per second for 2048^2 images and a rate of 10 frames per second for 4096^2 images using 4 GPUs. The performance is comparable to an implementation using the native OpenGL on similar GPUs.

We can see that the ray-casting time dominates the overall time for a smaller number of GPUs, which constantly decreases with the increasing number of GPUs. The measured timing results are close to the ideal speedup time. For a high-resolution image of 4096^2, our approach achieves a parallel efficiency of 85% from 4 GPUs to 64 GPUs, and a parallel efficiency of 50% from 4 GPUs to 1024 GPUs.

The compositing time is nearly constant with the increasing number of processors because of its logarithmic complexity. As shown in Figure 4.8, when the processor number is less than 64, as expected, the compositing time can be hidden by overlapping ray-casting and image compositing. However, when the processor number is larger than 64, the compositing time starts to dominate the overall time. Most existing parallel visualization solutions for supercomputers use CPU-based rendering algorithms, and thus the rendering time is longer than the image compositing time until a large number of processors are used. In our new GPU-based parallel visualization framework, the rendering time can be significantly reduced by leveraging GPUs available on supercomputers, while compositing is still conducted using CPUs and MPI. Thus, the curve of composting time can quickly intersect with the curve of rendering time even for a smaller number of processors.

Figure 4.9 shows the timing results for rendering the MPAS dataset that features a high variation of mesh density. Even for this dataset, our approach demonstrates a similar performance trend as for the GCRM data. The ray-casting time is still close to the ideal speedup time.

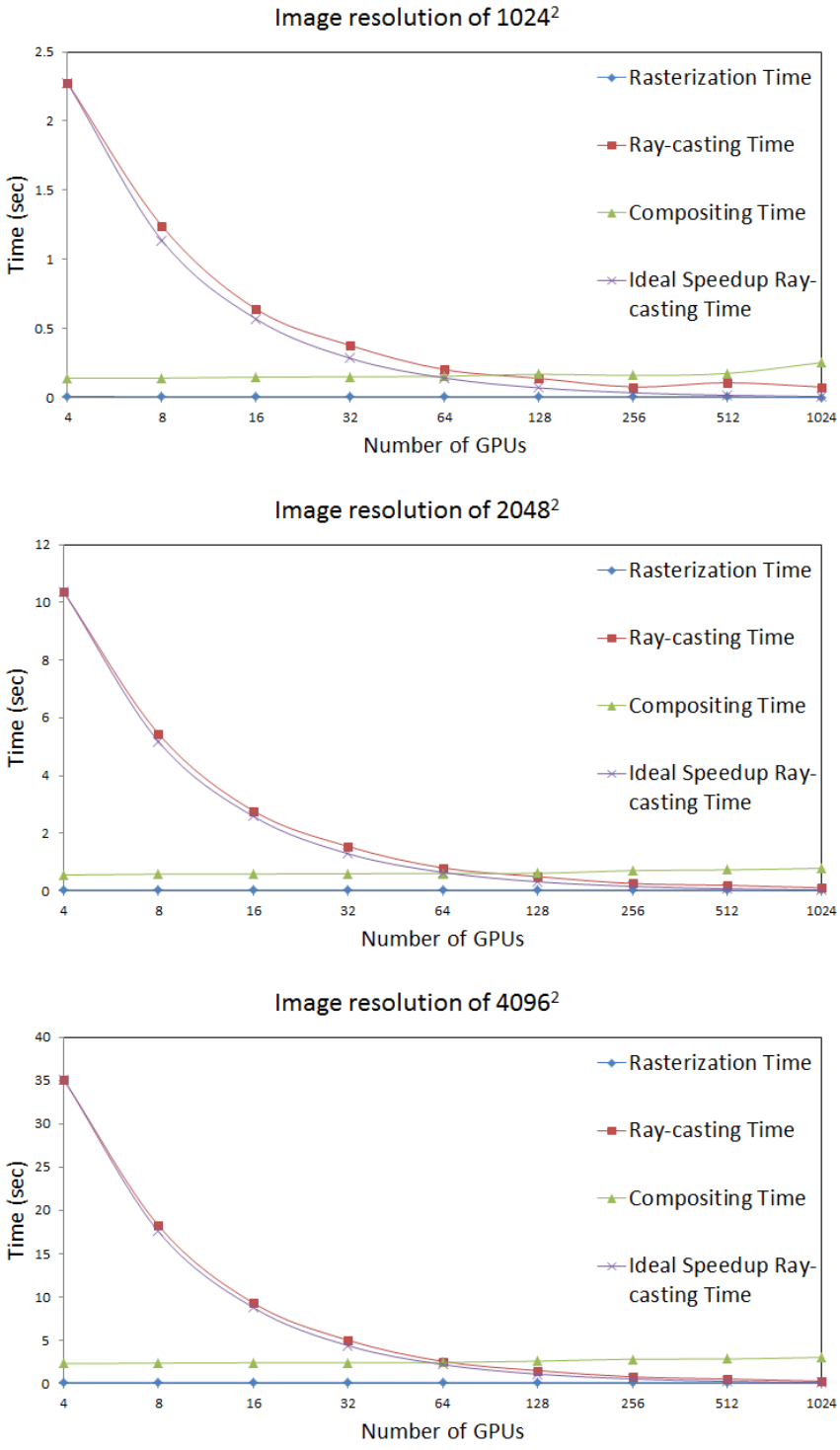


Figure 4.8. The timing results of the GCRM dataset. We measured the timing of the rasterization, ray-casting, and image compositing operations with the number of GPUs ranging from 4 to 1024 and the output image resolutions of 1024^2 (top), 2048^2 (middle), and 4096^2 (bottom). The timing results are plotted in a logarithmic scale.

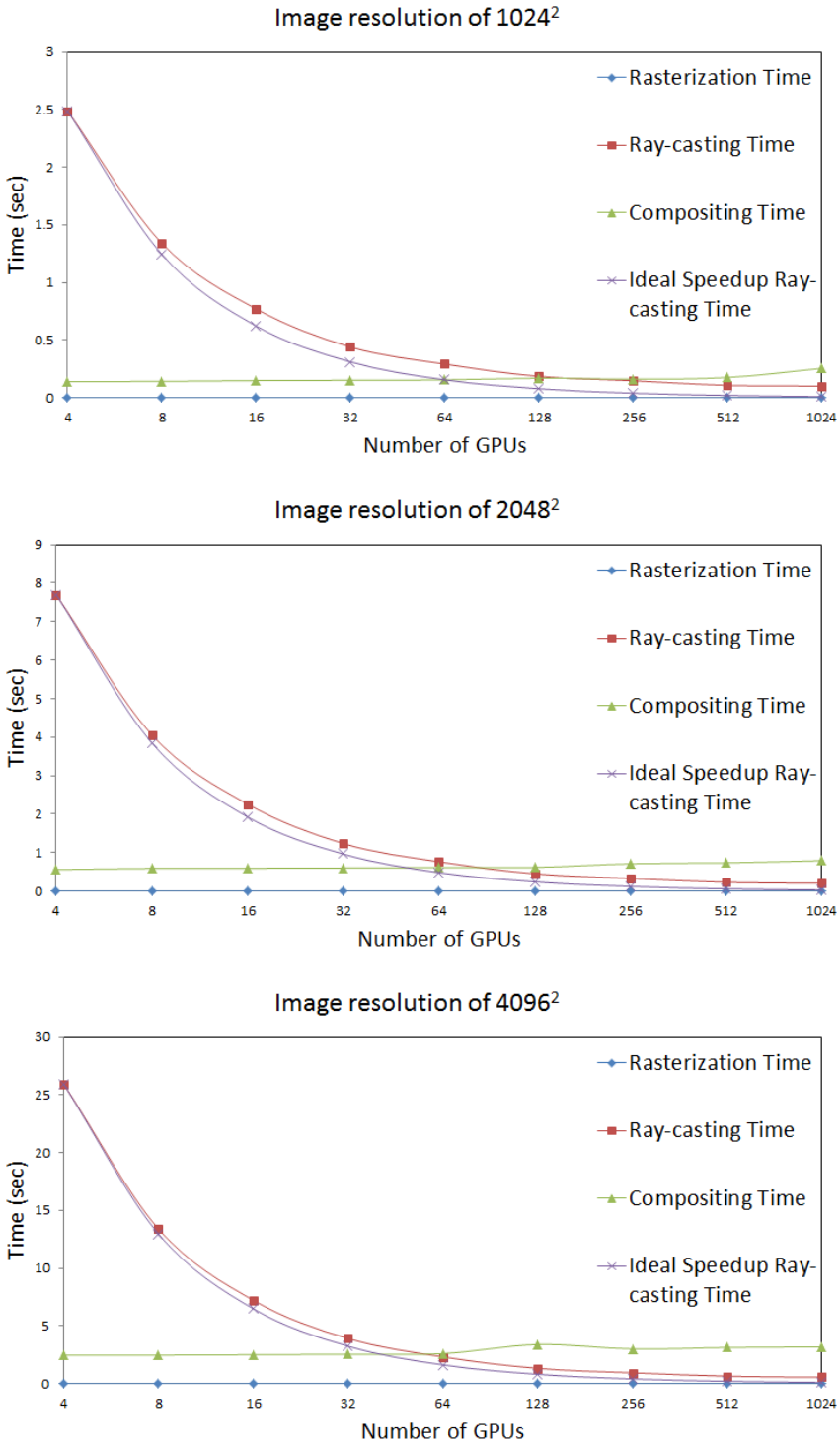


Figure 4.9. The timing results of the MPAS dataset. We measured the timing of the rasterization, ray-casting, and image compositing operations with the number of GPUs ranging from four to 1024 and the output image resolutions of 1024^2 (top), 2048^2 (middle), and 4096^2 (bottom). The timing results are plotted in a logarithmic scale.

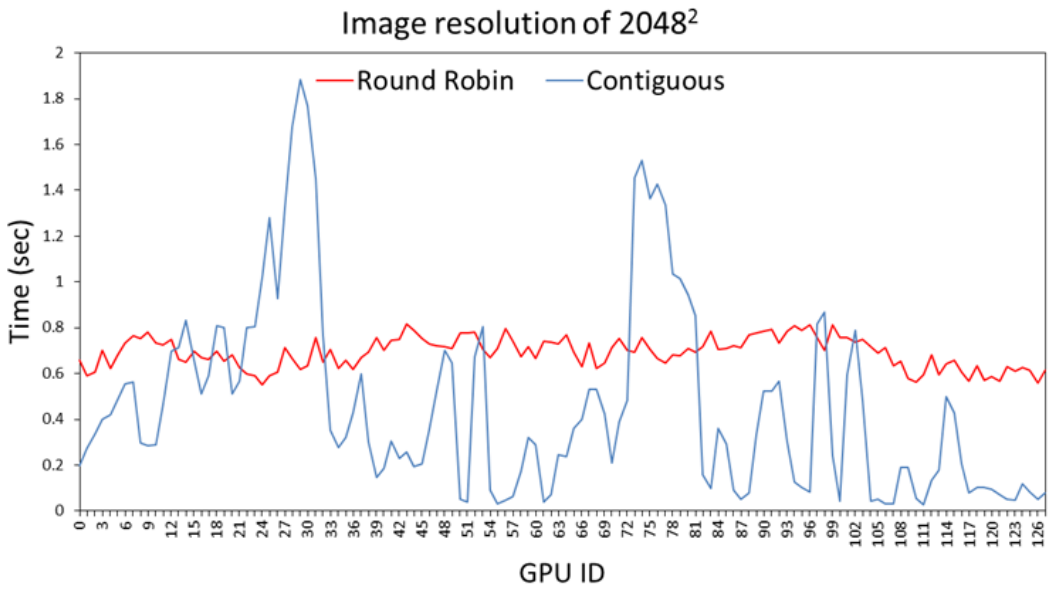


Figure 4.10. The ray-casting time for each GPU using the MPAS dataset and different data distribution schemes. The output image resolution is 2048^2. The blue and red curves correspond to the contiguous data assignment (Figure 4.7(a)) and our round robin data assignment (Figure 4.7(b)), respectively.

Figure 4.10 shows the ray-casting time for each GPU when 128 GPUs are used in rendering of the MPAS dataset with an output image resolution of 2048^2. The time is averaged over the 10 viewing directions. The difference ratio, defined as $(max\_time - min\_time)/max\_time$, is 32.5% for the red curve and 98.6% for the blue curve. This shows that the workloads are well-balanced among the GPUs from different view directions using our data partitioning and distribution scheme.

Figures 4.11 and 4.12 show the overviews and zoom-in views of the GCRM and MPAS datasets for three selected time steps. Our high-resolution parallel visualization solution delivers high-quality results that enable scientists to interactively explore fine details of the volume data. A supplementary video demonstrating an interactive exploration of each time-varying dataset using our renderer is provided at http://youtu.be/1bspVTsGSY8.

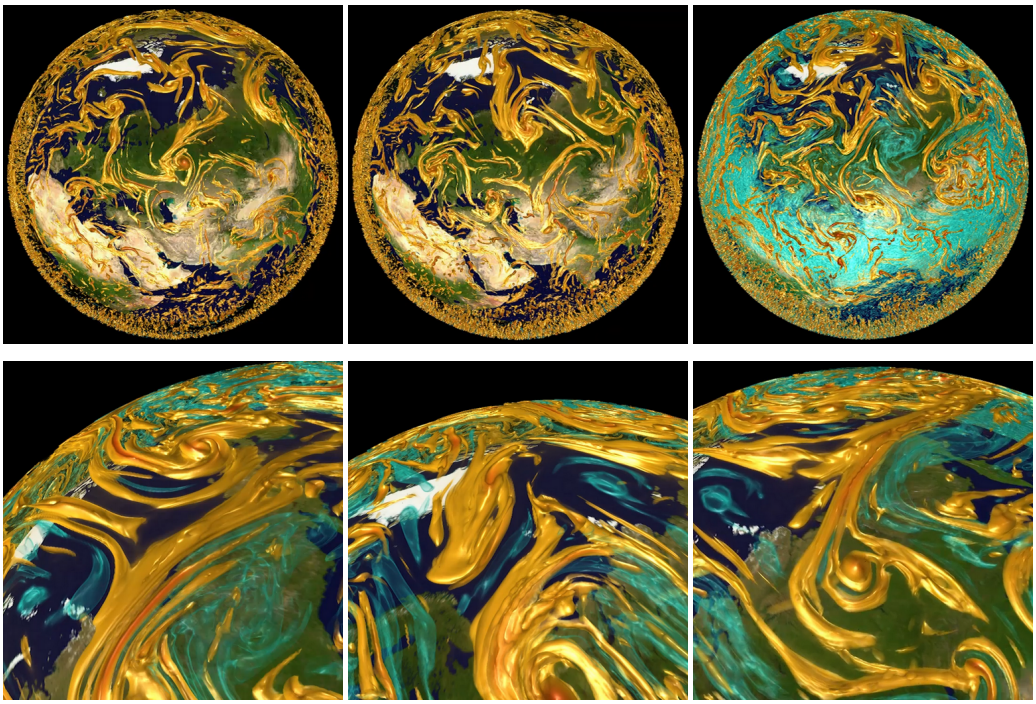


Figure 4.11. The visualization results of the GCRM dataset. The top row of images shows the whole global atmosphere vorticity variable over three time steps. The bottom row of images shows the close-up views of regions of interest.

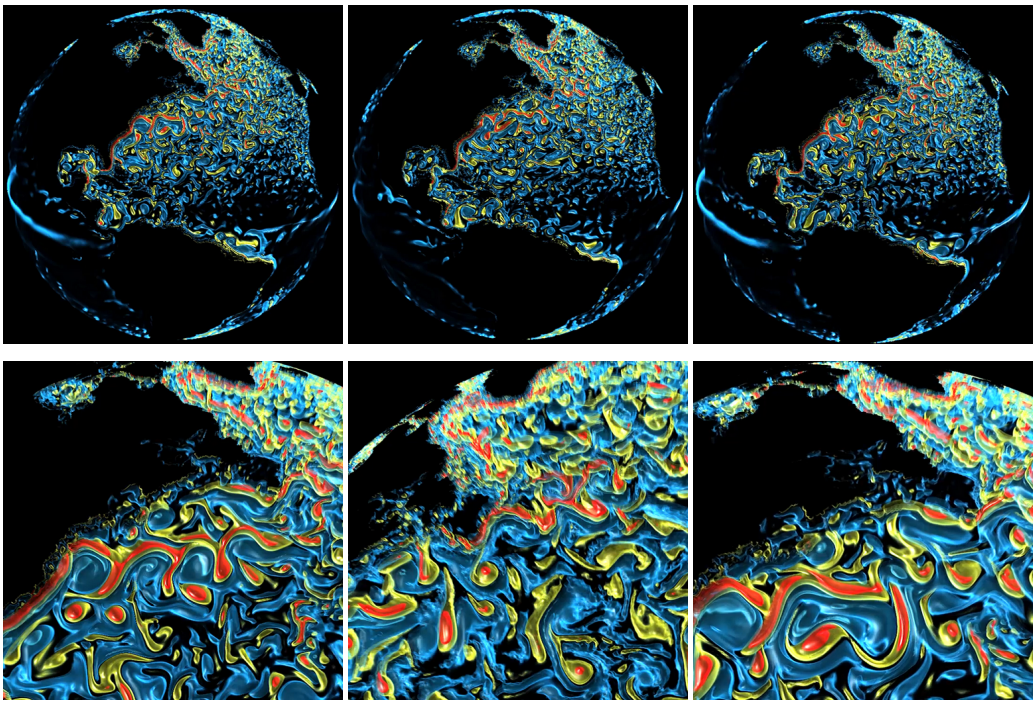


Figure 4.12. The visualization results of the MPAS dataset. The top row of images shows the whole global ocean vorticity variable over three time steps. The bottom row of images shows the close-up views of regions of interest.

4.4 Summary

This chapter presents a scalable solution for visualizing large-scale 3D geodesic grid data using massively distributed GPUs on state-of-the-art supercomputers. Based on a careful characterization of geodesic grids, it uses spherical quadtrees to partition and distribute workloads. The design achieves a balanced workload across processors, making it practical to interactively visualize large geodesic grid data. The visualization framework directly takes the original mesh as input and is ready to be integrated with simulations. The data structure for the spatial indexing scheme in this chapter is shared among all processors. In other words, each processor has exactly the same local copy of the data structure, i.e., the spherical quadtrees. The next chapter will seek and explore more advanced data partitioning and indexing scheme for more general application whose indexing data structure is distributed on thousands of processors in order to adapt to very large in-situ visualization. In the future, we plan to experiment with other data partitioning and distribution schemes, including the ones deployed by the simulations, to enable in-situ visualization. We also would like to develop a more advanced rendering cost estimation method by considering special viewing ray directions (e.g., the ones close to perpendicular or tangent to the Earth's surface).

Chapter 5

Scalable Parallel Distance Field Construction for Large-scale Applications

A distance field is defined as the following: Suppose we have a set Γ consisting of *elements* in $\mathbf{R}^3$, where elements can be polygonal objects, point clouds, or isosurfaces extracted from volumetric data. The value at each point $\mathbf{p}$ of a distance field domain $\Omega \subset \mathbf{R}^3$ is the distance from $\mathbf{p}$ to its nearest element of Γ:

$$df_{\Gamma}(\mathbf{p}) = \inf_{\mathbf{x}\in\Gamma} dist(\mathbf{x}-\mathbf{p}), \tag{5.1}$$

where the distance function *dist* is application specific, and the commonly used ones include the Euclidean distance, the Manhattan distance [90] and the chessboard distance [89].

Computing distance fields is a fundamental requirement for many algorithms of computer graphics and visualization. Distance fields are also referred to as distance transforms or distance maps. Their usage has been widely found in diverse scientific and engineering fields, such as volume graphics, computer vision, image processing, and computational geometry. Beyond their conventional applications, distance fields also receive new attention in the era of big data. Researchers have shown that distance fields can play a critical role in addressing visualization of large and complex data. For example, it is viable to effectively reduce visual clutter while accentuating visual foci via prioritizing data objects according to their distances to regions of interest [25]. Distance fields can also serve for indexing and compression approaches to managing and exploring data at extreme resolutions [31, 58, 81].

There has been extensive prior work in developing algorithms for computing distance fields. We refer readers to the references [18] and [28] for an overview of 2D and 3D distance field con-

struction and applications. However, most of this research has not focused on the requirements imposed by large-scale scientific applications:

First, there are few general distance field construction algorithms that deal effectively with large data generated from tera-to-petascale scientific simulations. It is imperative to improve the scalability of algorithms to achieve high levels of parallelism for large data processing.

Second, extrapolating current technology trends towards the exascale computing reveals the increasing disparity between I/O speed and compute speed [70]. Due to the lagged I/O speed, scientists can only store a small fraction of the data during their detailed modeling and simulation processes, thus possibly missing highly intermittent transient phenomena. A promising solution is to process the data *in-situ* on the same machines as the simulation runs, which can minimize the I/O cost and lead to exploration of data with full extent. This requires new in-situ algorithms to scale as well as simulations when executed with thousands to hundreds of thousands of CPU cores. However, such a scale has not been considered in any existing distance field construction algorithms that are almost exclusively done as an offline *post-processing* step.

Third, many real-world datasets are large in size and heterogeneous in type, structure, and semantics. As a result, it is desired to design distance field representations in support of varied data in a uniform and scalable way. Such representations can also facilitate a wide range of subsequent processing operations while minimizing data storage and transformation overhead. This requirement, however, has rarely been addressed in previous work.

It is non-trivial to design scalable solutions for computing distance fields within massive and complex data. Although researchers have exploited parallel and distributed computing in distance field construction, the existing algorithms are typically characterized with high memory access and intensive communication overhead in a distributed environment. Additionally, data elements are at risk of being unevenly partitioned and distributed over space. This produces difficulties in achieving a balanced workload among a large number of processors using the existing algorithms.

In this chapter, we present a highly scalable algorithm for computing distance field of large-scale applications. We design a new spatial hierarchical data structure, named *parallel distance tree*, that allows us to efficiently capture, track, and manage the essential information of data,

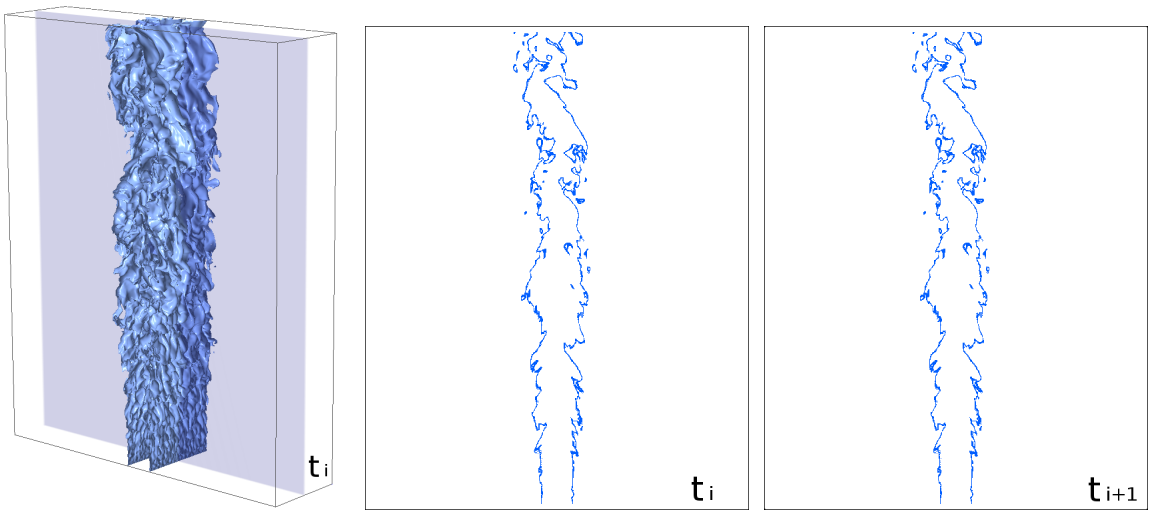
and minimize the communication and computation costs across processors to compute the distance field. In addition, our method supports multiple data types including polygonal objects, point clouds, or volumetric data. It is also flexible with different distance metrics, including Euclidean distance, City block distance, and Chessboard distance. Thus, our method can be used in a wide range of real-world large applications with minimal implementation efforts. We have conducted case studies using different 3D datasets, and shown that our achievement is generalizable and beneficial to researchers from different areas.

We have also integrated our method with real-world large simulations. This enables scientists to compute distance fields in-situ and capture highly intermittent and detailed phenomena that were hardly perceived in post-processing. Our method does not depend on any particular architectures, and the experiments have been conducted on state-of-the-art supercomputers. Our results have scaled up to 69,120 CPU cores of parallel distance field constructions, and clearly demonstrated the improvement over the previous state-of-the-art.

5.1 Background

Figure 5.1 (a) shows an example of application requirement from a large-scale combustion simulation. The bounding box corresponds to the 3D simulation domain, and the blue surface is an isosurface extracted from one time step of data. Using this surface as Γ, a scientist wants to compute the distance from any point inside the domain Ω to the surface.

It is very challenging to achieve optimal parallel efficiency to compute distance field with large and complex data in a distributed environment [10, 33]. This is first because distance field construction is characterized with high memory access. As shown in Equation 5.1, in the worst case an exhaustive search of elements may be required to find the nearest element of a point. As such, in a distributed environment where each processor may hold a portion of the elements, this can incur intensive message exchanges among processors. In addition, data elements can be unevenly partitioned and distributed over space, and thus it is difficult to achieve balanced workload among processors.



(a) (b) (c)

Figure 5.1. (a) shows a 3D rendering of an isosurface extracted from a combustion dataset at a time step t_i. We may only need to search a small portion of the surface to compute the distance at any point. (b) and (c) show the slices of the surface evolved at two consecutive time steps, t_i and t_{i+1}. The difference between the two surfaces is around 0.1% of the surface area. Only a small fraction of distance field may need to be updated accordingly.

5.1.1 Spatial and Temporal Coherence

To address these issues, we are inspired by the spatial and temporal coherence inherent in the data of modelings and simulations. Given the example shown in Figure 5.1, we have the following observations. First, by leveraging the spatial coherence (Figure 5.1 (a)), we may only need to search a set of nearby elements to compute the distance at any point. In a distributed environment, this implies that it is possible for one processor (PE) [1] to only communicate with a fewer number of vicinal processors, thus reducing communication costs. Furthermore, Equation 5.1 shows that the overall computation cost is proportional to the number of elements, and thus we may achieve more balanced workload by evenly partitioning and distributing the workload with respect to the elements and the distance field domain. Second, by leveraging the temporal coherence (Figure 5.1 (b) and (c)), we may only need to update a small fraction of the distance field with respect to the changes of elements, which can further reduce the communication and computation costs over time.

[1] A processor refers to a single-core processor or a core in a multiple-core processor.

5.1.2 Octree-based Distance Field Construction

At first glance, it may appear straight-forward to first build a spatial hierarchy, such as an octree, to index the set Γ of elements by subdividing the distance field domain Ω adaptively, and then, for a point $\mathbf{p}$, we can query the tree to find its nearest point among the elements. However, the elements within the same octree node as $\mathbf{p}$ may not contain the nearest point to $\mathbf{p}$. Thus, we need to continue to traverse the nearby octree nodes that can be at different tree levels, and this operation will require additional overhead.

To solve this problem, Strain [76] modified the way to build an octree to facilitate the nearest point query. For each octree node C with the center $\mathbf{c}$ and the edge length $2r$, Strain defined the *concentric triple* (or shortened as *triple*) T of C as:

$$T = \{\mathbf{x} \in \mathbf{R}^3 : dist(\mathbf{x} - \mathbf{c}) \leq 3r\}. \tag{5.2}$$

Based on this definition, an octree is built from a root node that encloses Γ. Each tree node has an element list, and initially the root's element list contains all elements of Γ. A tree node is recursively split if its triple intersects Γ. For each new child node, we check every element in the parent's element list, and add it to the child's element list if the element intersects the child's triple. We continue this recursive splitting until a tree node has an empty element list or the tree depth reaches a maximum criterion.

After building such an octree, for a point $\mathbf{p}$, we can locate the leaf node C containing $\mathbf{p}$, and efficiently compute the distance of $\mathbf{p}$ to Γ through three steps:

1. Find the minimal distance m_1 from $\mathbf{p}$ to the elements of Γ in the element list of C.

2. If the element list of C is empty or m_1 does not satisfy

$$\{\mathbf{x} \in \mathbf{R}^3 : dist(\mathbf{p} - \mathbf{x}) \leq m_1\} \subset T, \tag{5.3}$$

   where T is the triple of C, it means that there are possibly some elements outside T but having the minimal distance to $\mathbf{p}$. Then we find the minimal distance m_2 from $\mathbf{p}$ to the elements of Γ in the element list of the parent node C' of C.

3. If m_2 does not satisfy

$$\{\mathbf{x} \in \mathbf{R}^3 : dist(\mathbf{p} - \mathbf{x}) \leq m_2\} \subset T', \tag{5.4}$$

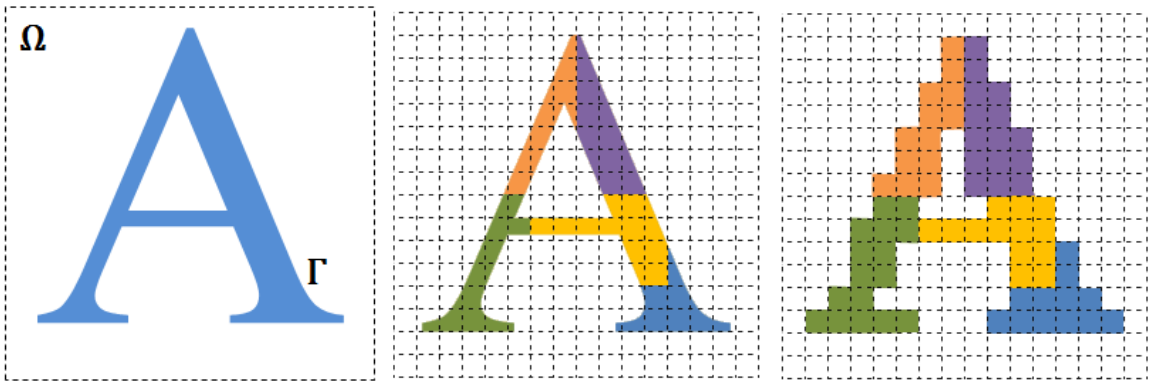
where T' is the triple of C', then we find the minimal distance m_3 from $\mathbf{p}$ to the elements of Γ in the element list of the grandparent node C'' of C.

The correctness of this procedure has been proved in [68, 76]. This procedure can provide the *exact* distance value at any $\mathbf{p}$. Alternatively, we can also first compute the distances at the octree vertices and then obtain the distance at $\mathbf{p}$ using interpolation. Such an octree equipped with the vertex distances and the element lists is called a *distance tree* that can provide both exact and interpolated distance values [76].

5.2 Parallel Distance Tree

Our intuition is to extend the octree-based method to index data distribution across processors and exploit the spatial and temporal coherence. A straight-forward solution is to let each processor first compute a global distance tree, and then use the tree to query the vicinal processors to construct the distance field collaboratively. However, this requires each processor to collect the information of global data distribution before global tree construction. Such a collective operation often incurs a significant amount of communication. In addition, a full-grown global distance tree is typically too large to be handled by a single processor. Hence, it is non-trivial to efficiently construct and maintain distance tree in parallel, and use it to derive workload assignment and communication schedule.

We introduce a novel distributed spatial data structure, named *parallel distance tree*, to index the set Γ of elements and the distance field domain Ω for parallel distance field construction. Based on our assumptions (Section 5.2.1) and data structures design (Section 5.2.2), our strategy is to first build a coarse global distance tree at each processor (Section 5.2.3). This step only needs a coarse-grained description of global data distribution at each processor, and the memory and communication cost is marginal. We use the global tree to derive a balanced assignment of local trees among the processors (Section 5.2.4). Each processor can use its local tree to independently find the processors that it needs to communicate with, and establish the communication schedule without message exchanges (Section 5.2.5). Each processor then independently constructs a full-grown local instance of the underlying global tree (Section 5.2.6). A local instance is also a distance tree that contains the information of local and remote ele-



(a) (b) (c)

Figure 5.2. (a) shows a set Γ of elements in the domain Ω. (b) shows that the elements are partitioned and distributed among five processors in a block fashion. The assignments of Γ among the processors are represented in different colors. Each processor fills a bitmap according to its local element block distribution. After a collective reduction operated on the bitmap, each processor obtains the information of the global distribution of the element blocks and the empty blocks, as shown in (c).

ments, which enables the processors to collectively compute the vertex distances of their local distance trees with the exploitation of data parallelism (Section 5.2.7). Our method also takes advantage of temporal coherence in time-varying simulation data so that only a small subset of the distance tree can be updated (Section 5.2.8). The final distance field is organized and stored in a distributed fashion that can facilitate large data analytics. We have integrated our method with a simulation (Section 5.2.9), and explored performance acceleration (Section 5.2.10).

5.2.1 Assumptions

Given a set Γ of elements in $\mathbf{R}^3$, we assume that the distance field domain Ω is rectangular and encloses Γ, as shown in Figure 5.2 (a). (For clarity, we use 2D elements, domains and quadtrees in the figures and examples.) The scalability of parallel distance field construction depends on both partitioning and distribution of Γ and Ω.

Partition and distribution of Γ is typically application specific. Without loss of generality, we assume that Γ is partitioned in a representative block fashion [57]. Each block is rectilinearly oriented and can intersect multiple elements. The bounding boxes of all blocks have the same size and shape. The blocks can be assigned to the processors in different fashions. For post-processing, the block assignment can suit the needs of parallel distance field construction, and

the amount of elements at each processor can be roughly the same to achieve balanced workload. For in-situ processing, the elements of Γ are generated during modelings or simulations. The partition and distribution of Γ is dictated by the applications that generate data, which can be highly uneven among processors: some processors may have multiple blocks of elements, while some processors may have none.

Given the block-based data partition scheme, we assume that the rectangular domain Ω consists of a set of blocks, $\{1,...,b_x\} \times \{1,...,b_y\} \times \{1,...,b_z\}$, where b_x, b_y, and b_z are the numbers of blocks along each axis. A block can be an *element block* that intersects elements, or an *empty block* without any elements. Each element block is assigned to a processor. Figure 5.2 (b) shows an example that the set Γ in Ω are divided and assigned to five processors.

5.2.2 Data Structures

The information on global data distribution is imperative for distance field construction. However, it is infeasible to collect precise distribution of elements at each processor for a large dataset. We address this issue by letting each processor gather the coarse-grained information at a block level. To this end, the first data structure that each processor has is a bitmap which records the global distribution of element blocks and empty blocks. The bitmap contains $b_x \times b_y \times b_z$ bits. Each bit corresponds to a block in Ω, and is set to zero or one for an empty block or an element block, respectively. Each processor first initializes its bitmap by filling 0s. Then, if a processor's element intersects a block in Ω, the corresponding element block is marked as one in the bitmap. Finally, each processor combines the bitmaps from all processors. This can be easily implemented using a *collective reduction* routine with the bitwise OR operation, such as the MPI_ALLREDUCE function. The size of the bitmap is marginal. For example, a bitmap of 128KB can represent more than one million blocks which is sufficient for current large-scale supercomputers. Each processor then obtains an identical bitmap that records the global distribution of the element blocks and the empty blocks in Ω. Figure 5.2 (c) shows the bitmap of the global block distribution obtained by each processor, given the partitioning and distribution of Γ among five processors in (b).

Each processor also has two arrays to store the information of elements. The first array is an element array, *elementarray*, which records the local elements. The element array is empty

if a processor does not have any local elements. The second array is an element block array, *elementblockarray*, which records the global element blocks. Each element block *blk* contains its index *blk.id* in the bitmap and the processor ID *blk.proc* to which the *blk* is assigned.

Finally, each processor has a local instance of the underlying global distance tree. A local instance is also an octree with the standard *linear octree* representation [21]. It is part of the octree overlapping all triples of octree nodes owned by a processor (Section 5.2.6). It consists of two arrays, the vertex array and the node array. The vertex array, *vertexarray*, records the octree vertices. Each vertex *v* contains:

- *v.key*: the locational key (to be explained below).
- *v.node*: the index of the node that creates *v* in the node array.
- *v.coord*: the coordinate of *v* in Ω.
- *v.distance*: the distance value at *v*.

The node array, *nodearray*, records the octree nodes. Each node *n* contains:

- *n.key*: the locational key.
- *n.boundingbox*: the geometry of *n*'s bounding box.
- *n.vertices*: the indices of *n*'s vertices in the vertex array.
- *n.elements*: the indices of the elements that intersect *n*'s triple in the element array.
- *n.elementblocks*: the indices of the element blocks that intersect *n*'s triple in the element block array.
- *n.processors*: the IDs of the processors whose element blocks intersect *n*'s triple.
- *n.location*: the flag indicating if *n* is *local* or *remote*.
- *n.parent*, *n.children*: the indices of *n*'s parent and children in the node array.

We use the *locational key* [21, 4], also known as the *shuffled zyx key* [92], to encode and unambiguously distinguish octants[2]. It is a commonly used linear octree technique For an octant at the depth m, the locational key is a bit string, $z_1y_1x_1z_2y_2x_2\cdots z_my_mx_m$, indicating the path from the root to this node. The convention is that if the bit x_i is set to 0, the child at the depth i covers the left side in x of its parent; otherwise, it covers the right side. The interpretation of the y and z bits are similar. The key of each vertex is computed using the same convention. If we use a 4-byte integer for a locational key, the maximum tree level is 10. An 8-byte long integer can be used to achieve a deeper tree level of 20.

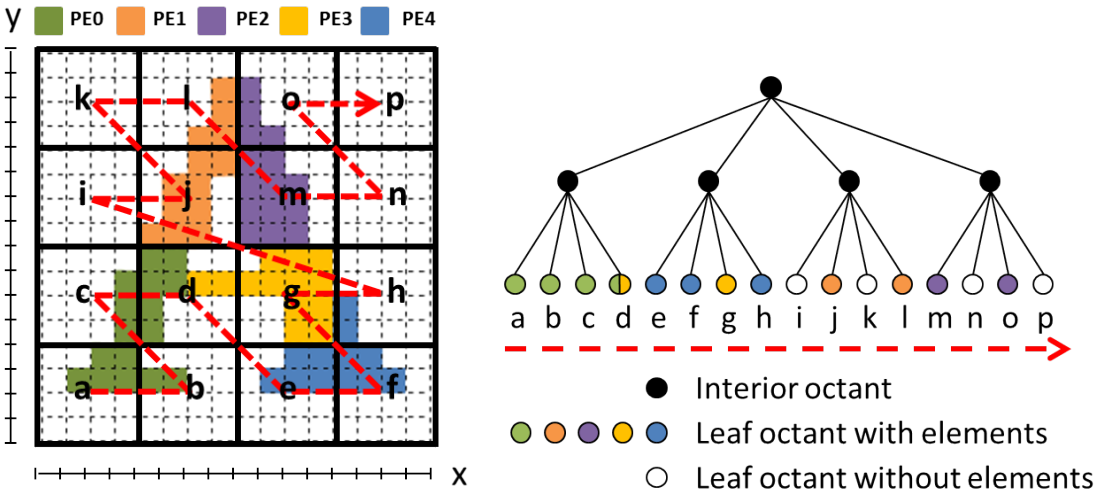
5.2.3 Initial Coarse Global Distance Tree Construction

Each processor independently builds an initial coarse distance tree to subdivide Ω with respect to the information of the element blocks provided by the bitmap. Each processor builds the tree in a similar way as the one in Section 5.1.2. The main differences are that the tree is built in the breadth-first order and only the element blocks are considered. Each processor starts with a root node, *root*, to cover Ω. The $root.elementblocks$ contains all the element blocks, and the $root.processors$ contains the IDs of the processors that have the element blocks. Then we recursively split the nodes in the breadth-first order if a node's triple intersects the element blocks. For each new child, we check every element block *blk* in the parent's $elementblocks$ list, and add *blk* into the child's $elementblocks$ if *blk* intersects the child's triple. We also check every processor *proc* in the parent's *processors* list, and add *proc* into the child's *processors* if *proc*'s element blocks intersect the child's triple. We continue this procedure until the ratio of the leaf node number to the processor number is greater than ε ($\varepsilon = 3$ in our current implementation) or the maximum tree depth has been reached. Because all processors have an identical bitmap, they generate an identical initial global distance tree after this step.

5.2.4 Leaf Octants Assignment

Each leaf octant in the initial global tree corresponds to a region of Ω (see Figure 5.3 (a)), and is associated with the workload of computing the distance field. We assign the leaf octants among processors. The leaf octants are not necessarily continuous in *nodearray*. We scan *nodearray*

[2]We use octant and octree node interchangeably.



(a) (b)

Figure 5.3. (a) shows that the decomposed global distance tree leaf octants (as indicated by the black square) are assigned to the PEs whose element blocks intersect the leaf octants bounding box. It is possible that one leaf octant is assigned to multiple processors, such as the leaf octant *d*. All the leaf octants are naturally ordered based on their locational keys, so that the traversal of all the leaf octants follows a Z-order space-filling curve that preserves the spatial locality of the leaf octants. (b) shows a quadtree corresponding to the spatial partition of (a). The traversal of the leaf octants from left to right in (b) is equivalent to the zigzag one in (a).

and collect the leaf octants into a temporary array *leafarray* while preserving their orders in *nodearray*. The leaf octants are naturally ordered according to their locational keys because of the linear octree technique. This order is identical to the pre-order traversal of the leaf octants as shown in Figure 5.3 (b). If we traverse the leaf octants in this order in Ω, we follow the well-known *Z-order space-filling curve* which can cluster the spatial nearby octants together, as shown in Figure 5.3 (a). This property facilitates partitioning and distribution of the workload among processors using a two pass procedure.

In the first pass, we assign the leaf octants whose *processors* lists are nonempty. Each of these leaf octants has the elements within its region. We assign such an leaf octant to the processors in its *processors* list to minimize data movement. As show in Figure 5.3, some leaf octants may be assigned to one processor, such as e, f and h that are assigned to PE_4, and some leaf octant may be assigned to multiple processors, such as d that is assigned to PE_0 and PE_3.

In the second pass, we assign the leaf octants with the empty *processors* lists. Each of these

leaf octants has no elements within its region, and we call it an empty leaf octant, such as the octants i and k in Figure 5.3. There are two cases for assigning the empty leaf octants:

In the first case, all processors have already been assigned with workload in the first pass. For example, as shown in Figure 5.3 (b), all five processors have been assigned with the leaf octants. Then, we assign each empty leaf octant oct to the processor that is responsible for the oct's neighboring leaf octants in $leafarray$ and these neighboring leaf octants have the closest ancestor with oct. For example, in Figure 5.3 (b), the empty leaf octants i and k will be assigned to PE_1, because the octants i-l are neighbors with the same ancestor, and j and l are assigned to PE_1 in the first pass. Similarly, the empty leaf octants n and p will be assigned to PE_2.

In the second case, some processors have not been assigned workload in the first pass. This is a typical case for in-situ processing, where the region-of-interest can be extracted or identified on only a small fraction of processors. In this case we adopt a simple rule that we evenly partition and distribute the empty leaf octants according to their order in $leafarray$ among the rest of processors. This rule can generally provide well balanced workload in practice because (1) we build the initial distance tree that has a sufficient large number of octants with respect to the processor number, and (2) the leaf octants are ordered along the spacing-fill curve. For example, in Figure 5.3, assume there are another two processors, PE_5 and PE_6, who do not have any elements and have not been assigned workload in the first pass. Then we assign the octants i and k to PE_5, and n and p to PE_6, and the spatial locality of the assignment is ensured.

Each processor independently performs the same procedure to compute the workload assignment. Thus, each processor obtains an identical assignment of leaf octants of all processors. Then a processor goes through each octant oct, sets $oct.location = local$ if oct is assigned to itself, and otherwise sets $oct.location = remote$.

5.2.5 Communication Schedule

After the leaf octant assignment, each processor PE_i takes charge of a sub-region ω_i of Ω which is the union of its assigned octants. In order to compute the vertex distance, PE_i first computes the minimum distance from each local vertex to the local elements in PE_i's $elements$ list. However, such a local minimum vertex distance is not necessarily the global minimum. Each processor needs to further go through the nearby but remote octants, and compute the

minimum distances from the local vertices to the elements in the remote octants. To achieve this, PE_i needs to send its local vertices to its neighboring processors, outsource the distance computation to those processors, gather the results and finally find the minimum distance value.

A communication schedule plays an indispensable role that helps each processor to identify the neighboring processors that it will exchange the vertices with. Each processor PE_i builds a *sent_table* and a *receive_table* for the communication schedule. Each entry of the tables contains the IDs of the remote processors that PE_i needs to exchange data with, and the corresponding data buffer for sending or receiving. The communication schedule is built based on the information stored in the initial global distance tree:

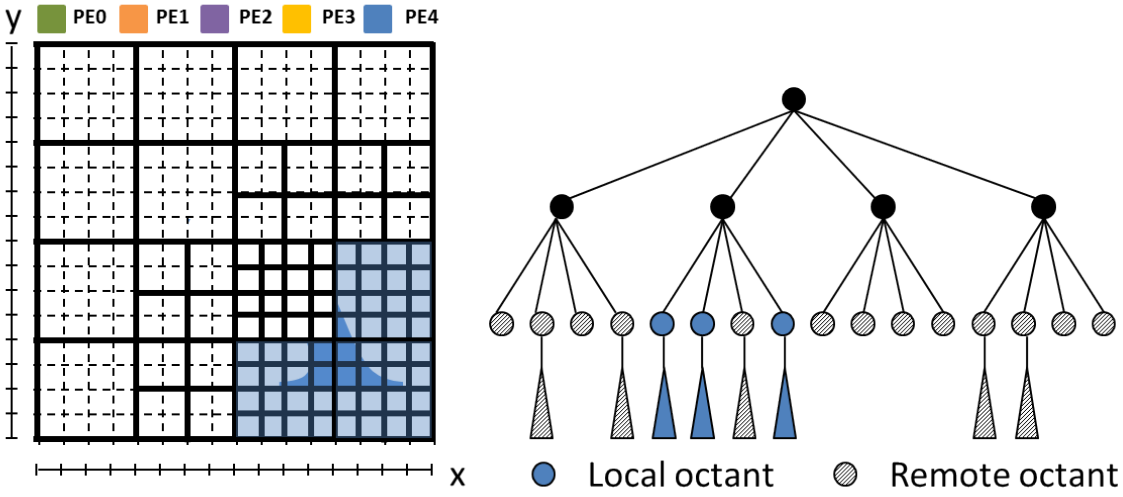
To create *send_table*, PE_i scans its assigned leaf octants. For each of leaf octant, $leaf$, the $leaf.processors$ list contains the remote processors whose element blocks intersect $leaf$'s triple. These processor IDs are then added in *send_table*, because their elements need to be considered and PE_i needs to send local vertices to them. If $leaf.processors$ is empty, PE_i checks each of $leaf$'s ancestors along the path to the root until PE_i finds the first nonempty *processors* list, and adds the processor IDs in *send_table*.

For *receive_table*, PE_i simply returns an empty one if it has no local element, as no neighboring processors send data to PE_i in this case. Otherwise, PE_i scans the global tree and identifies the remote leaf octants or their closest ancestors whose *processors* lists contain PE_i's ID. This means that PE_i's element blocks intersect the triples of these remote leaf nodes, and their host processors send message to PE_i to request distance computation. In this case, these processor IDs are added in *receive_table*.

5.2.6 Full-grown Local Distance Tree Construction

The initial global tree is constructed according to the information of element blocks, and thus is coarse-grained. To compute the distance field, each processor independently grows the initial distance tree with respect to its local elements and the global element blocks in two passes:

In the first pass, each processor PE_i propagates its local elements along the initial global tree, where each tree node's $elements$ list records the elements that intersect its triple. After this pass, the $elements$ and $elementblocks$ lists of each tree node are filled with the elements and the element blocks intersecting its triple, respectively.



(a) (b)

Figure 5.4. The light-blue shaded region in (a) is a set of local leaf octants (4x4 square) assigned to PE4. (b) shows the corresponding full-grown local distance tree at PE4. The blue triangles are local distance trees refined in the assigned global tree leaf octants (blue circles), while the grey triangles are also the refined local distance trees in the rest of the global tree octants whose triple intersects the local elements of a processor.

In the second pass, each processor PE_i grows its distance tree in a similar way as the one in Section 5.1.2. For each one of PE_i's local leaf octants, it is recursively split if there are still elements or element blocks within its triple. During the splitting, if the parent's elements or element blocks intersect the child's triple, they will be added into the child's *elements* or *elementblocks*, respectively. For the other remote leaf octants in the initial tree, they are recursively split in the similar way but with only considering PE_i's *elements*. The recursive procedure stops when the maximum tree depth is reached or no more octant can be split.

Figure 5.4 shows the full-grown local distance tree at PE_4 from the example in Figure 5.3. We can see that, apart from the refined local octants, some neighboring remote octants are refined according to PE_4's elements as well. The refinement facilitates the following distance field computing.

5.2.7 Distance Field Computing

Having constructed the communication schedule and built full-grown local distance tree at each processor, we continue to compute the vertex distances in parallel.

First, each processor independently computes the distance values from the local vertices to the local elements using the method in Section 5.1.2 that is flexible with different distance functions. Given a local vertex v, it can be shared by multiple octree nodes. We assume that the node n contains the element that has the local minimum distance *min_dist* to v. *min_dist* is the global minimum if and only if it satisfies

$$min\_dist \leq 2 \times n.radius \tag{5.5}$$

Otherwise it implies that there possibly exists an element *elm* closer to v, and *elm* can be in the triple of n's parent but not the triple of n. Strain [76] showed that to find the minimum, searching the element list of leaf node's parent is sufficient for the chessboard distance, and searching the element list of leaf node's grandparent is sufficient for the Euclidean distance. Figure 5.5 shows the case when Equation 5.5 is violated and searching the element list of the parent is required. In the case of our parallel distance tree, this means that for the chessboard distance, v needs to be sent to the processors in the *processors* list of n's parent; and for the Euclidean distance, v needs to be sent to the processors in the *processors* list of n's grandparent. Thus, for each v whose local *min_dist* does not satisfy Equation 5.5, its locational key is added into the entries of its destination processors in *send_table* (Section 5.2.5). Given our communication schedule, these vertex keys are exchanged in a bundle. This significantly reduces the communication cost when the number of exchanged vertices is large. In addition, a locational key is much more compact than coordinate values and can be used to further reduce the communication cost.

After exchanging the vertex keys, each processor translates the received vertex keys into the coordinate values and compute the minimum distance from every remote vertex to its local elements. Once each processor completes calculation, they exchange the results back to their neighboring processors according to the communication schedule but in a reverse direction. Finally, after receiving the remote vertex distances from its neighboring processors, each processor updates the *min_dist* values if the remote vertex distances are smaller. Up to this point, all *min_dist* values are the global minimums. We note that the vertex distances are stored in

a distributed fashion and the local distance fields of all processors collectively cover the entire domain. We can use the vertex distances to interpolate the distance at any point **p** via the distance tree method in Section 5.1.2. On the other hand, our method to compute vertex distances can be easily applied to compute the exact distance at any point. Specifically, computing the exact distance of any point is almost the same as computing the distance of a vertex, but with extra care. For vertex distance, each processor already knows its local vertices, so it can immediately start distance computing following the procedure of Section 5.2.7 after the local tree is constructed (Section 5.2.6). However, for any point, we take additional step to use the global tree to quickly locate a region (leaf octant) containing this point, and assign this point to the processor which is assigned this region. Then we treat this point just like a normal vertex and follow the procedure of Section 5.2.7 to compute the exact distance. Given n arbitrary points in Ω, all processors can load and locate these points using the global tree in parallel, and then pick up the ones that belongs to their own assigned octants. This is considered as a data partition step. Finally, all processors compute the distance value of their own points in parallel using procedure in Section 5.2.7. Therefore, our parallel distance tree can provide both exact and interpolated distance values at users' disposal according to their precision requirement.

5.2.8 Tree and Distance Field Updates

Detailed modelings and simulations are typically characterized by temporal coherence in their output time steps. This gives us two important implications. First, only a marginal portion of the tree structure needs to be updated with respect to the field evolution between two consecutive time steps. Second, we do not need to update the distance field of a region if there are no field changes within the region's triple.

Based on these implications, we can significantly reduce the cost for updating the tree and the distance field compared with the cost of initial tree construction and full distance field computing. At a time step when distance field construction is evoked, each processor first updates its local elements, fills its local bitmap, and collectively reduces the global bitmap in the same way as we described in Section 5.2.2. Each processor also records the global bitmap of the previous time step. By comparing the global bitmaps of the current and previous time steps, each processor can detect the blocks that are changed. For a block whose value is changed from

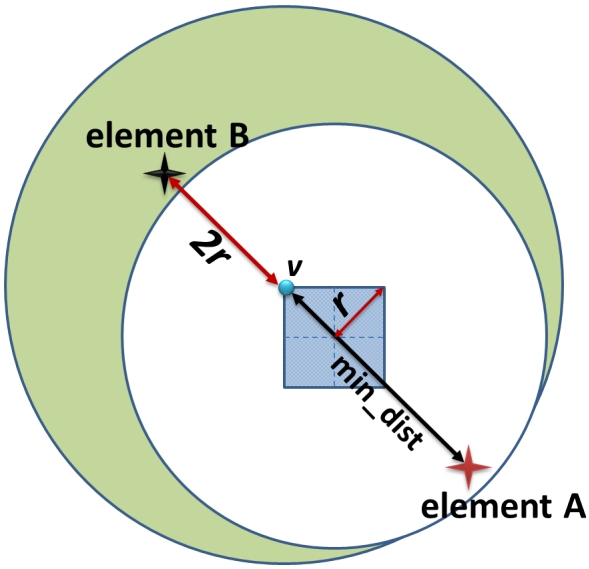


Figure 5.5. The blue shaded square corresponds to the current node of interest, n, and its radius is r. The white circle is the *concentric triple* of n. For simplicity, A is the only element within n's *concentric triple* and hence the only entry in $n.elementarray$. We assume that the minimal distance min_dist from vertex v to A within the triple is between $2r$ and $3r$. Thus, there is possibly an element B which is outside the white circle but whose distance to v is shorter than min_dist. In this case, it is necessary to continue searching the elements of n's parent to find B.

one to zero in the bitmap, it means that this block contains no elements at the current time step, and needs to be removed from the distance tree. For a block whose value is changed from zero to one, it means that this block contains the elements and needs to be added into the tree.

For a processor that identifies its local changes, we also need to update its element list. For a point or volumetric dataset, we can easily track the updated elements according to their coordinates. For the polygonal data, however, we do not track the updated elements, but simply replace the elements with the newly identified polygons for maintaining the accuracy of Euclidean distance computing.

If a processor detects that its lists of element blocks or elements are changed, it then scans the distance tree, and removes/adds the element blocks or the elements from/into the tree nodes. After modification, for a leaf node that contains no element blocks or elements, it needs to be merged with its siblings in a bottom-up fashion, until the new leaf node satisfies the stopping criteria of tree splitting (Section 5.1.2). For a leaf node that contains new element blocks or elements, it needs to be further split, until the stopping criteria of tree splitting are satisfied.

In practice, the amount of leaf nodes that need to be adjusted is marginal compared with the overall tree size. In our current design, we do not change the assignment of spatial regions among the processors. This means that, for each processor, the assigned regions remain the same, but the distance tree with the regions might become coarser or finer. After tree updating, only the processors having the elements changed in its local and/or triple regions need to update their communication schedules. The rest of steps of communication schedules updating and distance field computing are the same as what we described in Sections 5.2.5 and 5.2.7.

5.2.9 Integration with Simulation

We integrate parallel distance field construction with a petascale combustion simulation based on the APIs developed in our previous work [101]. Through the APIs, the simulation provides the size and coordinates of each processor's global domain and local partition. The simulation also provides the pointer to the buffer of the local field data. The distance field construction module is initialized and invoked by the solver at a given rate. Our integration method can avoid interference between the simulation and the distance field construction.

5.2.10 Acceleration

The performance of our method can be further improved using GPU or multithreading acceleration. The most intensive computation in our method is to compute the distance values at the tree vertices. For example, the Euclidean distance calculation between the points and the polygons can be accounted for nearly 90% of the computation cost of a processor. However, we can easily parallelize the distance calculation using GPUs or multithreading. This is first because the computation associated with the vertices are independent of each other. Secondly, the linear representations of the distance tree and the elements can facilitate concurrent data access. In our current implementation, we use MPI for distributed computing and process-to-process communication, while OpenMP with four threads per MPI process is used for further acceleration: each processor mentioned in our algorithm maps to an MPI process, while each thread is assigned to be responsible for a subset of vertices owned by an MPI process, and the linear tree and the element arrays of an MPI process are shared among these four threads. This approach has achieved noticeable speedup in practice.

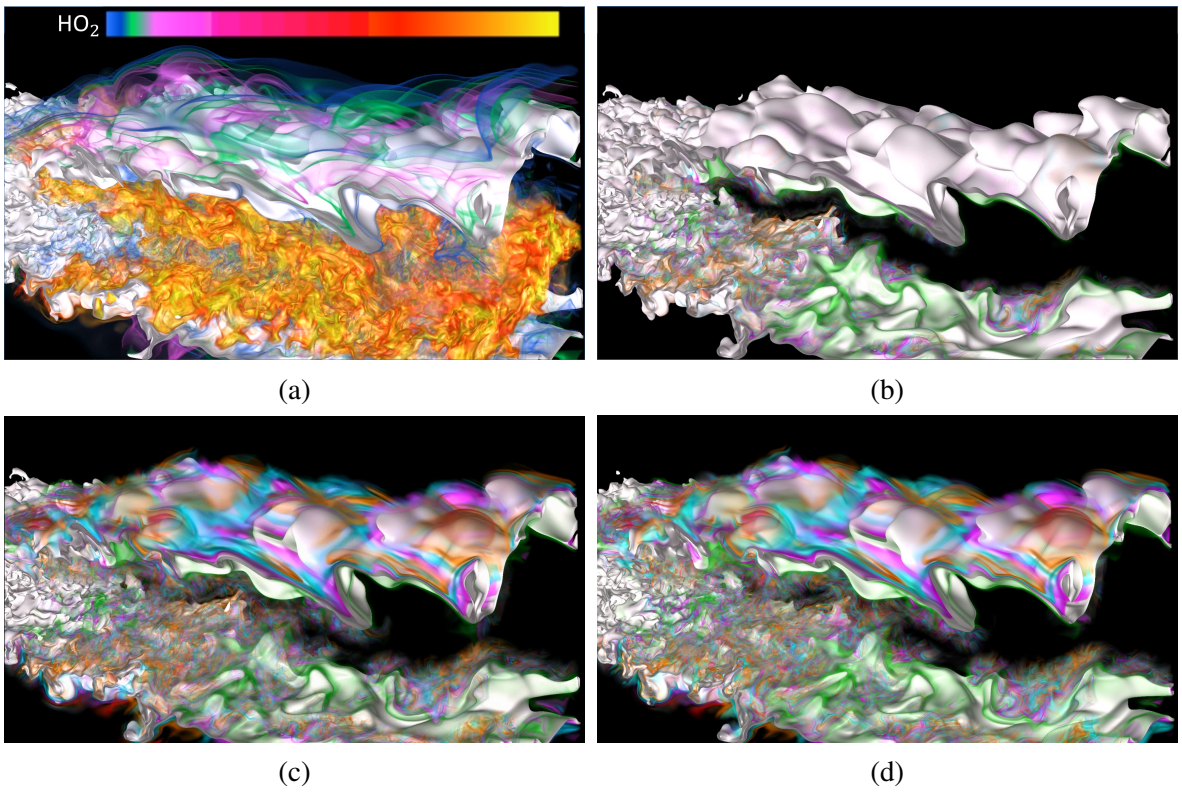


Figure 5.6. (a) shows the simultaneous rendering of two variables, temperature (T) and hydroperoxyl radical (HO_2) of the first combustion simulation dataset. The T surface at an isovalue is white; the HO_2 variable is volume rendered. We construct the distance field based on the isosurface. (b) and (c) show the scientists can interactively change the distance threshold and control the amount of information of HO_2 displayed around the T surface to better observe variable relationships. Following this natural coordinate system of a flame, the scientists can see the interaction of small turbulent eddies with the preheat layer of a turbulent flame, a region that was previously obscured by the multi-scale nature of turbulence. (c) and (d) show the results of two consecutive time steps, which convey the smooth evolution of eddies via in-situ distance field construction.

5.3 Results and Discussion

We use four datasets with different data types in our experimental study. The first two combustion datasets are volumetric data generated from the turbulent combustion simulations performed at the Sandia National Laboratories. The Car dataset is a geometric model used in a computational fluid dynamics (CFD) simulation for car design. The Boeing 777 dataset is generated from a geometric CAD model constructed for the Boeing Company.

The distance field domain, Ω, of each dataset is evenly partitioned and distributed among the

Dataset	Data Type and Scale	Mode
Combustion	volume (1.3B grid points)	in-situ processing
Combustion	volume (1.6B grid points)	post-processing
Car	polygonal (3.4M triangles)	post-processing
Boeing 777	polygonal (350M triangles)	post-processing

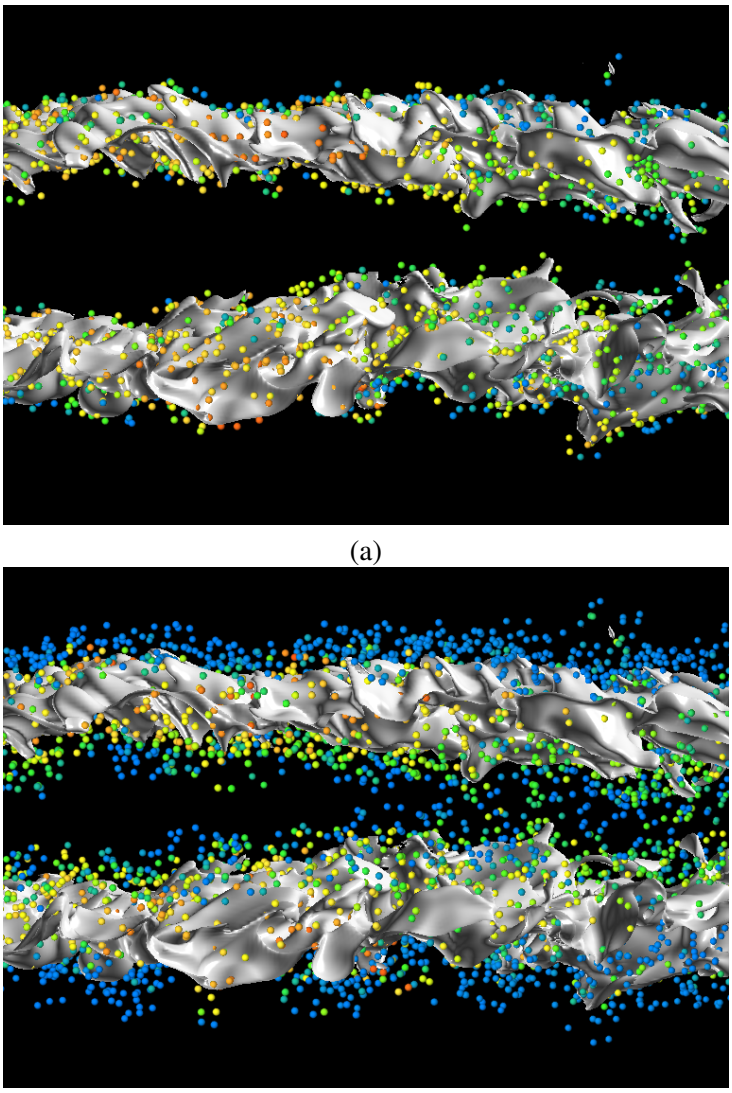
Table 5.1. The datasets used in our evaluation.

processors. The distance values are stored at the vertices of the parallel distance tree. We use the Euclidean distance that has more communication and computation requirements compared with the other distance functions. For the volumetric data, we first use the features consisting of a set of voxels. The distance values to the features such as an isosurface can be approximated as the Euclidean distances between the points and the voxels covered by the isosurface, which is commonly used in 2D and 3D image processing [18, 28]. We also explicitly extract the polygonal surfaces of the features using the parallel marching cubes algorithm on-the-fly, and compute the exact Euclidean distances between the points and the polygons, which meet the precision requirement for a detailed simulation but increase the computation cost for our evaluation study. Furthermore, to test the interpolation performance, we generate a distance field volume by regularly sampling the resulting parallel distance tree. For the combustion datasets, each distance field volume has the same resolution as the original volumetric data. For the polygonal dataset, we use a high resolution value for the distance field volume to sufficiently capture the fine structures. We test both in-situ processing and post-processing to verify the parallelism of our approach. Table 5.1 lists the datasets, the data types and scales, and the processing modes.

5.3.1 Application Results

The first combustion dataset has a spatial resolution of $2025 \times 1600 \times 400$ and each grid point contains 27 variables. Figure 5.6 shows an example of distance-based visual analytics for this dataset, where two variables, temperature (T) and hydroperoxyl radical (HO_2), are used. Figure 5.6 (a) shows an overview of the data. The main flame structure corresponds to the T surface at a particular isovalue. We use the geometry of the isosurface as the elements to construct the

distance field. After generating the distance field, a scientist can assign the opacity values to the voxels at the given distance threshold, and control the amount of information displayed around the surface to better observe variable relationships and fine structural information of small turbulent eddies, as shown in Figure 5.6 (b) and (c). This distance-based visualization can clearly separate previously hidden features that exist at the interior of larger structures. Figure 5.6 (c) and (d) show the results of two consecutive time steps, where we can perceive the smooth evolution of structures as we construct in-situ distance field construction at a high temporal frequency. This approach not only allows scientists to see previously hidden features but also enables more aggressive encoding of the data according to the distance field, leading to greater storage saving [81].



(a)

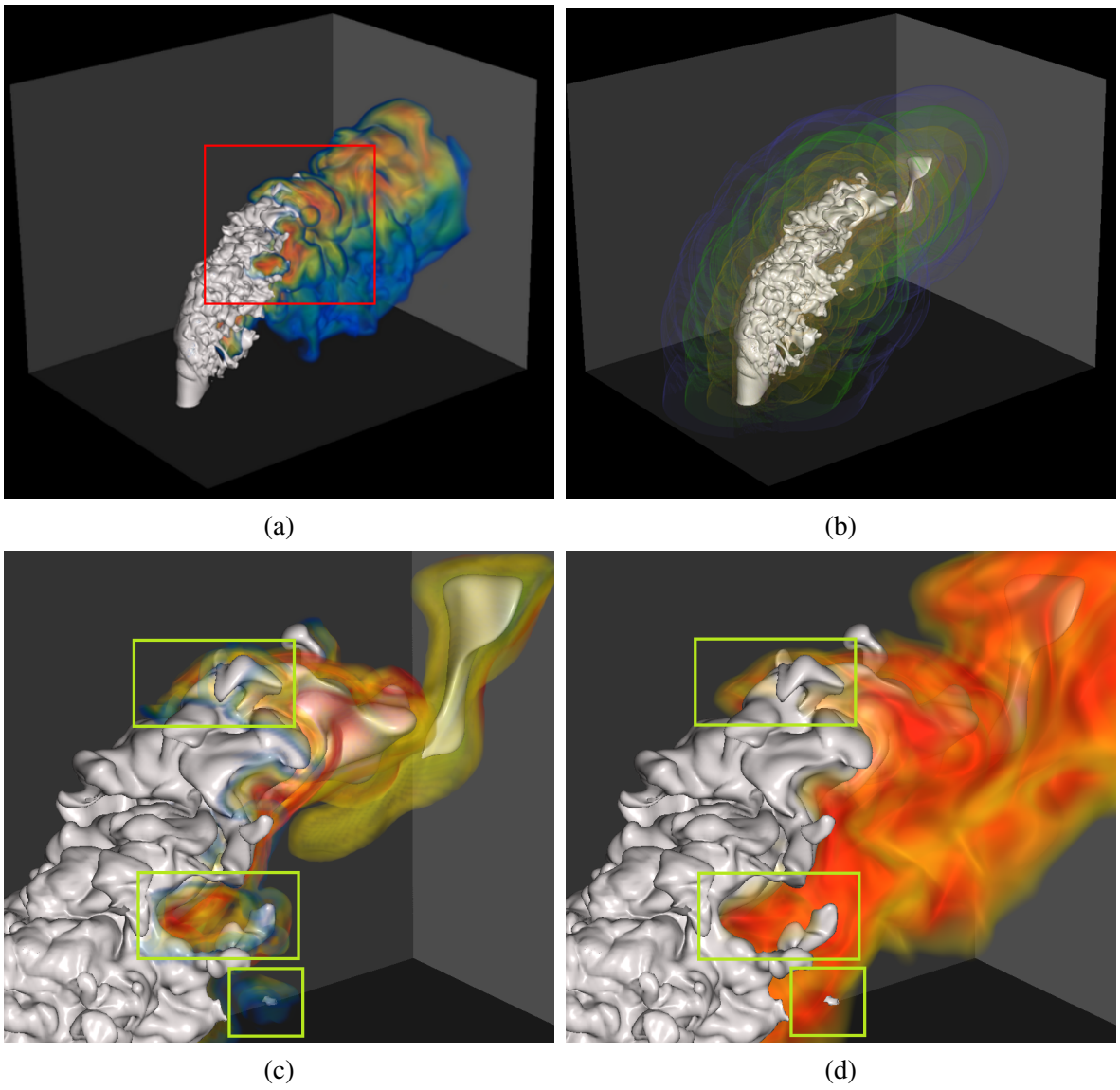
(b)

Figure 5.7. During a combustion simulation, the distance field is constructed based on an iso-surface of the T variable. (a) and (b) show that the scientists can control the initial placement and amount of particles with respect to the different distance thresholds. The particles are colored according to the values of HO_2.

Recently scientists have also instrumented their simulations with particles to capture and better understand the turbulent dynamics in combustion processes [84]. The initial placement of each particle is desired to be close to the flame surface. Through our integration method, we can construct the distance field with respect to the dynamic flame surface during simulations, and allow scientists to fine tune the placement and amount of particles with respect to the different distance thresholds, as shown in Figure 5.7.

The second combustion dataset has a spatial resolution of $1408 \times 1080 \times 1100$. In this case, we used the variables of HO_2 and stoichiometric mixture fraction ($mixfrac$). The main flame structure corresponds to the $mixfrac$ surface for which the isovalue is 0.2. Figure 5.8 shows the overview of the original simulation data, the resulting distance field, and the distance-based visualization. By interactively changing the distance threshold, we can reveal the distribution of HO_2 around the flame surface clearly. These details are occluded by the larger exterior structures (Figure 5.8 (a)) and cannot be easily visualized using traditional transfer functions, as shown in the comparison between Figure 5.8 (c) and (d). Our visualization provides scientists a clear exploration of the relationship between the combustion variables and the flame surface for detailed chemical mechanism.

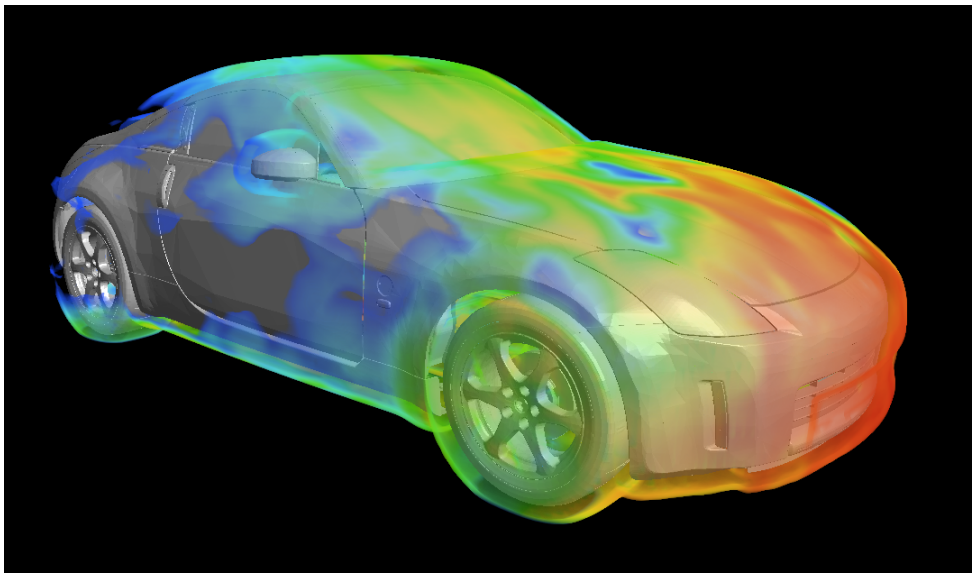
Apart from scientific simulation applications, our method can also naturally support the traditional geometric models used in engineering applications. The third dataset is a geometric model used in a numerical study on air flow effects on a passenger car. We build a parallel distance tree and generate a distance field with a spatial resolution of $512 \times 512 \times 512$. The researchers have used the CFD approach to obtain the aerodynamic data and complex flow structure around the car. Although the researchers are equipped with advanced visualization and analytic tools to capture important flow features, distance fields provide the researchers another dimension for investigation. For example, the researcher are interested in the effects of different material types applied on the front hood of the car to effectively control the distribution of air-temperature. As shown in Figure 5.9, we can clearly see the distribution of high temperature close to the car front (engine), as well as the attenuation of temperature with respect to the different distances from the car. This distance-based visualization can facilitate the researchers' investigation on material types with various air flow conditions.



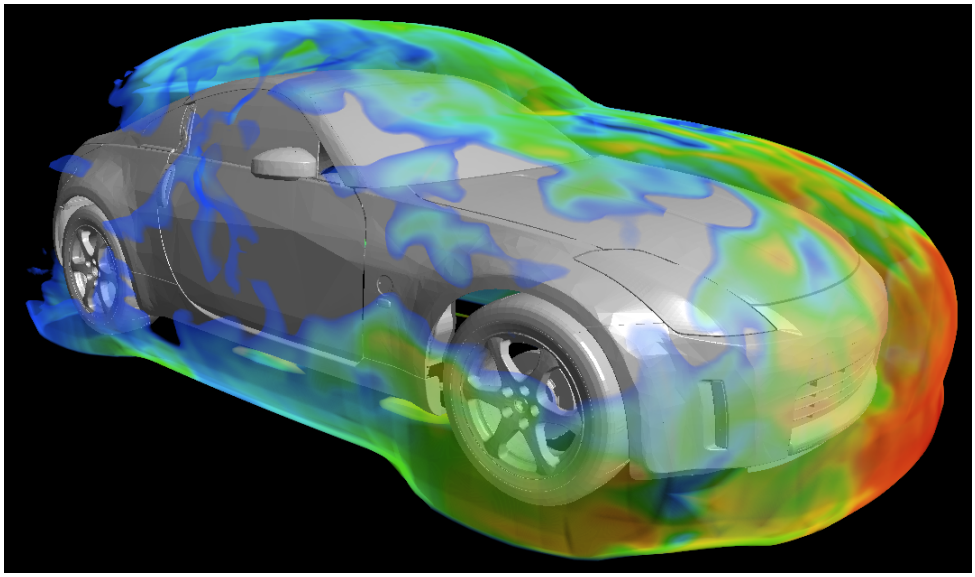
(a) (b)

(c) (d)

Figure 5.8. (a) shows the simultaneous rendering of two variables of the second combustion simulation dataset. The *mixfrac* isosurface is white. The HO_2 variable is volume-rendered, where higher values are indicated by orange or red, and lower values by green or blue. (b) shows the volume rendering of the distance field, where the white *mixfrac* surface is surrounded by a set of isosurfaces corresponding to the different distance values. (c) shows a distance-based rendering of the highlighted region in (a). With distance control, we can clearly see the distribution of HO_2 around the surface of the flame, which was occluded by the larger exterior structures in (a). This result cannot be easily obtained using traditional value-based transfer functions. (d) shows a rendering where we make the region of lower HO_2 (the green and blue region in (a)) translucent to make the interior *mixfrac* surface visible. However, the result cannot clearly reveal the relationship between higher HO_2 and the surface. It has lost the distribution information of lower HO_2. The highlighted regions in (c) and (d) provide the examples of comparison.



(a)



(b)

Figure 5.9. Volume rendering of the temperature field at different distances from the car (in particular, the car engine). Note that the apparent holes on the temperature field are due to the transfer function.

We also compute the exact Euclidean distance between points and polygons for the Boeing 777 model, where in this case, a 64-bit integer is used for the locational key. After building the parallel distance tree, we generate a volume of distance field with a resolution of $2048 \times 1942 \times 624$.

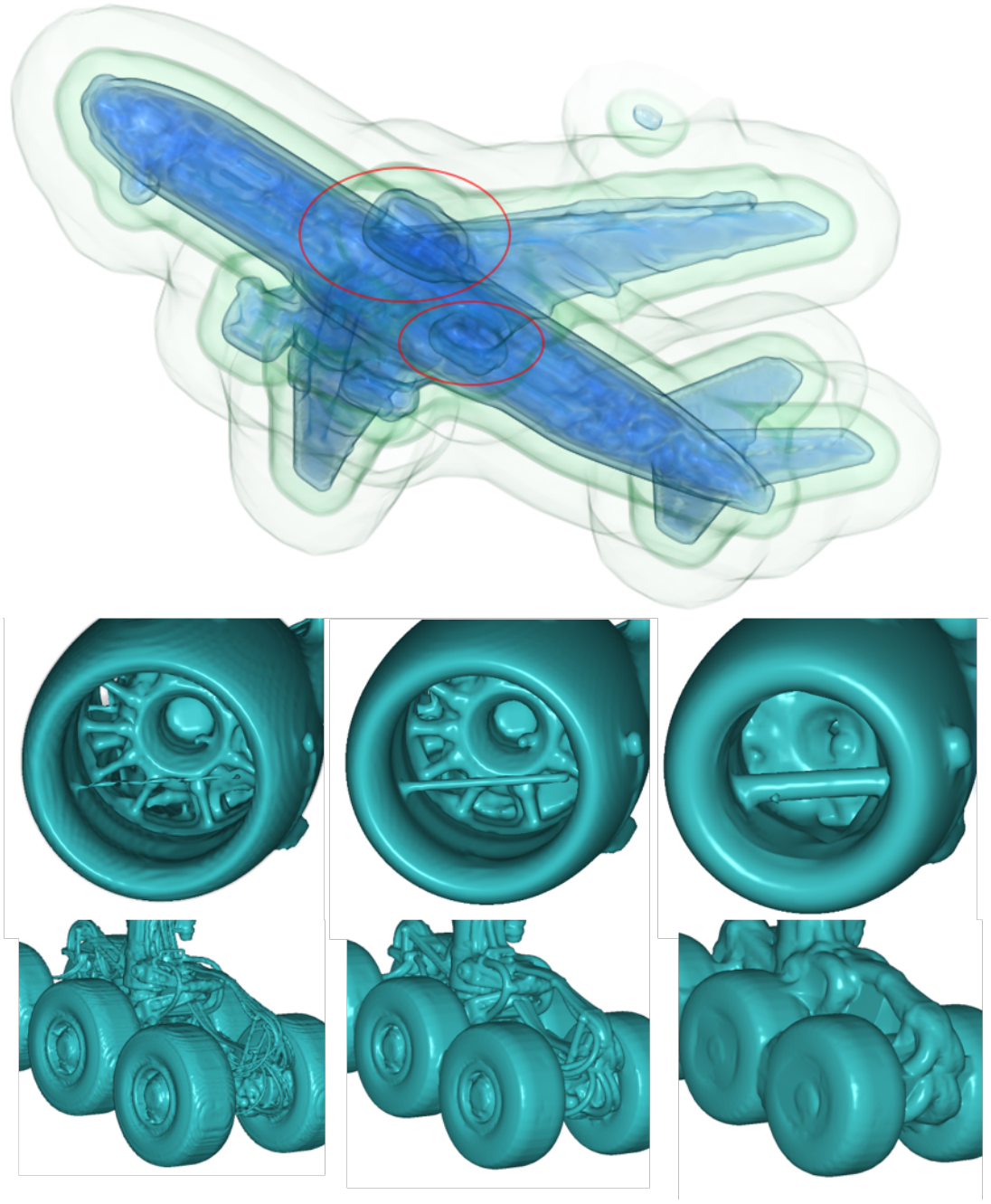


Figure 5.10. Isosurface rendering of the distance field generated from the Boeing 777 model. The top image shows the overviews of the distance field. The images in the second and third rows provide the close-up views of two highlighted components of the model, where we use isosurface rendering to depict the distance field, and the isovalues increase from left to right to mimic the effect of dilation. Our method can capture the fine structural details, for example, a long and very thin wire across the engine.

The constructed distance field features smooth contours and preserves fine details of individual components. We can apply different operations using the distance field, for example, the morphological operation of dilation shown in Figure 5.10. Note that the left images in the second and third rows of Figure 5.10 are generated using the distance field rather than the original model. Because the distance tree is refined adaptively around the vicinity of objects, our method can effectively capture fine structural details in such a large geometric model, which can be potentially useful in distance-related applications, such as collision detection in maintenance and assembly task simulations [27].

5.3.2 Performance Evaluation

We tested our parallel distance field method on two supercomputers. The first one is *Hopper*, a Cray XE6 supercomputer at the Lawrence Berkeley National Laboratory. The system contains 6384 nodes interconnected by the Cray Gemini Network. Each node has two 2.1 GHz twelve-core CPU with 32 GB of RAM. The second one is *Intrepid*, an IBM Blue Gene/P supercomputer at the Argonne National Laboratory. The system contains 40960 nodes interconnected by the InfiniBand Ethernet. Each node has one 850 MHz quad-core CPU with 2 GB of RAM. For clarity, we only present the performance results using the two combustion datasets and the Boeing 777 dataset in Table 5.1. Table 5.3 lists the major time components contributing to the overall cost.

In the first test, we conduct in-situ distance field construction using the first combustion dataset with the following three configurations:

- Config. 1: computing the Euclidean distances between points and isosuface polygons without OpenMP acceleration.
- Config. 2: computing the Euclidean distances between points and isosuface polygons with OpenMP acceleration. See section 5.2.10..
- Config. 3: computing the Euclidean distances between points and isosuface voxels without OpenMP acceleration.

PEs	4320	8640	17280	34560	69120
T_sim	123.64	62.72	27.21	14.62	24.39
Config.1: Euclidean distances between points and polygons (w/o OpenMP)					
T_dist_i	60.62(100%)	19.73(153.5%)	10.30(147.1%)	7.29(103.9%)	
T_dist_i'	60.46(100%)	19.39(156.0%)	9.28(163.0%)	5.90(128.1%)	
T_g_tree	0.16	0.36	1.03	1.39	
T_l_tree	1.76	0.96	0.73	0.38	
T_l_dist	11.27	4.04	1.93	1.13	
T_c_dist	47.42	14.37	6.61	4.38	
T_exg	0.01	0.01	0.01	0.01	
T_dist_u	58.83(100%)	18.47(159.2%)	8.59(171.3%)	5.54(132.7%)	
T_u_tree	0.13	0.06	0.04	0.02	
T_u_exg	0.01	0.01	0.01	0.01	
T_u_dist	58.69	18.40	8.54	5.51	
Config.2: Euclidean distances between points and polygons (with OpenMP)					
T_dist_i	20.12(100%)	7.25(138.7%)	4.41(114.2%)	3.53(71.3%)	
T_dist_i'	19.96(100%)	6.89(144.8%)	3.38(147.7%)	2.14(116.8%)	
T_g_tree	0.16	0.36	1.03	1.39	
T_l_tree	1.76	0.96	0.73	0.38	
T_l_dist	4.16	1.33	0.56	0.37	
T_c_dist	14.03	4.59	2.08	1.37	
T_exg	0.01	0.01	0.01	0.01	
T_dist_u	18.33(100%)	5.99(153.0%)	2.69(170.4%)	1.78(128.9%)	
T_u_tree	0.13	0.06	0.04	0.02	
T_u_exg	0.01	0.01	0.01	0.01	
T_u_dist	18.19	5.91	2.64	1.74	
Config.3: Euclidean distances between points and voxels (w/o OpenMP)					
T_dist_i	0.67(100%)	0.79(42.5%)	0.69(24.2%)	1.30(6.4%)	1.45(2.9%)
T_dist_i'	0.59(100%)	0.43(68.5%)	0.35(42.7%)	0.11(68.6%)	0.09(43.6%)
T_g_tree	0.08	0.35	0.34	1.19	1.37
T_l_tree	0.12	0.11	0.08	0.03	0.03
T_l_dist	0.14	0.08	0.07	0.02	0.01
T_c_dist	0.33	0.24	0.19	0.05	0.03
T_exg	0.01	0.01	0.01	0.01	0.01
T_dist_u	0.076(100%)	0.030(126.7%)	0.031(61.7%)	0.022(43.8%)	0.018(26.4%)
T_u_tree	0.065	0.020	0.018	0.010	0.008
T_u_exg	0.006	0.007	0.011	0.011	0.009
T_u_dist	0.005	0.003	0.002	0.001	0.001

Table 5.2. Timing breakdown for in-situ distance field construction with the combustion simulation on Hopper. The time is measured in seconds per time step. The percentage numbers represent the parallel efficiency where the times measured with 4320 CPU cores are used as the references.

We use the T isosurface in Figure 5.6 to construct the distance fields. Table 5.2 shows the detailed performance results on Hopper [3]. In Configuration 1 (the top green section of Table 5.2), we achieve the ideal speedup for the distance tree initialization (T_dist_i) with a parallel efficiency of 103.9% from 4320 to 34560 CPU cores. However, we also note that the parallel efficiency is about 67.6% from 8640 to 34560 CPU cores. This is because the initial global tree construction (T_g_tree) is not scalable in our current design, which becomes the performance bottleneck with a large number of processors. However, we note that this is one-time initialization and the cost can be amortized for a large number of time steps. Without considering T_g_tree, for T_dist_i', the parallel efficiency is 82.2% from 8640 to 34560 CPU cores, and 128.1% from 4320 to 34560 CPU cores. On the other hand, as we pointed out in Section 5.2.8, the cost to update the tree can be significantly lower than the initial tree construction cost. We can clearly see that T_u_tree is much smaller than the sum of T_g_tree and T_l_tree. Furthermore, we compute the Euclidean distances based on polygons in this configuration. As discussed in Section 5.2.8, we do not reuse the results from the previous time step for a precision purpose, and thus there are no savings for updating the distance field (T_u_dist). The overall parallel efficiency of tree and distance field updating (T_dist_u) is 132.7% from 4320 to 34560 CPU cores.

In Configuration 2 (the middle orange section of Table 5.2), we use OpenMP with four threads per MPI process to accelerate distance computation. The average speedups of T_l_dist and T_c_dist from OpenMP are about 305% and 322%, respectively, compared with the times in Configuration 1. Averagely, T_dist_i is 17% of T_sim, which is reduced from 42% in Configuration 1. T_dist_u is around 11% of T_sim, showing that we can compute the exact Euclidean distances at a low cost during a simulation run.

In Configuration 3 (the bottom pink section of Table 5.2), we compute the Euclidean distances based on isosurface voxels. The corresponding computation cost is much lower than the one based on polygons. The sum of T_l_dist and T_c_dist is averagely 1% of its counterpart in Configuration 1. Because of the lower computation cost, the overhead of T_g_tree clearly dominates T_dist_i with a large number of processors. However, in this configuration we can

[3]Not all data points of the time components have been obtained in our tests due to the limit of our supercomputer time allocation.

Time	Operation
T_sim	Simulation
T_g_tree	Build initial global distance tree
T_l_tree	Build full-grown local distance tree
T_l_dist	Compute local vertex distances
T_c_dist	Compute remote vertex distances
T_exg	Exchange vertices and vertex distances
T_vol	Construct sampled distance volume
T_u_tree	Update local distance tree
T_u_exg	Exchange vertices and vertex distances for updating
T_u_dist	Update local and remote vertex distances
T_dist_i	Initialize: $T\_g\_tree + T\_l\_tree + T\_l\_dist + T\_c\_dist + T\_exg + T\_vol$
T_dist_i'	Initialize: $T\_l\_tree + T\_l\_dist + T\_c\_dist + T\_exg + T\_vol$
T_dist_u	Update: $T\_u\_tree + T\_u\_exg + T\_u\_dist$

Table 5.3. Major execution time components and their corresponding operations.

easily track the updated voxels overtime, and only update the portion of distance fields related to these voxels. Therefore, we can significantly lower both T_u_tree and T_u_dist. As shown in Table 5.2, we can see that T_dist_u is only about 1.2% of T_dist_i with 69120 CPU cores, and is negligible to the simulation time T_sim.

Figure 5.11 shows the detailed performance results using the second combustion data to construct a distance field on Intrepid. Figure 6.7 (a) and (b) show the overview of the original data and the resulting distance field. Our method achieves 86.9% average parallel efficiency from 1024 to 32768 CPU cores, without considering T_g_tree. We can see that the most expensive operations, the local tree construction and the local vertex distances computing, are still close to the ideal speedup. Compared with the first test, an interesting observation is that the computation times are significantly higher in the second test. An explanation could be that Intrepid has a slower CPU speed compared with Hopper. Nevertheless, our method achieves desired performance on both architectures.

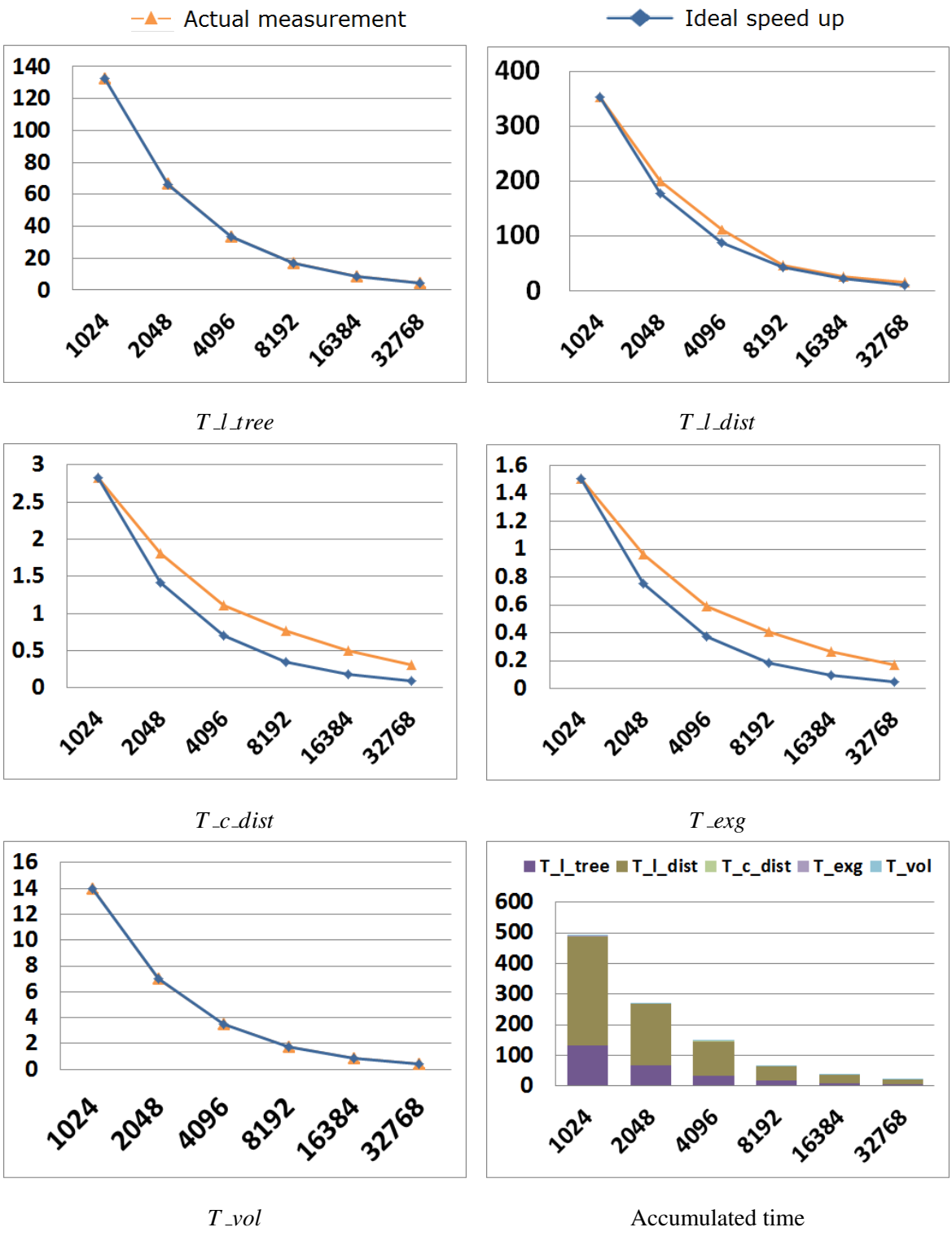


Figure 5.11. Scalability study using the second combustion dataset for post-processing on Intrepid. In each plot, the horizontal axis represents the number of processors, and the vertical axis represents the running time in seconds.

PEs	1024	2048	4096	8192
T_dist_i	4083.79(100%)	2327.11(87.74%)	1259.33(81.07%)	678.26(75.26%)
T_dist_i'	4083.74(100%)	2327.02(87.75%)	1259.06(81.09%)	677.78(75.31%)
T_g_tree	0.05	0.09	0.27	0.48
T_l_tree	426.93	293.62	168.08	101.84
T_l_dist	2812.12	1548.52	803.37	405.44
T_c_dist	836.55	477.48	281.10	165.72
T_exg	8.14	7.39	6.51	4.78

Table 5.4. Timing breakdown for computing the distance field of the Boeing 777 model on Hopper. The time is measured in seconds. The percentage numbers represent the parallel efficiency where the times measured with 1024 CPU cores are used as the references.

Table 5.4 shows the detailed performance of distance field construction around the Boeing 777 model on Hopper. The overall parallel efficiency is 75.26% from 1024 to 8192 CPU cores. To the best of our knowledge, we present for the first time the scalable distance field construction of a geometric data at this large scale.

5.3.3 Discussion

The performance study shows several advantages of our method. First, we can clearly see that the overall parallel distance field construction scales very well with the increasing number of processors. The effectiveness of our distance tree is clearly illustrated in the two rather challenging combustion cases where the set Γ of elements is only distributed among a small set of the processors. Moreover, although T_g_tree increases with the number of processors, it is only a one-time cost. The subsequent cost for updating tree is much smaller by leveraging the temporal coherence of simulation data. Such a low cost makes it feasible to compute a distance field during the simulation run, and thus significantly reduces time to solution by avoiding expensive data movement.

Second, our method has a low communication cost even with a large number of processors. This is first because our communication schedule is constructed by leveraging the spatial coherence of simulation data, and each processor only needs to exchange data with a small

set of neighboring[4]processors that are identified efficiently with our parallel distance tree (Section 5.2.5). In addition, we only send the vertex key that is a 4- or 8-byte integer, rather than three floating-point coordinates, while the returned result is a 4-byte floating point distance value. Thus, our method only incurs a small communication footprint relative to the size of the entire distance field volume. For example, in the second test, the exchanged data is under 0.01% of the total data on 32768 CPU cores.

Third, the experimental results show that our method does not depend on any particular architecture. The linear data representation can facilitate the acceleration of distance computation using different techniques, such as GPU and OpenMP. Furthermore, our examples highlight the flexibility of our method with respect to different data types and applications in the real-world. Given our novel design of the parallel distance tree, we obtain clear improvement over the previous state of the art, and believe that the design principle is applicable to a wider class of scientific and engineering applications.

However, a native octree may not provide effective I/O once we store a large tree on persistant storage. This is because an octree is characterized as a narrow/deep tree such that visiting a leaf node may require a long traversal path from the root and incur intensive I/O operations. In the future, we will study approaches to store an out-of-core octree for large distance fields. In addition, we have not implemented a direct visualization of the parallel distance tree yet. Thus, although exact distance values at any point can be computed from a tree, we have to interpolate the tree to generate a dense distance field volume for visualization. We plan to leverage our parallel octree-based volume rendering method [99] to enhance visualization.

5.4 Summary

A highly scalable parallel distance field construction algorithm is critical for data analysis and visualization of large-scale applications. The presented method employs inexpensive load balancing at the global tree construction stage and efficiently parcels out workloads among the processors to minimize the communication traffic. It features a highly scalable scheme of data

[4]The adjacency of two processors defined here is not determined by the topology of the communication network, instead, by the adjacency of the assigned element blocks of Γ among the processors in the 3D Cartesian space. For example, in Figure 5.3(a), processor PE_3 is the direct neighbor of processor PE_4, because their element blocks are adjacent in the Cartesian space.

partition and representation compatible to large systems. It also has been directly integrated with simulation codes to minimize the data transformation overhead. Moreover, it supports interactive exploration of fine details in large data. It is believed that the algorithm sets a record with regard to highly scalable performance of real-world large data on distributed architectures for distance field construction.

Researchers have exploited massive GPUs to accelerate simulations [34]. However, the disparity between I/O speed and compute speed is further aggravated. The communication cost and load balancing remain the fundamental challenges for scalability. Our ultimate goal is a fast, scalable distance tree that can handle large scale data with different underlying representations taking advantage of hundreds of thousands of CPU cores or GPU accelerators. Our approach provides a foundation for big data management and is key to bridge the gap between simulations and data analytics in the exascale computing era.

Chapter 6

Fast Uncertainty-driven Large-scale Volume Feature Extraction on Desktop PCs

Feature extraction in volumetric datasets is an extremely popular tool with many different uses. Firstly, it allows researchers to isolate regions based on a desired set of qualities. The location, shape, and evolution of these regions allows users to gain scientific insight into the system they are studying. It is also a powerful data reduction tool. With the growing size of datasets, the ability to extract and operate on subsets from data has become a necessity. Lastly, feature extraction allows a system to distinguish objects from a "background" and even from one another. Such a distinction becomes important for techniques such as transfer function generation and even has numerous applications in computer vision.

The increasing size of large-scale datasets often forces researchers to rely on the increased computational resources of a distributed computing environment. As a result, many of the current feature extraction techniques which operate on large-scale data are designed for a distributed setting. However, these environments do have a number of drawbacks in terms of interactivity and convenience. For example, there is the need to compete with other users over computational resources resulting in long queues and wait times before executing a job. In addition, the computing environment and available toolkits/libraries are often limited. In visualization, interactivity in data exploration becomes key. Working in a remote environment limits any feedback and interactivity according to available network bandwidth. As a result, many scientists tend to prefer analyzing their datasets locally on a desktop PC where they have more

control over their system. The ability to overcome the memory and computational limitations of desktop PCs can make them more applicable towards large-scale applications.

Traditional feature extraction and data clustering techniques can generally be classified into two main approaches. The first approach evaluates the volume as a whole and groups discrete components based on a similar set of properties. This can provide a meaningful approximation of the underlying segments, but may lose track of precise data discontinuities. Using a voxel-based K-means clustering would be an example of such an approach. This normally requires the number of clusters to be known a priori and has a complexity that depends on the number of initial clusters, the data domain size, and the number of iterations it takes to converge. The other approach works on a local level and identifies coherent or connected components. It can accurately detect discontinuities but is sensitive to noise perturbation and may segment a region into too many components. An example of this approach would be seeded region growing and has a complexity based heavily on the size and resolution of the extracted feature.

Each method alone has a number of advantages and disadvantages which become increasingly evident in large-scale datasets. In this work, we present a new hybrid feature extraction technique which combines a GPU-accelerated version of the SLIC [6] clustering technique with the multi-resolution advantages of "supervoxels" in order to handle large-scale datasets on standard desktop PCs. Rough clustering results are then further refined using an uncertainty-driven approach to enhance extraction results into a desired level of detail. A hybrid approach like the one presented here allows one to retain many of the advantages of various feature tracking techniques. Furthermore, our GPU-accelerated method allows users to process and explore their large-scale datasets in real time on a local desktop PC without the drawbacks of a remote distributed system.

This chapter presents a new hybrid extraction technique and make the following contributions:

- It is a new GPU-accelerated hybrid feature extraction technique for large-scale data.
- It is paired with the multi-resolution advantages of supervoxels in order to handle large datasets on a desktop PC.

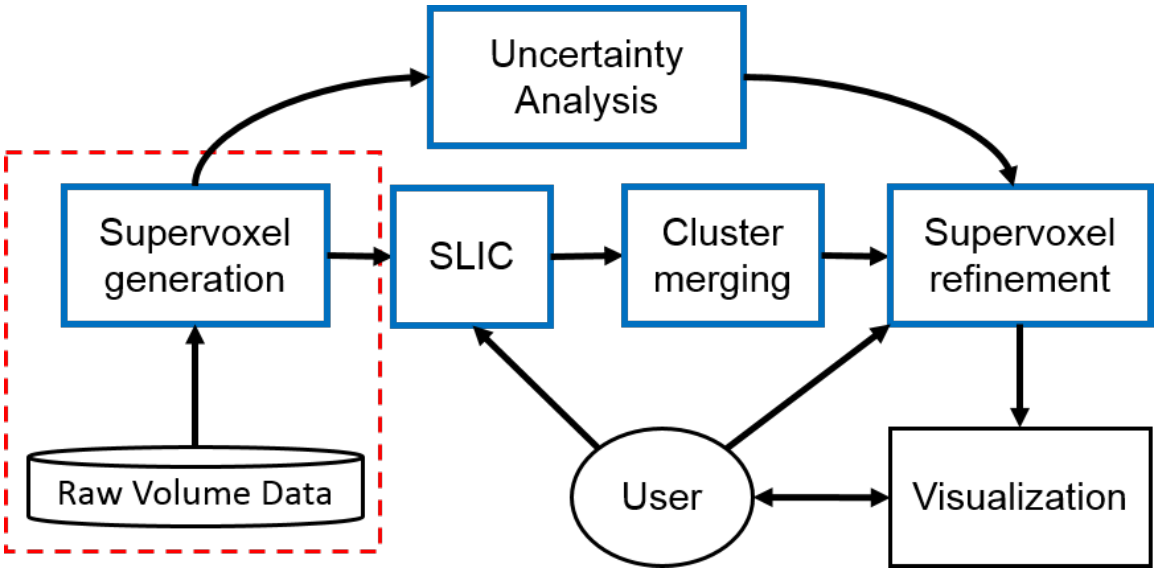


Figure 6.1. An overview of our system pipeline, blue components represent the technique introduced in this chapter. The section outlined in red can be done either as a preprocessing step or during a simulation run (in situ).

- An uncertainty-driven feature refinement method is provided to enhance extraction results.

We demonstrate the effectiveness of our algorithm using several real world large-scale datasets, including large combustion, ocean, and flow simulation data. Using each of these datasets we can achieve real time extraction and exploration capabilities on a local desktop computer.

6.1 Methods

Our approach is motivated by the idea of giving users fast response time when exploring and extracting features of large volume datasets on a local desktop machine. As a result, many of the steps involved in our algorithm are designed to provide a (potentially) user-controlled balance between the performance and accuracy of each step. Different datasets have a very large range in underlying data patterns and in many cases must be treated differently to achieve a quick accurate extraction result.

6.1.1 Overview

An overview of our approach can be seen in Figure 6.1. Our approach starts with a preprocessing step that partitions the volumetric data into a set of equally sized blocks called supervoxels. Each supervoxel can represent a 2^3, 3^3, etc. space of voxels and is similar to a down-sampling of the full volumetric dataset. This ensures interactive response times when generating rough initial extraction results. While this is often a fast preprocessing step, it can alternatively be implemented in situ and saved along with the rest of the simulation data. This is to alleviate the burden of disk I/O when working with massive datasets. During preprocessing step, supervoxels are generated using the GPU where additional statistical metrics are computed, such as the average intensity value and standard deviation of internal raw voxel values. These metrics are later used to determine which supervoxels need to be refined to generate a more accurate extraction result.

Next, we apply a GPU-accelerated version of the Simple Linear Iterative Clustering (SLIC) algorithm [6] to partition similar supervoxel blocks into a set of clusters. Note that the term supervoxel in [6] is used differently than in this chapter. Since the number of clusters is predetermined by this algorithm, we include a merge step which combines similar clusters in a hierarchical fashion. Lastly we use uncertainty metrics based on the standard deviation of values within a supervoxel to guide users in selecting which parts of the extraction result must be refined. This refinement step will fetch the high resolution raw voxel data only for portions of the extraction result that display a high uncertainty in accuracy. The result contains data values at both the lower supervoxel resolution and the higher raw data resolution depending on the underlying nature of the dataset.

6.1.2 Supervoxel Generation

The supervoxel generation step is similar to many approaches used to down-sample large scale data. In our implementation, each supervoxel represents an equally sized block of the original volume data. The resolution of the supervoxel depends on the number of raw data voxels contained inside. For example, we say that a supervoxel containing 3x3x3 = 27 raw data voxels has a resolution of 3. We can choose the resolution of supervoxels to use based on the size of the original raw dataset, with larger resolutions required as the data becomes larger.

This conversion step is implemented using GPU-acceleration. The raw volume data is loaded into the GPU as a 3D texture, where it resides in global memory. We then define one GPU thread for every supervoxel which accesses the raw data values of the volume texture. The average intensity and standard deviation are computed for each supervoxel in parallel. This intensity value is used as input into clustering in the next step while the standard deviation is used later during the uncertainty-based refinement step. Since this step can be executed very quickly, we perform it at run time and give users control over the desired supervoxel resolution. Note that if the volume does not fit entirely in GPU memory we process it in chunks. Performance results for the supervoxel generation step are discussed in detail in the results section of this chapter.

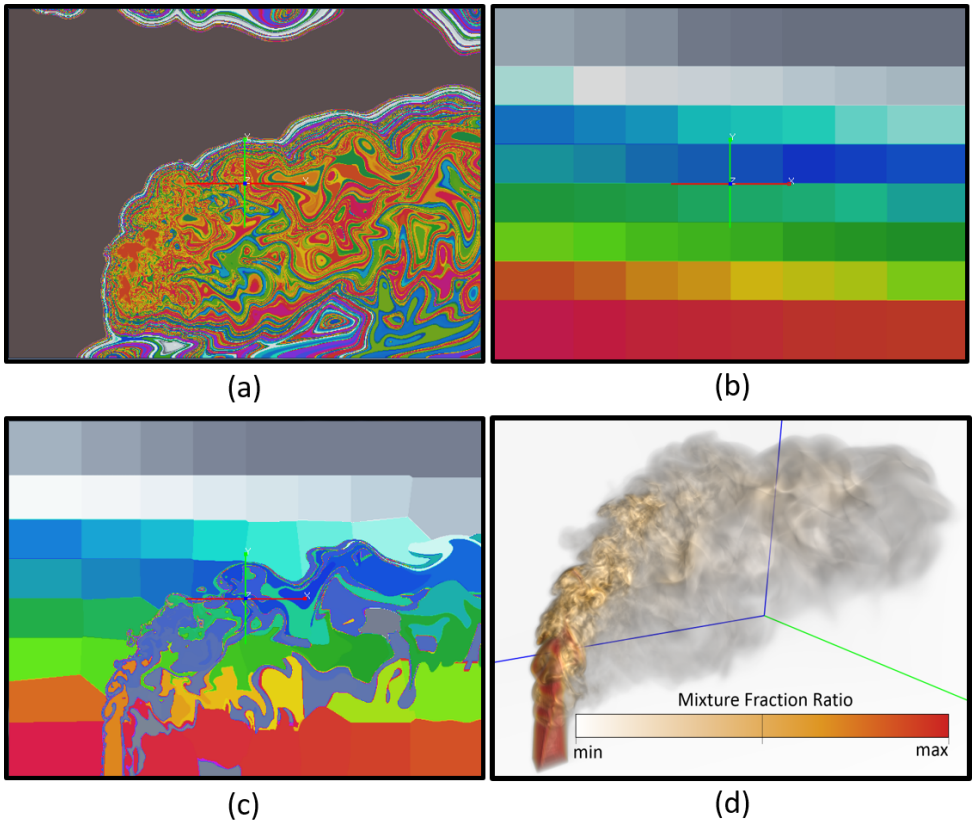


Figure 6.2. SLIC clustering results ((a), (b), and (c)) on a combustion simulation dataset (d) shown on a 2D slice. The color in (a), (b), and (c) indicates different clusters. This particular example (c) shows a balance between the proximity and intensity weights used. (a) and (b) images show the two extreme cases where the proximity and intensity weights are set to zero respectively.

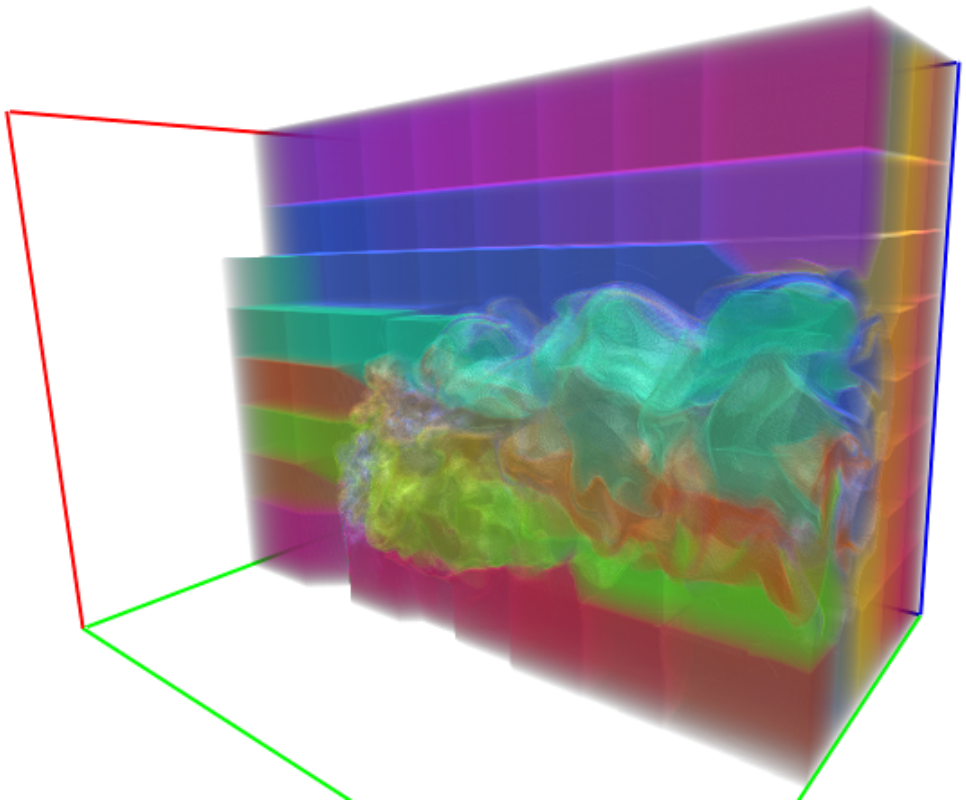


Figure 6.3. The result of the first iteration after our 3D SLIC implementation on a combustion simulation dataset. The color encodes different cluster IDs. In this case, 8x8x8 cluster centers in a grid layout are initialized. Even after one iteration, the salient structure and similar features are captured by the SLIC approach.

6.1.3 3D Supervoxel Clustering via SLIC

To efficiently identify features from the supervoxel volume, we transform a simple but efficient image clustering technique called Simple Linear Iterative Clustering (SLIC) [6] to segment the supervoxel volume into a number of different clusters. In our implementation, this algorithm is treated in a 3D sense and accelerated using the GPU to ensure fast clustering times. Since we are operating on the supervoxel volume rather than raw large-scale data, this clustering result can also be done on the fly in real time.

The SLIC approach is very similar to k-means clustering, but varies in two main aspects. First, instead of computing the distance from each cluster center to all voxels in the entire domain, the search space is limited to within a region proportional to the cluster size. Second, the distance metric used combines intensity and spatial proximity which provides control over

the size and compactness of the clusters. The initial cluster centers are selected on a regular grid in 3D volume with interval length of $S_i (i \in \{0,1,2\})$ for each dimension. The length S_i is determined by the number of initial clusters k_i and the extent of the volume N_i in each corresponding dimension; $S_i = N_i/k_i$. In each iteration, the algorithm assigns each voxel to the most similar cluster, updates the cluster centers, and repeats until a certain stopping criteria is met.

SLIC is computationally less expensive than the conventional k-means algorithm because its distance metric measuring similarity is computed within a $2S_0 \times 2S_1 \times 2S_2$ local region centered at each cluster center. We define the distance metric as:

$$d_{k,i} = w_1 ||c_k - v_i||_2 + w_2 |I_k - I_i|$$

where I_k and I_i are the scalar values at the kth cluster center and voxel i respectively; c_k is the position vector of cluster center k, and v_i is that of voxel i. w_1 and w_2 are weights. Increasing w_1 would result in a clustering biased towards spatial proximity and as a result preserves more compactness. Increasing w_2 would cause the clustering to adhere more tightly to the boundaries between varying scalar values. Figure 6.2 demonstrates a series of different clustering results on a 2D slice of a volume data with different combinations of w_1 and w_2. Figure 6.3 shows an example of a clustering result in 3D after just one iteration.

By localizing the search within a $2S_0 \times 2S_1 \times 2S_2$ volume during clustering, the computational complexity of the SLIC algorithm is independent of k and scales with $O(N)$. In contrast, the classical k-means has $O(kN)$ complexity for each iteration. This distinction is important when dealing with large-scale 3D volume data rich in features where k must also be large. There are also other popular algorithms whose clustering results are similar to that of SLIC such as Turbopixels [35] and Quickshift [79]. However, SLIC is superior to many of these approaches in terms of segmentation quality and is faster with a smaller memory requirement. Lastly, we accelerate this step using the GPU. The supervoxel volume is loaded as a 3D texture into global memory. The embarrassingly parallel nature of this technique allows us to compute the distance metric between each cluster and supervoxel simultaneously via a separate GPU thread.

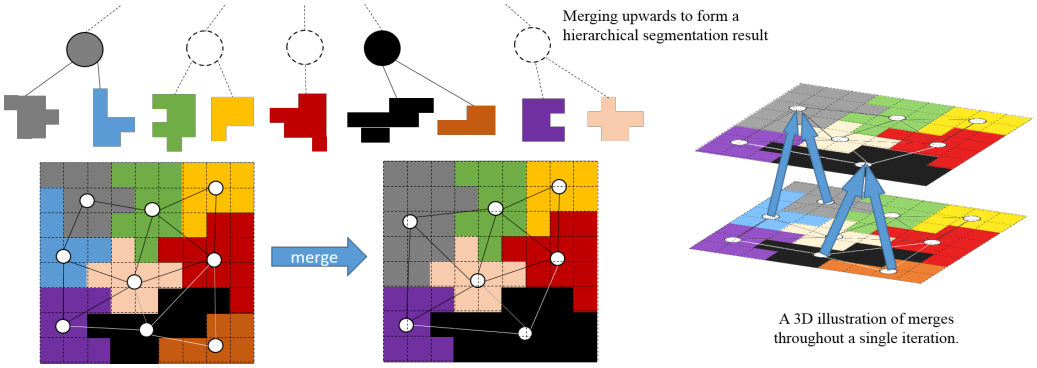


Figure 6.4. An example of merging sets of clusters to construct a hierarchical partitioning. The blue and gray clusters are merged as shown by solid filled circles. This continues until there is a desired level of variation within each cluster. For example, the green and gold clusters may merge in the next iteration as shown by the dotted circle.

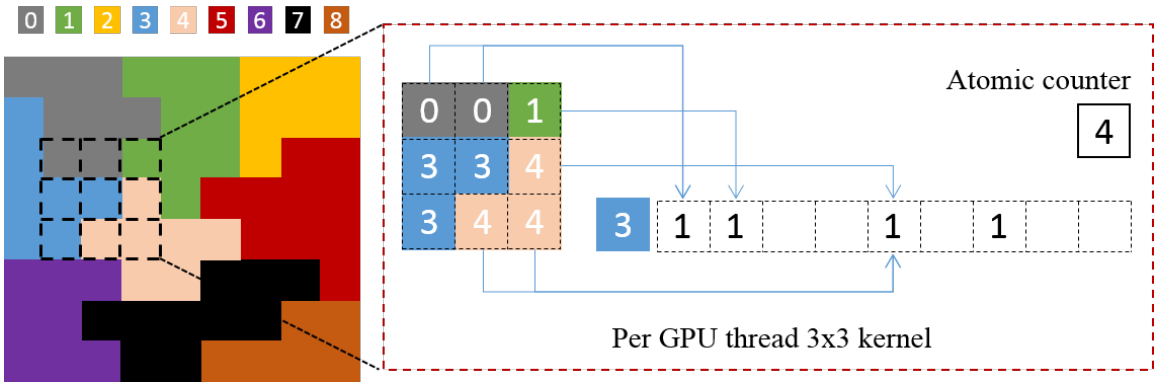


Figure 6.5. A 2D illustration of running template scheme to detect neighboring clusters for every cluster. Each thread is mapped to the center of a 3×3 template and iterates each pixel within the template. The neighboring clusters i are marked on the ith column of row j corresponding to the cluster j at the template center.

6.1.4 Hierarchical Merging

The aforementioned SLIC method often over-segments the supervoxel volume into many small clusters, and within each cluster the supervoxels share similar intensities and are within a close spatial proximity. After applying SLIC, small clusters of supervoxels are generated. We use these clusters to generate a hierarchical clustering result by merging the clusters of supervoxels. If one imagines a hierarchical clustering as tree-like structure, the output from the SLIC step would generate nodes found towards the bottom of the tree and merging clusters will build up this structure towards a root node.

One disadvantage of employing SLIC as a clustering step is that it produces many redundant

clusters in homogeneous regions such as the background. As seen in Figure 6.3, while capturing basic flame structure, SLIC also creates many similar cube-like clusters in the background region. Ideally, one cluster suffices to represent the background even though it has subtle variation in its scalar value. Therefore, it is necessary to combine similar clusters into one larger cluster. We would like to first consider those clusters that are both spatially adjacent and are close in terms of the user-specified distance metric. Therefore, a reduce operation is needed to coalesce groups of clusters identified by the SLIC step and reconstruct the connected structure.

Unfortunately, SLIC does not compute the topological connectivity of the output clusters (similar to the marching cubes algorithm that produces a 'soup' of disconnected triangle elements [38]). As the initial step of cluster merging, we work on building up the topological connectivity of clusters on the GPU. The key idea of our topology reconstruction step is to build an undirected graph G such that each vertex $v_i \in V$ represents cluster i. There exists an edge $e_{ij} \in E$ if and only if cluster i and j are spatially connected.

To efficiently build-up the adjacency matrix M where $M[i,j] = 1$ indicates an edge $e_{ij} \in E$ in graph G, we utilize a running template over each supervoxel in the cluster and check the neighboring cluster information. At each stop of the template, a $3 \times 3 \times 3$ supervoxel region is searched. Let i denote the cluster ID of the supervoxel at template center and j denote the cluster ID of one of 26 neighboring supervoxels in the template. $M[i,j]$ is tagged with 1 if $i \neq j$, otherwise $M[i,j] = 0$. The running template scheme is computationally intensive since each supervoxel will check its 26 neighborhoods. However, each supervoxel is independent from the others, and we can parallelize the process by assigning a GPU thread to a supervoxel. Each thread checks its neighboring supervoxels in parallel and tags the corresponding entry on the adjacency matrix. A 2D example case, is shown in Figure 6.5. A thread is assigned to work within a 3×3 template. The center of the template belongs to cluster three and it has neighboring clusters zero, one, and four shown in different colors. Hence column zero, one, and four of row three in the adjacency matrix are tagged with one. Note that parallel thread hazard might exist when multiple threads are setting the global entry $M[i,j]$ to one. But it does not matter which succeeds.

Once the graph is constructed we can merge similar clusters by combining the respective

nodes in the graph and removing their connecting edge. This step can be performed on the CPU since the number of clusters is much smaller than the number of individual voxels. An example of a merge can be seen in Figure 6.4.

6.1.5 Uncertainty-based Refinement

The aforementioned steps produce an extraction result on the supervoxel level. However, studying the full resolution data is often necessary in identifying subtle patterns. This final step focuses on refining the supervoxel based clustering in regions of high uncertainty. The uncertainty metric we use is based on the standard deviation of the scalar data values of the raw voxels contained within a supervoxel. This has already been computed during the supervoxel generation step. Instead of refining every supervoxel in our cluster of interest (a very computationally intensive task), users can choose a desired level of uncertainty criteria through which the final result must meet. Any supervoxels whose uncertainty is higher than the user-defined threshold are therefore selected for refinement.

This refinement step requires retrieving the original raw data values associated with the supervoxel from disk. As they are loaded into memory, each raw voxel value is compared against the clustering attributes used to generate the particular supervoxel cluster. If this distance criteria is met, then the raw voxel is kept, otherwise it is checked against any neighboring clusters and potentially added to one of them. Once a supervoxel has been refined, its uncertainty value drops to zero since we are now operating on the raw data values. This results in an extraction result with multi-resolution components and is represented as an additional level in our hierarchical extraction result.

6.2 Results and Discussion

We have conducted tests using three datasets from different application domains. The first is a dataset generated from the turbulent combustion simulations performed at Sandia National Laboratories [12]. The second is a tropical oceanic dataset simulated by the National Oceanic and Atmospheric Administration Geophysical Fluid Dynamics Laboratory using Community Climate Model (CCSM): Parallel Ocean Program 2 (POP2) [23]. The third dataset comes from

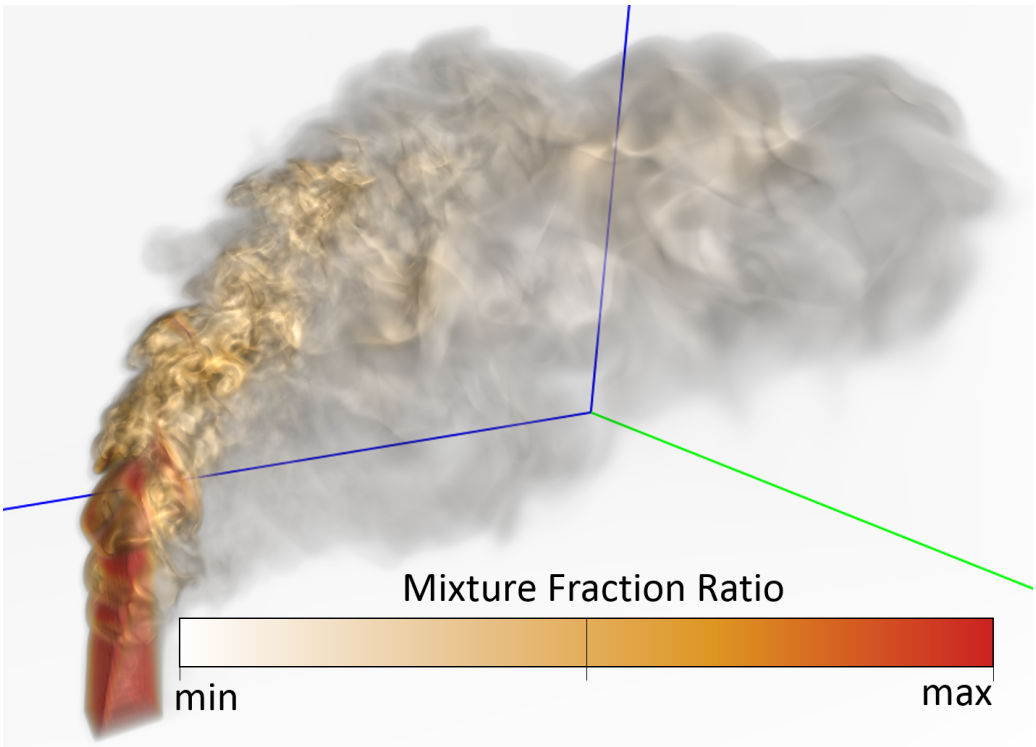


Figure 6.6. A 3D volume rendering of the combustion simulation data using a mixture ratio variable. Fuel is injected from the lower left part of the image into a region containing oxidizer. As it travels to the top right, it mixes and burns with the surrounding medium.

large-scale flow studies conducted by the Argonne National Laboratory [7].

6.2.1 Combustion Data

The phenomenon of combustion in 3D highly turbulent conditions and mixed-modes has many new physical and chemical properties. To better understand the complicated interactions, direct numerical simulation is required to record and delineate the key turbulence-chemistry interactions. Scientists at Sandia National Laboratories have developed S3D [12] to tackle this challenge at large scale using massive parallel supercomputers. The dataset in our study has a spatial resolution of $704 \times 540 \times 550$ and each grid point contains several variables. During the experiment, we consider the mixture fraction field since it is one of the two key combustion parameters associated with turbulent mixing and autoignition. It is a mixing measure of fuel and oxidizer. The highest value represents pure fuel and zero represents pure oxidizer. Figure 6.6 shows a 3D volume visualization of the mixture ratio.

Figure 6.7 shows examples of applying the clustering scheme to the combustion dataset. Af-

ter the merge step, we can remove the background and focus on exploring clusters that make up the flame itself. Since we are clustering based on the mixture fraction, such an extraction result is effective at highlighting the complex structures that form in such a turbulent environment.

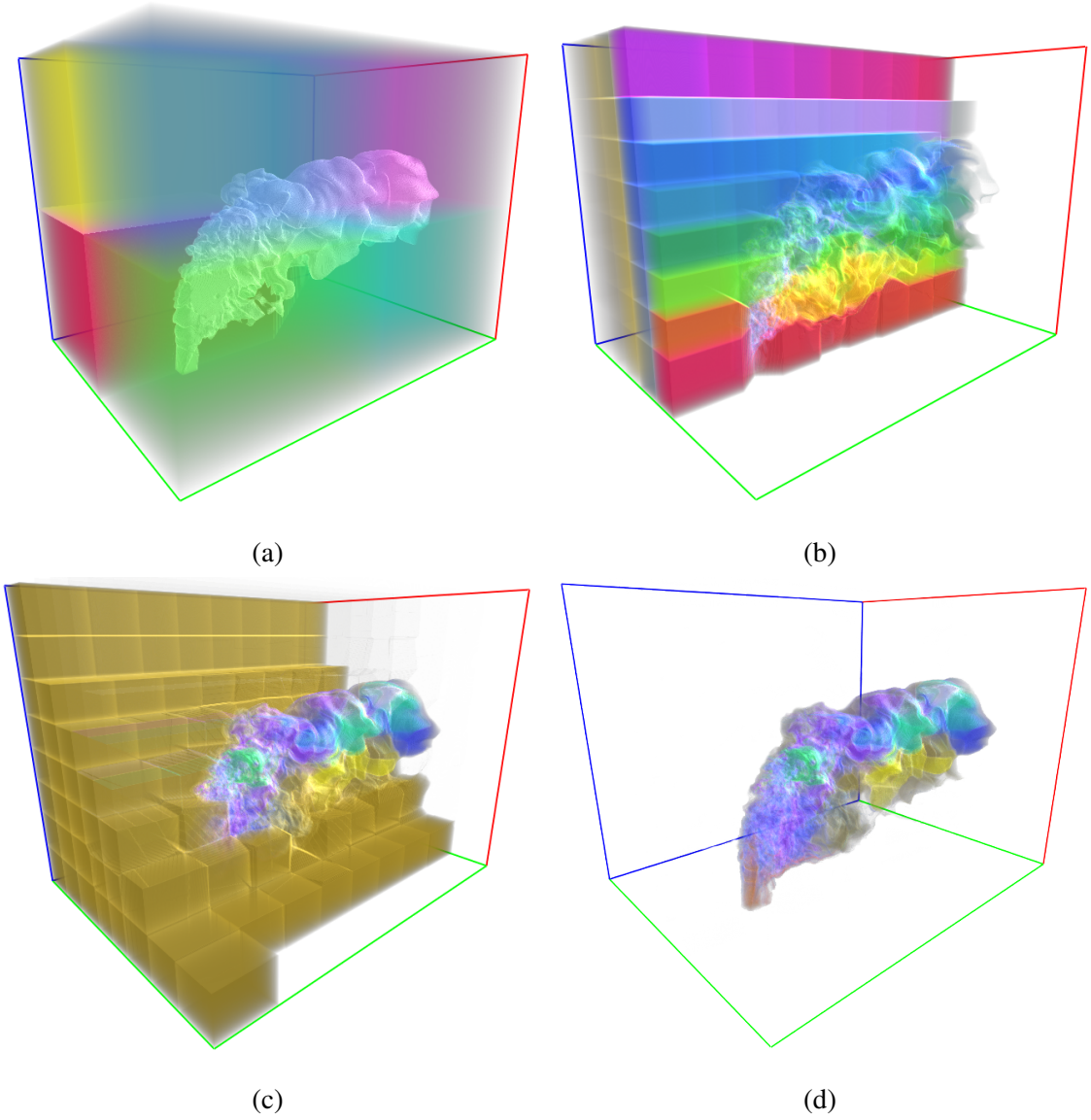


Figure 6.7. (a) The SLIC clustering results using $2 \times 2 \times 2$ initial cluster centers. (b) The clustering results after one iteration using $8 \times 8 \times 8$ cluster centers. (c) The result after applying merge. Many redundant cluster centers in the background are coalesced into one supercluster colored in dark yellow, while the clusters corresponding to the flame structure remain unchanged with a slight color variation due to the reassignement of cluster id's. (d) Removing the background cluster and emphasizing the entire extracted flame features using intensity value and Euclidean distance as distance metric. The supervoxels in each extracted cluster share both similar intensity and spatial proximity.

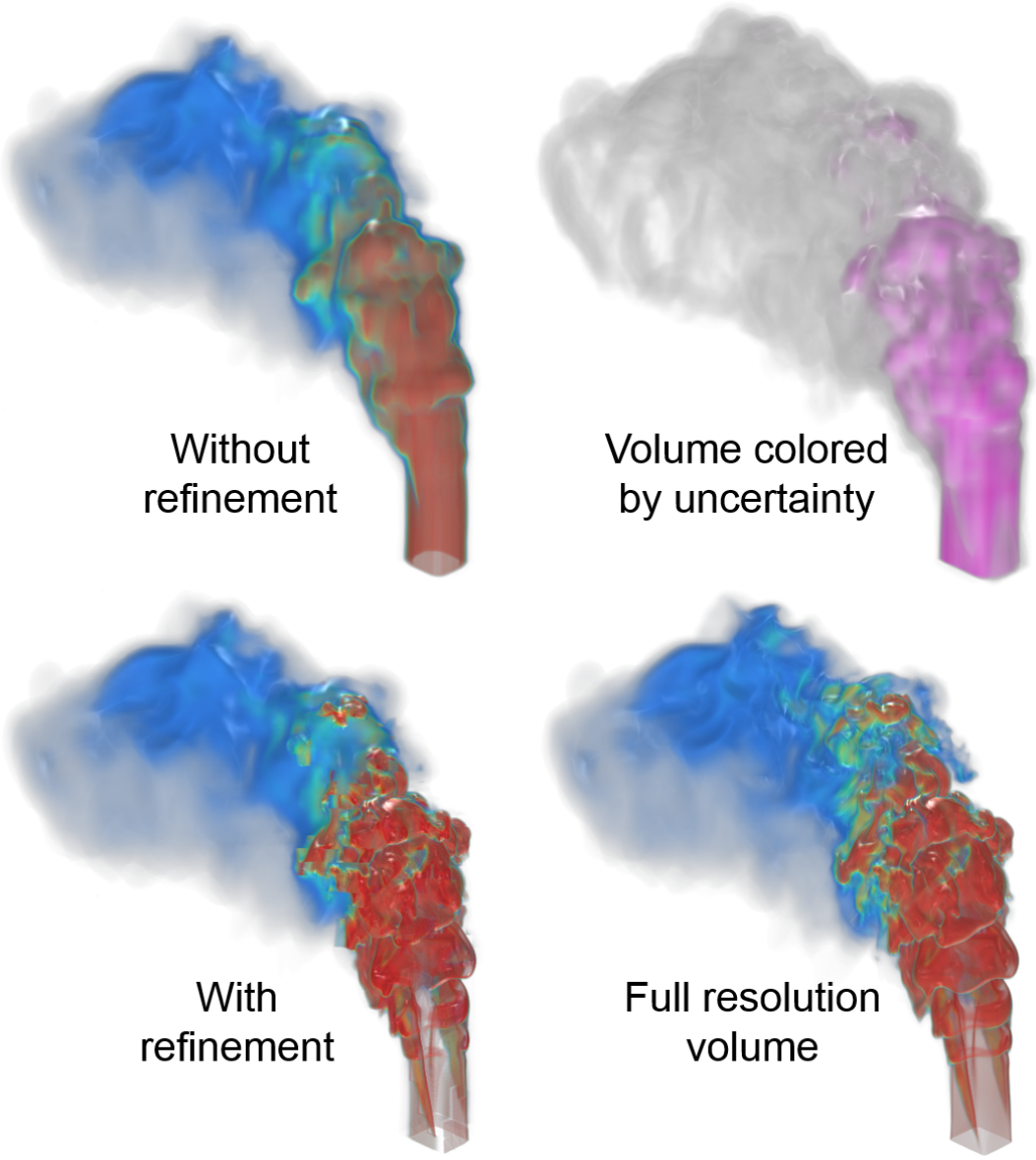


Figure 6.8. Top-left) Not performing refinement hides many of the details present in the flame structure. Top-right) Coloring supervoxels according to their level of uncertainty. Pink regions represent supervoxels which need to be refined to produce a more accurate extraction result. Bottom-left) Supervoxels with high levels of uncertainty are refined revealing additional details in the data. Bottom-right) The full resolution volume for comparison.

We also illustrate the usefulness of the refinement step. While clustering and visualizing the supervoxel volume greatly improves performance in large scale data, many important details can become lost. Figure 6.8 shows supervoxels colored according to their uncertainty level. We can see that the level of uncertainty varies throughout the volume and highlights which supervoxels needs to be refined to obtain a more accurate extraction result. Refining those supervoxels by

sampling the raw high resolution volume reveals the finer structure of the combustion flame and results in an image very similar to the full resolution volume.

6.2.2 Ocean Data

The POP simulation [23] is an ocean circulation model that is integral to many climate based research endeavors. Its complex 3D nature makes possible a detailed analysis of turbulent mixing processes between surface currents and the deep ocean. The ocean simulation dataset has a resolution of $3600 \times 2400 \times 42$. We use the magnitude of the ocean velocity field as the intensity value. In addition, since the depth of the ocean is small compared to its surface area, we use an asymmetric $20 \times 20 \times 1$ layout of initial SLIC cluster centers.

Figure 6.9 shows the output of our extraction scheme on this dataset. It is immediately evident that there are several interesting structures that form in the southern hemisphere, whereas the regions near the equator are more homogeneously distributed. Since we are clustering based on the velocity field, the large numbers of intertwined clusters, each containing a similar velocity magnitude, suggest an increased amount of turbulence in those regions. This is to be expected since regions near the poles are more strongly affected by the Earth's rotation and deep sea mixing. Real-time 3D feature extraction in such a dataset can allow users to explore the intricate structures and driving forces behind this system.

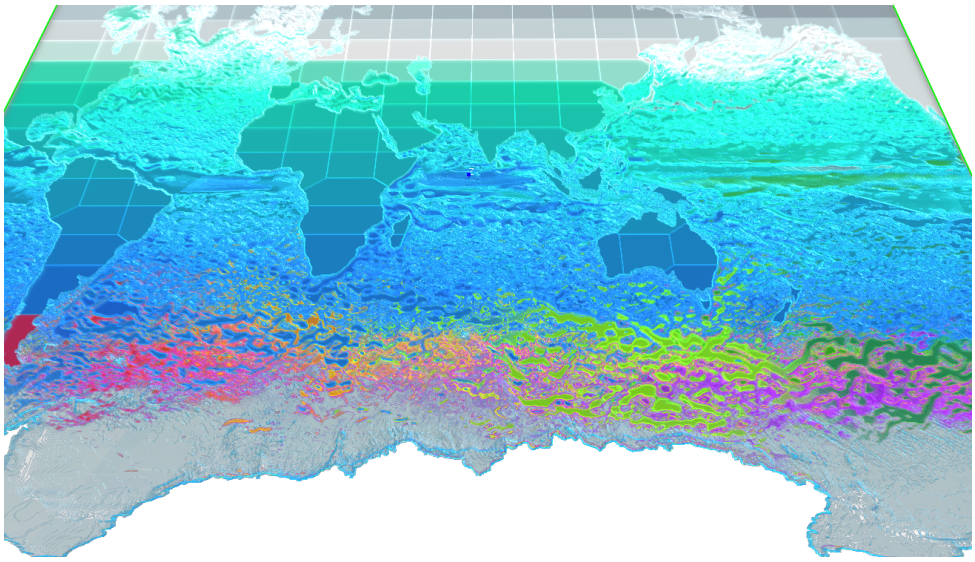


Figure 6.9. A 3D volume rendering of the clustering results on the ocean simulation dataset based on velocity magnitude. The presence of many interconnected features near the south pole show an increased variation in velocity when compared to regions near the equator.

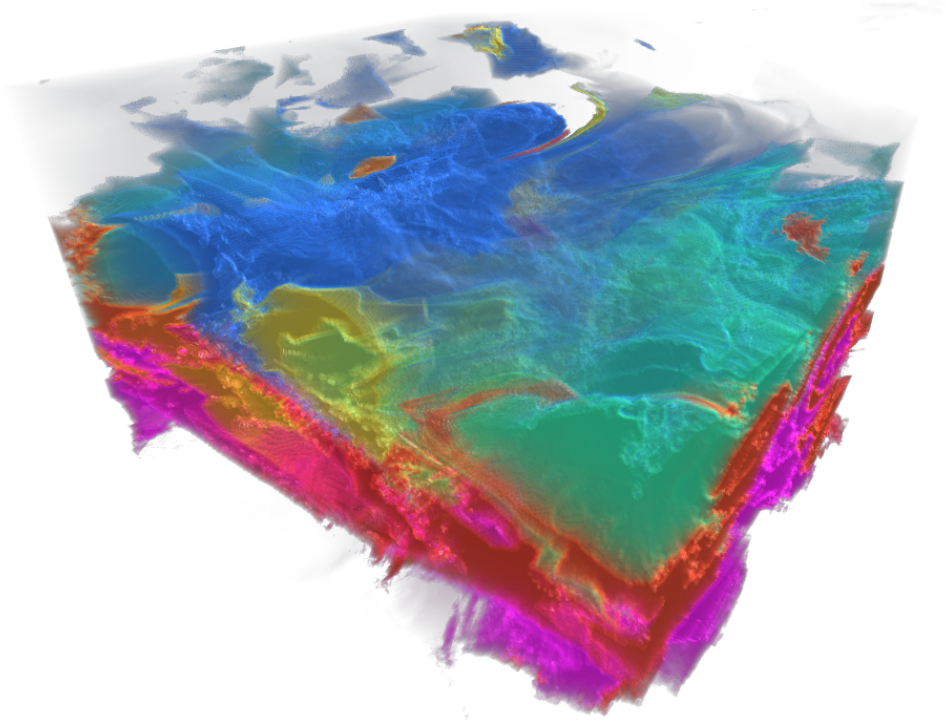


Figure 6.10. A 3D volume rendering of the clustering results based on the vorticity magnitude of the flow simulation dataset. The layer-like structures in the stratified flow become clear once extracted.

6.2.3 Flow Data

The complex interaction between waves and turbulence in flows drives many systems in a multitude of engineering applications and physical phenomena. This particular simulation and study "Parameter Studies of Boussinesq Flows" [7] focuses on understanding the coupling between waves and slow motion by studying layer-like structures in stratified flows as well as columnar structures in rotating flows. With a size of $4096 \times 4096 \times 4096$, it is the largest of all of our test cases. Figure 6.10 shows an example output using our extraction scheme with vorticity magnitude variable. As expected, the resulting features represent the layer-like structures found in the stratified flow and can aid scientists in studying upscale and downscale energy transfers in these systems.

6.2.4 Performance Results

The timing was measured on a desktop PC with two 2.4 GHz Intel Xeon CPUs and 24 GB DDR3 RAM with one GTX Titan X graphics card with 12 GB of GPU memory.

We begin with the combustion dataset and investigate how different supervoxel sizes affect the performance of each of the steps involved in our feature extraction technique. We focus on this particular parameter since it has the largest impact on the overall extraction time. In this example, 512 SLIC clusters were used for each test. Graph 6.11 shows the timing results for each step of the algorithm for a number of supervoxel sizes. We can see that the supervoxel generation time increases slightly as the supervoxel size increases. This is due to the fact that a larger number of raw voxels must be sampled and used to compute the necessary statistical information (mean and standard deviation). On the other hand, the time for the SLIC step decreases as supervoxel size increases because there are fewer supervoxels that need to be clustered. In addition, the merge step time (which also includes the time to generate connectivity information between clusters) also decreases as the supervoxel size increases. Once again, this is due to the ease of working with lower resolution volume. The time to refine the volume into a desired level of detail has the largest cost and depends heavily on the underlying structure of dataset. It will be shown soon that the other test datasets exhibit different trends when it comes to the cost of the refine step. For this particular configuration the optimum supervoxel size is 17^3 with an overall extraction time of 187 ms showing that this type of analysis can be done interactively in real time.

The ocean dataset shows similar trends. In this example, 400 SLIC clusters were used for each test. Graph 6.12 shows the timing results for each step of the algorithm. These times represent very similar trends to those produced by the combustion dataset. However, the overall times are higher since the ocean dataset is larger in scale. Nevertheless we can achieve decently interactive speeds when extracting features in the ocean data. The optimum supervoxel size might be 19^3 with an overall extraction time of 2299 ms. Since this is the last data point, it could be possible that a larger supervoxel size would result in an even lower extraction time.

The combustion and ocean datasets are manageable enough in size so that disk read times are small enough to ensure interactive speeds when performing the supervoxel generation step

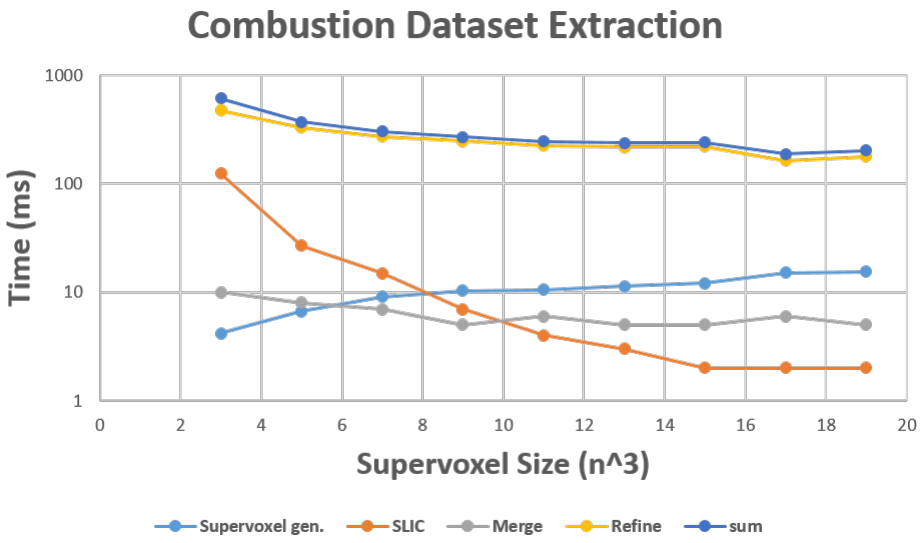


Figure 6.11. A breakdown of the performance results for each of the extraction steps vs. supervoxel size for the combustion dataset.

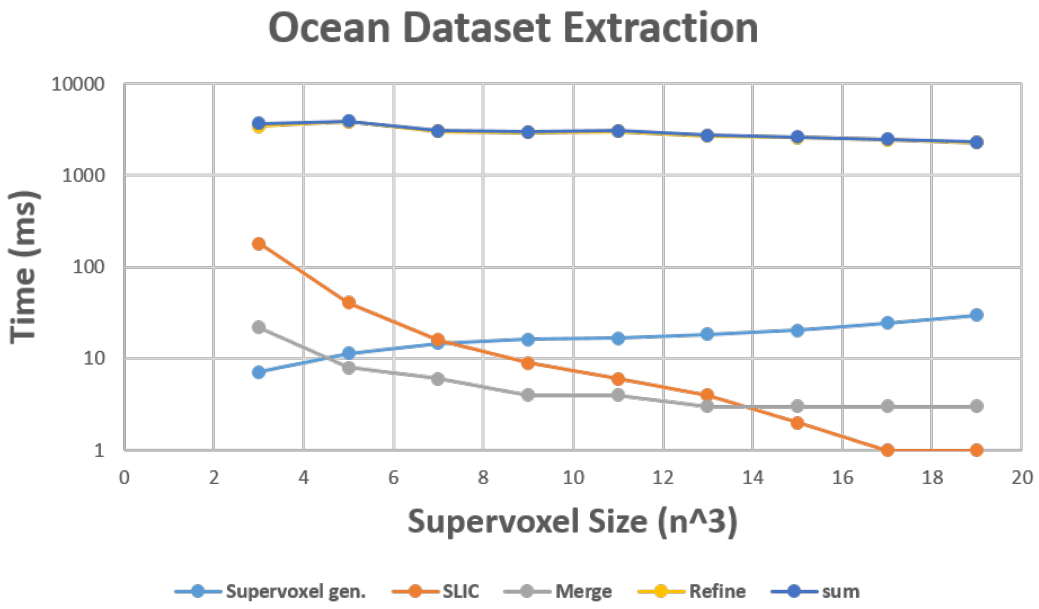


Figure 6.12. A breakdown of the performance results for each of the extraction steps vs. supervoxel size for the ocean dataset.

on a local machine on the fly. However, the flow dataset, which consists of hundreds of GB of information per time step, can incur a very large I/O overhead. As a result, it is often desirable to perform the supervoxel generation step in situ during the corresponding simulation. In this implementation, the simulation will not only produce the large-scale high-resolution volume,

but the much smaller supervoxel representation as well.

Table 6.1. Strong scaling. In-situ supervoxel generation timing for the flow dataset with a constant 8x8x8 supervoxel size.

#Nodes	CPU to GPU (ms)	Supervoxel gen. (ms)	Write (ms)
128	1133	74	527
256	755	37	630
512	618	19	694

We estimate the amount of computation necessary to construct the supervoxel representation if it were done in situ by conducting performance tests on the Titan supercomputer at Oak Ridge National Laboratory. Table 6.1 shows the timing results for constructing an 8x8x8 supervoxel representation using a varying number of compute nodes. We assume that the data for a particular timestep is already locally available in memory and report the time to transfer information into GPU memory, construct the supervoxel representation on the GPU, and then write the supervoxel data out to disk. We can see that the GPU transfer time decreases as the number of compute nodes increases since the overall volume can be split into increasingly smaller components. Similarly, the supervoxel generation step also decreases. Lastly, we see that the disk write times for the supervoxel volume increases slightly with more nodes since there is a larger communication overhead when many nodes need to write to a single file. These times are on the order of a few seconds or less and show that the extra computation necessary for this in-situ step is small in comparison to the normal simulation computation time (which tends to be much larger than a few seconds).

In addition, we can keep the number of compute nodes constant (256 in this case) and look at how different supervoxel sizes affect the time of each step. This is shown in Graph 6.13. The GPU transfer time remains relatively constant since the entire volume subset must be sent to the GPU regardless of the end resolution of the supervoxel volume. Like the earlier results, the supervoxel generation time increases slightly for larger supervoxel sizes since a larger number of raw data values must be sampled in the GPU. However, the time to output the new supervoxel volume to file decreases dramatically since a larger supervoxel size results in less data that needs to be saved. Once again these times are on the order of a few seconds or less per timestep and

will likely reflect only a small portion of the normal computation time of the simulation.

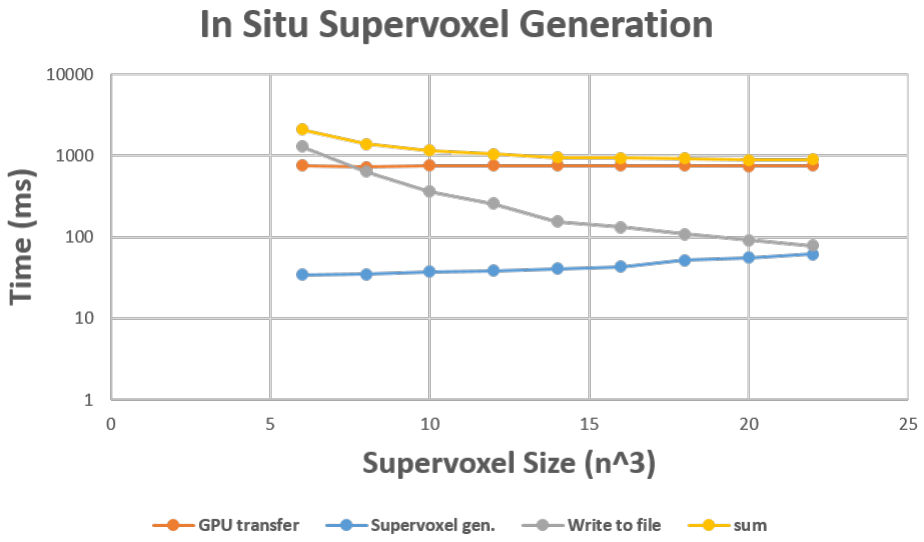


Figure 6.13. In-situ timing results for parallel supervoxel generation vs. supervoxel size.

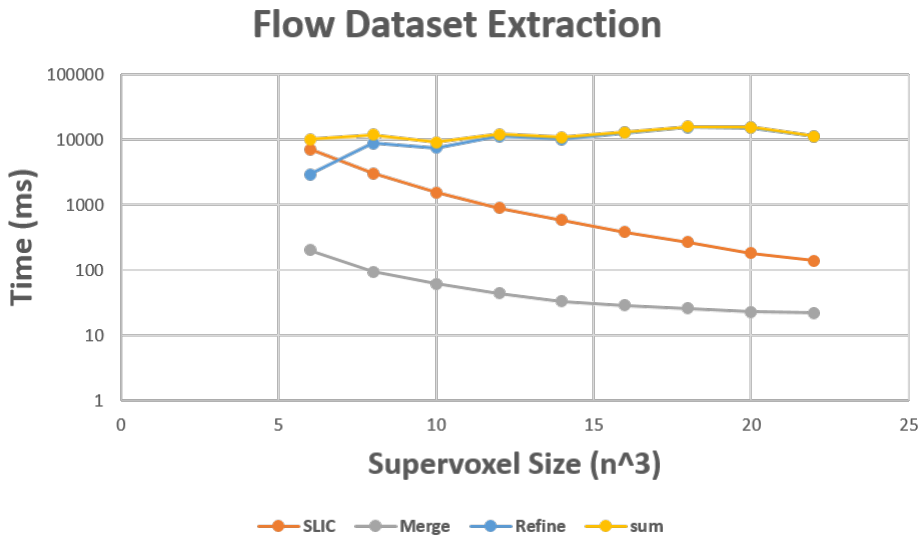


Figure 6.14. A breakdown of the performance results for each of the extraction steps vs. supervoxel size for the flow dataset.

Continuing with the other steps of the algorithm, which are now performed in real time on a local desktop machine, we can see the timing results shown in Graph 6.14. In this example, 1000 SLIC clusters were used for each test. Once again we see the same trends for each of the remaining steps as observed in the previous two datasets. However, the refinement step exhibits

an entirely different trend because it depends heavily on the underlying structure of this dataset. The overall times are larger than the previous examples since this dataset is much larger. In this case, the optimum supervoxel size is 10^3 with an overall extraction time of 9159 ms showing that we can simultaneously extract many coherent features in such a large dataset on a desktop PC with very little overhead.

6.2.5 Data Size Limits on the Desktop PC

The above results demonstrate the ability of this technique to handle datasets of various scales. While certain datasets are easily managed using this scheme, other extreme-scale cases can be handled through some additional in-situ processing. Take the datasets shown in this chapter as an example, we can see that the combustion ($704 \times 540 \times 550$) and ocean ($3600 \times 2400 \times 42$) datasets are easily manageable on a desktop PC using our extraction scheme. However, the much larger flow dataset ($4096 \times 4096 \times 4096$) incurs a disk I/O time that is too large for real time supervoxel generation. This is alleviated either through a potentially lengthy one time preprocessing step into the supervoxel volume or can be computed directly during the simulation itself. Such a representation can be saved in conjunction with the full resolution version. The technique presented in this chapter only fetches the full resolution voxels in regions of the domain with a high uncertainty and thus can extract high resolution features very quickly.

We were able to generate an extraction result for the flow dataset in about 9 seconds, the majority of which consisted of fetching high resolution voxels from disk during the refine step. Moreover, this scheme was able to extract $\sim 10^2$ features simultaneously in such a large volume on a desktop PC. This extraction cost only needs to be repeated if input parameters, such as the number of clusters, SLIC weights, refinement threshold, etc., need to change.

6.2.6 User-driven Extraction v.s. Automatic Extraction

Our feature extraction technique is designed to be run in either a user-driven or automatic manner. In the user-driven mode, input is provided into each step in the algorithm to ensure an extraction result that closely matches the interest of the user. While many of these decisions are based on choosing an appropriate balance between accuracy and performance, others can fundamentally alter which features are extracted. Note that the final uncertainty-driven super-

voxel refinement step attempts to minimize the differences observed in the final extraction result based on initial supervoxel sizes.

Choosing an appropriate distance metric for the SLIC cluster generation has the largest impact on the result. This is where the user can choose which data variables and spatial properties to emphasize. Knowledge over this matter is often domain specific, making a user-driven approach desirable in many cases. Lastly, the uncertainty-driven refinements step allows users to choose what level of accuracy the final extraction result must meet. In certain cases a fast rough extraction is sufficient, while in other cases a detailed feature extraction is necessary to study subtle patterns in the data.

In the automatic approach, the algorithm will generate its best estimate of a desirable feature extraction result without any user input. First, the system automatically chooses an appropriate supervoxel resolution based on the scale of the raw dataset, namely one that allows the supervoxel volume to fit entirely into GPU memory for fast access. Next, it performs the SLIC utilizing a single data variable with equal weights on intensity and spatial proximity. Lastly, the system chooses to refine a default fraction of supervoxels. While the automatic approach uses a very simple set of inputs, users may want to generate a fast general decomposition of the data in order to become better acquainted with its structure before attempting a more detailed exploration.

6.3 Summary

Overall, this chapter introduced a new hybrid feature extraction technique which combines the multi-resolution advantages of supervoxels data representation with a GPU accelerated version of SLIC to efficiently manage large-scale datasets on a desktop PC. Using an uncertainty-based refinement method, users can explore extraction results quickly while enhancing detail in certain regions only when necessary. We are able to show the applicability of this technique to a number of large-scale real world datasets and justify its efficiency through performance results.

Chapter 7

Summary

The goal of this dissertation is to advance visualization and data representation methodologies for large-scale scientific simulations. In this dissertation, several advanced visualization, data representation and management methods are introduced to address some of the grand challenges in the visualization and analysis of volume data from large-scale scientific simulations.

Chapter 3 introduces a memory efficient ray-casting scheme for visualization of unstructured geodesic grid data with GPU acceleration. Such method directly takes the original simulation data representation and provides an efficient and precise analytic solution for scalar and gradient interpolation achieving smooth shading and salient depth cues.

Chapter 4 extends the ray-casting render in Chapter 3 to efficiently perform parallel visualization of geodesic grid data on distributed GPU cluster while keeping the communication overhead and memory footprint at a minimum and avoiding memory contention between simulation and visualization. The design achieves a balanced workload across a thousand GPU processors, and makes it practical to interactively visualize large geodesic grid data. In the future, other data partitioning and distribution schemes are worthwhile to experiment with, including the ones deployed by the simulations, to enable in-situ visualization.

Chapter 5 introduces a scalable data structure for the distance transform of massive data in a distributed environment. Thanks to a carefully designed parallel spatial data structure, it enables efficient management and indexing of large-scale datasets that typically cannot fit into a single machine. It also significantly reduces the computation and communication cost for distance transform of time-varying data using massively parallel machines.

Chapter 6 combines the multi-resolution advantages of a supervoxels data representation with GPU accelerated version of SLIC to facilitate a new hybrid feature extraction technique.

Coupled with an uncertainty-based refinement method, it enables a user to explore extraction results quickly while enhancing detail in certain regions only when necessary. The wide applicability of this technique is tested on a number of large-scale real world datasets and its efficiency is well justified through a series of performance results.

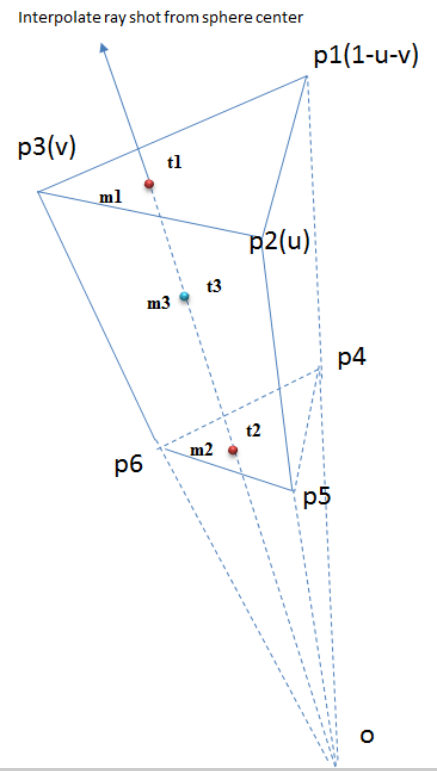
The performance of these methods have been tested on several state-of-art supercomputers, including Cray XK7, Cray XE6 and IBM Blue Gene/P. The performance study has demonstrated great feasibility and scalability for using up to tens of thousands of cores. Many of the approaches have been applied to visualizing large-scale datasets from climate simulations, combustion simulations, Boussinesq flow modeling, and airplane/car aerodynamics simulations. The methods are by no means limited to the tested datasets and are anticipated to be applicable to a broader range of scientific applications. Moreover, some of these methods are also applicable to an in-situ framework, ready to be performed at petascale and exascale.

Appendix A

Math Derivation of Equation 3.10 and 3.12

This section contains a detailed mathmetical derivation of Equation 3.10 and 3.12 using Wolfram Mathematica software.

Data preparation:




```
P1 = {x1, y1, z1}; P2 = {x2, y2, z2}; P3 = {x3, y3, z3};
P4 = {x4, y4, z4};
P5 = {x5, y5, z5};
P6 = {x6, y6, z6};
```

```
V1 = v1, V2 = v2, V3 = v3, V4 = v4, V5 = v5, V6 = v6;
```

```
Clear[ox, oy, oz]
```

```
(*Org is the center of the Earth,i.e.(0,0,0),
and is also the origin of the interpolation ray
 shown as the blue arrow in the above illustration.
  Q is actually the sample point m3 in Figure 3.7 in the dissertation*)
Org = {0, 0, 0};
Q = {qx, qy, qz};
```

Linear system for ray-triangle intersection:

```
E21 = P2 - P1; E31 = P3 - P1; T1 = Org - P1;
```

```
E54 = P5 - P4; E64 = P6 - P4; T2 = Org - P4;
```

```
(*Direction of the interpolation ray*)
Dir = Normalize[Q - Org];
```

```
Dir
```

$$\left\{\frac{qx}{\sqrt{Abs[qx]^2 + Abs[qy]^2 + Abs[qz]^2}},\right.$$
$$\left.\frac{qy}{\sqrt{Abs[qx]^2 + Abs[qy]^2 + Abs[qz]^2}}, \frac{qz}{\sqrt{Abs[qx]^2 + Abs[qy]^2 + Abs[qz]^2}}\right\}$$

$$\mathbf{Dir = \left\{\frac{qx}{\sqrt{(qx)^2 + (qy)^2 + (qz)^2}}, \frac{qy}{\sqrt{(qx)^2 + (qy)^2 + (qz)^2}}, \frac{qz}{\sqrt{(qx)^2 + (qy)^2 + (qz)^2}}\right\}}$$

$$\left\{\frac{qx}{\sqrt{qx^2 + qy^2 + qz^2}}, \frac{qy}{\sqrt{qx^2 + qy^2 + qz^2}}, \frac{qz}{\sqrt{qx^2 + qy^2 + qz^2}}\right\}$$

```
M11 = 1 / Det[{E21, E31, -Dir}]; M21 = 1 / Det[{E54, E64, -Dir}];
```

```
M12 = {Det[{T1, E31, -Dir}], Det[{E21, T1, -Dir}], Det[{E21, E31, T1}]};
M22 = {Det[{T2, E64, -Dir}], Det[{E54, T2, -Dir}], Det[{E54, E64, T1}]};
```

```
M12
```

$$\left\{-\frac{qz\ x3\ y1}{\sqrt{qx^2 + qy^2 + qz^2}} + \frac{qz\ x1\ y3}{\sqrt{qx^2 + qy^2 + qz^2}} + \frac{qy\ x3\ z1}{\sqrt{qx^2 + qy^2 + qz^2}} - \frac{qx\ y3\ z1}{\sqrt{qx^2 + qy^2 + qz^2}} - \right.$$
$$\frac{qy\ x1\ z3}{\sqrt{qx^2 + qy^2 + qz^2}} + \frac{qx\ y1\ z3}{\sqrt{qx^2 + qy^2 + qz^2}}, \frac{qz\ x2\ y1}{\sqrt{qx^2 + qy^2 + qz^2}} - \frac{qz\ x1\ y2}{\sqrt{qx^2 + qy^2 + qz^2}} -$$
$$\frac{qy\ x2\ z1}{\sqrt{qx^2 + qy^2 + qz^2}} + \frac{qx\ y2\ z1}{\sqrt{qx^2 + qy^2 + qz^2}} + \frac{qy\ x1\ z2}{\sqrt{qx^2 + qy^2 + qz^2}} - \frac{qx\ y1\ z2}{\sqrt{qx^2 + qy^2 + qz^2}},$$
$$\left.x3\ y2\ z1 - x2\ y3\ z1 - x3\ y1\ z2 + x1\ y3\ z2 + x2\ y1\ z3 - x1\ y2\ z3\right\}$$

```
R = {u, v, t1} = M11 * M12; R2 = {u2, v2, t2} = M21 * M22;
```

Simplify the solution R

```
R = Simplify[R];
```

```
R
```

$$\Big\{(qz\ x3\ y1 - qz\ x1\ y3 - qy\ x3\ z1 + qx\ y3\ z1 + qy\ x1\ z3 - qx\ y1\ z3)\ /\ (qz\ (x3\ y1 + x1\ y2 - x3\ y2 - x1\ y3 + x2\ (-y1 + y3)) + qy\ (x2\ z1 - x3\ z1 - x1\ z2 + x3\ z2 + x1\ z3 - x2\ z3) + qx\ (-y2\ z1 + y3\ z1 + y1\ z2 - y3\ z2 - y1\ z3 + y2\ z3)),\ (-qz\ x2\ y1 + qz\ x1\ y2 + qy\ x2\ z1 - qx\ y2\ z1 - qy\ x1\ z2 + qx\ y1\ z2)\ /\ (qz\ (x3\ y1 + x1\ y2 - x3\ y2 - x1\ y3 + x2\ (-y1 + y3)) + qy\ (x2\ z1 - x3\ z1 - x1\ z2 + x3\ z2 + x1\ z3 - x2\ z3) + qx\ (-y2\ z1 + y3\ z1 + y1\ z2 - y3\ z2 - y1\ z3 + y2\ z3)),\ \Big(\sqrt{qx^2 + qy^2 + qz^2}\ (-x3\ y2\ z1 + x2\ y3\ z1 + x3\ y1\ z2 - x1\ y3\ z2 - x2\ y1\ z3 + x1\ y2\ z3)\Big)\Big/ (qz\ (x3\ y1 + x1\ y2 - x3\ y2 - x1\ y3 + x2\ (-y1 + y3)) + qy\ (x2\ z1 - x3\ z1 - x1\ z2 + x3\ z2 + x1\ z3 - x2\ z3) + qx\ (-y2\ z1 + y3\ z1 + y1\ z2 - y3\ z2 - y1\ z3 + y2\ z3))\Big\}$$

```
R = R /. {(x3 y1 + x1 y2 - x3 y2 - x1 y3 + x2 (-y1 + y3)) → A7,
    (x2 z1 - x3 z1 - x1 z2 + x3 z2 + x1 z3 - x2 z3) → -A8,
    (-y2 z1 + y3 z1 + y1 z2 - y3 z2 - y1 z3 + y2 z3) → A9, (y3 z1 - y1 z3) → A3}
```

$$\Big\{(qz\ x3\ y1 - qz\ x1\ y3 - qy\ x3\ z1 + qx\ y3\ z1 + qy\ x1\ z3 - qx\ y1\ z3)\ /\ (A9\ qx - A8\ qy + A7\ qz),\ (-qz\ x2\ y1 + qz\ x1\ y2 + qy\ x2\ z1 - qx\ y2\ z1 - qy\ x1\ z2 + qx\ y1\ z2)\ /\ (A9\ qx - A8\ qy + A7\ qz),\ \Big(\sqrt{qx^2 + qy^2 + qz^2}\ (-x3\ y2\ z1 + x2\ y3\ z1 + x3\ y1\ z2 - x1\ y3\ z2 - x2\ y1\ z3 + x1\ y2\ z3)\Big)\Big/ (A9\ qx - A8\ qy + A7\ qz)\Big\}$$

```
R = {(qz (x3 y1 - x1 y3) - qy ( x3 z1 - x1 z3) + qx ( y3 z1 - y1 z3)) / (A9 qx - A8 qy + A7 qz),
    (qz (x1 y2 - x2 y1) - qy (x1 z2 -  x2 z1) + qx (y1 z2 - y2 z1)) / (A9 qx - A8 qy + A7 qz),
    (Sqrt[qx^2 + qy^2 + qz^2] (-x3 y2 z1 + x2 y3 z1 + x3 y1 z2 - x1 y3 z2 - x2 y1 z3 + x1 y2 z3))/
     (A9 qx - A8 qy + A7 qz)};
```

```
(*//////////////////////  SIMPLIFY R  ///////////////////*)
```

```
R = R /. {(x3 y1 - x1 y3) → A1, (x3 z1 - x1 z3) → A2, (y3 z1 - y1 z3) → A3,
    (-x2 y1 + x1 y2) → A4, (-x2 z1 + x1 z2) → A5, (-y2 z1 + y1 z2) → A6,
    -x3 y2 z1 → -A10, (x2 y3 z1 + x3 y1 z2 - x1 y3 z2 - x2 y1 z3 + x1 y2 z3) → -A11}
```

$$\left\{\frac{A3\ qx - A2\ qy + A1\ qz}{A9\ qx - A8\ qy + A7\ qz},\ \frac{A6\ qx - A5\ qy + A4\ qz}{A9\ qx - A8\ qy + A7\ qz},\ \frac{(-A10 - A11)\ \sqrt{qx^2 + qy^2 + qz^2}}{A9\ qx - A8\ qy + A7\ qz}\right\}$$

Barycentric coordinates is the components of R

```
{u, v, t1} = R
```

$$\left\{\frac{A3\ qx - A2\ qy + A1\ qz}{A9\ qx - A8\ qy + A7\ qz},\ \frac{A6\ qx - A5\ qy + A4\ qz}{A9\ qx - A8\ qy + A7\ qz},\ \frac{(-A10 - A11)\ \sqrt{qx^2 + qy^2 + qz^2}}{A9\ qx - A8\ qy + A7\ qz}\right\}$$

Two intersections m1 and m2 are generated when ray hits top and bottom face of prism. Their scalar values can be computed using barycentric coordinates.

```
sm1 = (1 - u - v) V1 + u * V2 + v * V3
```

$$\left(1 - \frac{A3\ qx - A2\ qy + A1\ qz}{A9\ qx - A8\ qy + A7\ qz} - \frac{A6\ qx - A5\ qy + A4\ qz}{A9\ qx - A8\ qy + A7\ qz}\right) V1 + \frac{(A3\ qx - A2\ qy + A1\ qz)\ V2}{A9\ qx - A8\ qy + A7\ qz} + \frac{(A6\ qx - A5\ qy + A4\ qz)\ V3}{A9\ qx - A8\ qy + A7\ qz}$$

```
sm2 = (1 - u - v) V4 + u * V5 + v * V6
```

$$\left(1 - \frac{A3\ qx - A2\ qy + A1\ qz}{A9\ qx - A8\ qy + A7\ qz} - \frac{A6\ qx - A5\ qy + A4\ qz}{A9\ qx - A8\ qy + A7\ qz}\right) V4 + \frac{(A3\ qx - A2\ qy + A1\ qz)\ V5}{A9\ qx - A8\ qy + A7\ qz} + \frac{(A6\ qx - A5\ qy + A4\ qz)\ V6}{A9\ qx - A8\ qy + A7\ qz}$$

```
(1 - u - v) P1 + u * P2 + v * P3
```

$$\left\{\left(1 - \frac{A3\ qx - A2\ qy + A1\ qz}{A9\ qx - A8\ qy + A7\ qz} - \frac{A6\ qx - A5\ qy + A4\ qz}{A9\ qx - A8\ qy + A7\ qz}\right) x1 + \frac{(A3\ qx - A2\ qy + A1\ qz)\ x2}{A9\ qx - A8\ qy + A7\ qz} + \frac{(A6\ qx - A5\ qy + A4\ qz)\ x3}{A9\ qx - A8\ qy + A7\ qz},\right.$$
$$\left(1 - \frac{A3\ qx - A2\ qy + A1\ qz}{A9\ qx - A8\ qy + A7\ qz} - \frac{A6\ qx - A5\ qy + A4\ qz}{A9\ qx - A8\ qy + A7\ qz}\right) y1 + \frac{(A3\ qx - A2\ qy + A1\ qz)\ y2}{A9\ qx - A8\ qy + A7\ qz} + \frac{(A6\ qx - A5\ qy + A4\ qz)\ y3}{A9\ qx - A8\ qy + A7\ qz},$$
$$\left.\left(1 - \frac{A3\ qx - A2\ qy + A1\ qz}{A9\ qx - A8\ qy + A7\ qz} - \frac{A6\ qx - A5\ qy + A4\ qz}{A9\ qx - A8\ qy + A7\ qz}\right) z1 + \frac{(A3\ qx - A2\ qy + A1\ qz)\ z2}{A9\ qx - A8\ qy + A7\ qz} + \frac{(A6\ qx - A5\ qy + A4\ qz)\ z3}{A9\ qx - A8\ qy + A7\ qz}\right\}$$

The origin of the interpolation ray is the center of the spherical.

```
ox = oy = oz = 0
```

```
t1 * Dir + Org
```

$$\left\{\frac{(-A10-A11)\,qx}{A9\,qx-A8\,qy+A7\,qz},\ \frac{(-A10-A11)\,qy}{A9\,qx-A8\,qy+A7\,qz},\ \frac{(-A10-A11)\,qz}{A9\,qx-A8\,qy+A7\,qz}\right\}$$

```
Dir[[1]]
```

t3 can be considered as the distance on the interpolation ray between the center O and the sampling point m3(qx,qy,qz).

```
t3 = (qx - 0) / Dir[[1]]
```

$$\sqrt{qx^2+qy^2+qz^2}$$

t1 can be considered as the distance on the interpolation ray between the center O and the sampling point m1, which is already computed. I rewrite it here for the ease of reference.

```
t1
```

$$\frac{(-A10-A11)\sqrt{qx^2+qy^2+qz^2}}{A9\,qx-A8\,qy+A7\,qz}$$

Similarily, t2 can be considered as the distance on the interpolation ray between the center O and the sampling point m2. By similar triangles:

```
t2 = t1 * EuclideanDistance[Org, P4] / EuclideanDistance[Org, P1]
```

$$\left((-A10-A11)\sqrt{qx^2+qy^2+qz^2}\ \sqrt{Abs[x4]^2+Abs[y4]^2+Abs[z4]^2}\right)\Big/\left((A9\,qx-A8\,qy+A7\,qz)\sqrt{Abs[x1]^2+Abs[y1]^2+Abs[z1]^2}\right)$$

```
t2 = (-A10 - A11) Sqrt[qx^2 + qy^2 + qz^2] Sqrt[(x4)^2 + (y4)^2 + (z4)^2] / ((A9 qx - A8 qy + A7 qz) Sqrt[(x1)^2 + (y1)^2 + (z1)^2])
```

$$\frac{(-A10-A11)\sqrt{qx^2+qy^2+qz^2}\ \sqrt{x4^2+y4^2+z4^2}}{(A9\,qx-A8\,qy+A7\,qz)\sqrt{x1^2+y1^2+z1^2}}$$

t is the ratio on the line segment m2m1, which is used as parameter for linear interpolation of scalar value of m3 based on both the scalar values of m1 and m2.

```
t = Simplify[(t3 - t2) / (t1 - t2)]
```

$$-\left(\left(A9\ qx\ \sqrt{x1^2+y1^2+z1^2}\ -A8\ qy\ \sqrt{x1^2+y1^2+z1^2}\ +\right.\right.$$
$$\left.A7\ qz\ \sqrt{x1^2+y1^2+z1^2}\ +A10\ \sqrt{x4^2+y4^2+z4^2}\ +A11\ \sqrt{x4^2+y4^2+z4^2}\right)\Big/$$
$$\left.\left((A10+A11)\left(\sqrt{x1^2+y1^2+z1^2}\ -\sqrt{x4^2+y4^2+z4^2}\right)\right)\right)$$

$$-\left(\mathbf{A9\ qx\ \sqrt{x1^2+y1^2+z1^2}\ -A8\ qy\ \sqrt{x1^2+y1^2+z1^2}\ +}\right.$$
$$\mathbf{A7\ qz\ \sqrt{x1^2+y1^2+z1^2}\ +A10\ \sqrt{x4^2+y4^2+z4^2}\ +A11\ \sqrt{x4^2+y4^2+z4^2}}\Big)\Big/$$
$$\left.\mathbf{\left((A10+A11)\left(\sqrt{x1^2+y1^2+z1^2}\ -\sqrt{x4^2+y4^2+z4^2}\right)\right)}\right)$$

```
t = t /. {Sqrt[x1^2 + y1^2 + z1^2] → OP1, Sqrt[x4^2 + y4^2 + z4^2] → OP4}
```

$$-\frac{A10\ OP4+A11\ OP4+A9\ OP1\ qx-A8\ OP1\ qy+A7\ OP1\ qz}{(A10+A11)\ (OP1-OP4)}$$

```
t = -((A10 + A11) OP4 + OP1 (A9 qx - A8 qy + A7 qz))/((A10 + A11) (OP1 - OP4))
```

$$-\frac{(A10+A11)\ OP4+OP1\ (A9\ qx-A8\ qy+A7\ qz)}{(A10+A11)\ (OP1-OP4)}$$

Interpolate the scalar value on sampling point m3 using t, sm2 and sm1

```
sm3 = (1 - t) * sm2 + t * sm1
```

$$-\left(\left(\left((A10 + A11)\ OP4 + OP1\ (A9\ qx - A8\ qy + A7\ qz)\right)\right.\right.$$
$$\left.\left.\left(\left(1 - \frac{A3\ qx - A2\ qy + A1\ qz}{A9\ qx - A8\ qy + A7\ qz} - \frac{A6\ qx - A5\ qy + A4\ qz}{A9\ qx - A8\ qy + A7\ qz}\right) V1 + \frac{(A3\ qx - A2\ qy + A1\ qz)\ V2}{A9\ qx - A8\ qy + A7\ qz} + \right.\right.\right.$$
$$\left.\left.\left.\frac{(A6\ qx - A5\ qy + A4\ qz)\ V3}{A9\ qx - A8\ qy + A7\ qz}\right)\right) \Big/ \left((A10 + A11)\ (OP1 - OP4)\right)\right) +$$
$$\left(1 + \frac{(A10 + A11)\ OP4 + OP1\ (A9\ qx - A8\ qy + A7\ qz)}{(A10 + A11)\ (OP1 - OP4)}\right)$$
$$\left(\left(1 - \frac{A3\ qx - A2\ qy + A1\ qz}{A9\ qx - A8\ qy + A7\ qz} - \frac{A6\ qx - A5\ qy + A4\ qz}{A9\ qx - A8\ qy + A7\ qz}\right) V4 + \right.$$
$$\left.\frac{(A3\ qx - A2\ qy + A1\ qz)\ V5}{A9\ qx - A8\ qy + A7\ qz} + \frac{(A6\ qx - A5\ qy + A4\ qz)\ V6}{A9\ qx - A8\ qy + A7\ qz}\right)$$

```
sm3Temp = Simplify[-(((((A10 + A11) OP4 + OP1 (A9 qx - A8 qy + A7 qz))
          ((1 - (A3 qx - A2 qy + A1 qz)/(A9 qx - A8 qy + A7 qz) - (A6 qx - A5 qy + A4 qz)/(A9 qx - A8 qy + A7 qz)) V1 + (A3 qx - A2 qy + A1 qz) V2/(A9 qx - A8 qy + A7 qz) +
            (A6 qx - A5 qy + A4 qz) V3/(A9 qx - A8 qy + A7 qz)))/((A10 + A11) (OP1 - OP4))) +
    (1 + ((A10 + A11) OP4 + OP1 (A9 qx - A8 qy + A7 qz))/((A10 + A11) (OP1 - OP4)))
     ((1 - (A3 qx - A2 qy + A1 qz)/(A9 qx - A8 qy + A7 qz) - (A6 qx - A5 qy + A4 qz)/(A9 qx - A8 qy + A7 qz)) V4 +
       (A3 qx - A2 qy + A1 qz) V5/(A9 qx - A8 qy + A7 qz) + (A6 qx - A5 qy + A4 qz) V6/(A9 qx - A8 qy + A7 qz)))]
```

$$\frac{1}{(A10 + A11)\ (OP1 - OP4)\ (A9\ qx - A8\ qy + A7\ qz)}$$
$$(-(A10\ OP4 + A11\ OP4 + OP1\ (A9\ qx - A8\ qy + A7\ qz))$$
$$(A9\ qx\ V1 + A2\ qy\ V1 + A5\ qy\ V1 - A8\ qy\ V1 - A1\ qz\ V1 - A4\ qz\ V1 + A7\ qz\ V1 -$$
$$A2\ qy\ V2 + A1\ qz\ V2 + A3\ qx\ (-V1 + V2) - A5\ qy\ V3 + A4\ qz\ V3 + A6\ qx\ (-V1 + V3)) +$$
$$OP1\ (A10 + A11 + A9\ qx - A8\ qy + A7\ qz)\ (A9\ qx\ V4 + A2\ qy\ V4 + A5\ qy\ V4 -$$
$$A8\ qy\ V4 - A1\ qz\ V4 - A4\ qz\ V4 + A7\ qz\ V4 - A2\ qy\ V5 + A1\ qz\ V5 +$$
$$A3\ qx\ (-V4 + V5) - A5\ qy\ V6 + A4\ qz\ V6 + A6\ qx\ (-V4 + V6)))$$

```
Collect[
 (- (A10 OP4 + A11 OP4 + OP1 (A9 qx - A8 qy + A7 qz)) (A9 qx V1 + A2 qy V1 + A5 qy V1 - A8 qy V1 -
       A1 qz V1 - A4 qz V1 + A7 qz V1 - A2 qy V2 + A1 qz V2 + A3 qx (-V1 + V2) -
       A5 qy V3 + A4 qz V3 + A6 qx (-V1 + V3)) + OP1 (A10 + A11 + A9 qx - A8 qy + A7 qz)
     (A9 qx V4 + A2 qy V4 + A5 qy V4 - A8 qy V4 - A1 qz V4 - A4 qz V4 + A7 qz V4 - A2 qy V5 +
       A1 qz V5 + A3 qx (-V4 + V5) - A5 qy V6 + A4 qz V6 + A6 qx (-V4 + V6))), {qx, qy, qz}]
```

```
qz (A1 A10 OP4 V1 + A1 A11 OP4 V1 + A10 A4 OP4 V1 + A11 A4 OP4 V1 - A10 A7 OP4 V1 -
     A11 A7 OP4 V1 - A1 A10 OP4 V2 - A1 A11 OP4 V2 - A10 A4 OP4 V3 - A11 A4 OP4 V3 -
     A1 A10 OP1 V4 - A1 A11 OP1 V4 - A10 A4 OP1 V4 - A11 A4 OP1 V4 + A10 A7 OP1 V4 +
     A11 A7 OP1 V4 + A1 A10 OP1 V5 + A1 A11 OP1 V5 + A10 A4 OP1 V6 + A11 A4 OP1 V6) +
 qz^2 (A1 A7 OP1 V1 + A4 A7 OP1 V1 - A7^2 OP1 V1 - A1 A7 OP1 V2 - A4 A7 OP1 V3 -
     A1 A7 OP1 V4 - A4 A7 OP1 V4 + A7^2 OP1 V4 + A1 A7 OP1 V5 + A4 A7 OP1 V6) +
 qy^2 (A2 A8 OP1 V1 + A5 A8 OP1 V1 - A8^2 OP1 V1 - A2 A8 OP1 V2 - A5 A8 OP1 V3 -
     A2 A8 OP1 V4 - A5 A8 OP1 V4 + A8^2 OP1 V4 + A2 A8 OP1 V5 + A5 A8 OP1 V6) +
 qx^2 (-A9^2 OP1 V1 - A3 A9 OP1 (-V1 + V2) - A6 A9 OP1 (-V1 + V3) +
     A9^2 OP1 V4 + A3 A9 OP1 (-V4 + V5) + A6 A9 OP1 (-V4 + V6)) +
 qy (-A10 A2 OP4 V1 - A11 A2 OP4 V1 - A10 A5 OP4 V1 - A11 A5 OP4 V1 + A10 A8 OP4 V1 +
     A11 A8 OP4 V1 + A10 A2 OP4 V2 + A11 A2 OP4 V2 + A10 A5 OP4 V3 +
     A11 A5 OP4 V3 + A10 A2 OP1 V4 + A11 A2 OP1 V4 + A10 A5 OP1 V4 + A11 A5 OP1 V4 -
     A10 A8 OP1 V4 - A11 A8 OP1 V4 - A10 A2 OP1 V5 - A11 A2 OP1 V5 - A10 A5 OP1 V6 -
     A11 A5 OP1 V6 + qz (-A2 A7 OP1 V1 - A5 A7 OP1 V1 - A1 A8 OP1 V1 -
         A4 A8 OP1 V1 + 2 A7 A8 OP1 V1 + A2 A7 OP1 V2 + A1 A8 OP1 V2 + A5 A7 OP1 V3 +
         A4 A8 OP1 V3 + A2 A7 OP1 V4 + A5 A7 OP1 V4 + A1 A8 OP1 V4 + A4 A8 OP1 V4 -
         2 A7 A8 OP1 V4 - A2 A7 OP1 V5 - A1 A8 OP1 V5 - A5 A7 OP1 V6 - A4 A8 OP1 V6)) +
 qx (-A10 A9 OP4 V1 - A11 A9 OP4 V1 - A10 A3 OP4 (-V1 + V2) - A11 A3 OP4 (-V1 + V2) -
     A10 A6 OP4 (-V1 + V3) - A11 A6 OP4 (-V1 + V3) + A10 A9 OP1 V4 +
     A11 A9 OP1 V4 + A10 A3 OP1 (-V4 + V5) + A11 A3 OP1 (-V4 + V5) +
     A10 A6 OP1 (-V4 + V6) + A11 A6 OP1 (-V4 + V6) +
     qz (A1 A9 OP1 V1 + A4 A9 OP1 V1 - 2 A7 A9 OP1 V1 - A1 A9 OP1 V2 - A3 A7 OP1 (-V1 + V2) -
         A4 A9 OP1 V3 - A6 A7 OP1 (-V1 + V3) - A1 A9 OP1 V4 - A4 A9 OP1 V4 + 2 A7 A9 OP1 V4 +
         A1 A9 OP1 V5 + A3 A7 OP1 (-V4 + V5) + A4 A9 OP1 V6 + A6 A7 OP1 (-V4 + V6)) +
     qy (-A2 A9 OP1 V1 - A5 A9 OP1 V1 + 2 A8 A9 OP1 V1 + A2 A9 OP1 V2 + A3 A8 OP1 (-V1 + V2) +
         A5 A9 OP1 V3 + A6 A8 OP1 (-V1 + V3) + A2 A9 OP1 V4 + A5 A9 OP1 V4 - 2 A8 A9 OP1 V4 -
         A2 A9 OP1 V5 - A3 A8 OP1 (-V4 + V5) - A5 A9 OP1 V6 - A6 A8 OP1 (-V4 + V6)))
```

In[1]:=

```
(*Further simplify the expression by replacing the lengthy
 coefficient between a pair of parentheses with a single symbol*)
```

```
qz (A1 A10 OP4 V1 + A1 A11 OP4 V1 + A10 A4 OP4 V1 + A11 A4 OP4 V1 - A10 A7 OP4 V1 -
     A11 A7 OP4 V1 - A1 A10 OP4 V2 - A1 A11 OP4 V2 - A10 A4 OP4 V3 - A11 A4 OP4 V3 -
     A1 A10 OP1 V4 - A1 A11 OP1 V4 - A10 A4 OP1 V4 - A11 A4 OP1 V4 + A10 A7 OP1 V4 +
     A11 A7 OP1 V4 + A1 A10 OP1 V5 + A1 A11 OP1 V5 + A10 A4 OP1 V6 + A11 A4 OP1 V6) +
  qz^2 (A1 A7 OP1 V1 + A4 A7 OP1 V1 - A7^2 OP1 V1 - A1 A7 OP1 V2 - A4 A7 OP1 V3 -
     A1 A7 OP1 V4 - A4 A7 OP1 V4 + A7^2 OP1 V4 + A1 A7 OP1 V5 + A4 A7 OP1 V6) +
  qy^2 (A2 A8 OP1 V1 + A5 A8 OP1 V1 - A8^2 OP1 V1 - A2 A8 OP1 V2 - A5 A8 OP1 V3 -
     A2 A8 OP1 V4 - A5 A8 OP1 V4 + A8^2 OP1 V4 + A2 A8 OP1 V5 + A5 A8 OP1 V6) +
```

```
qx^2 (-A9^2 OP1 V1 - A3 A9 OP1 (-V1 + V2) - A6 A9 OP1 (-V1 + V3) +
    A9^2 OP1 V4 + A3 A9 OP1 (-V4 + V5) + A6 A9 OP1 (-V4 + V6)) +
 qy (-A10 A2 OP4 V1 - A11 A2 OP4 V1 - A10 A5 OP4 V1 - A11 A5 OP4 V1 + A10 A8 OP4 V1 +
    A11 A8 OP4 V1 + A10 A2 OP4 V2 + A11 A2 OP4 V2 + A10 A5 OP4 V3 +
    A11 A5 OP4 V3 + A10 A2 OP1 V4 + A11 A2 OP1 V4 + A10 A5 OP1 V4 + A11 A5 OP1 V4 -
    A10 A8 OP1 V4 - A11 A8 OP1 V4 - A10 A2 OP1 V5 - A11 A2 OP1 V5 - A10 A5 OP1 V6 -
    A11 A5 OP1 V6 + qz (-A2 A7 OP1 V1 - A5 A7 OP1 V1 - A1 A8 OP1 V1 -
        A4 A8 OP1 V1 + 2 A7 A8 OP1 V1 + A2 A7 OP1 V2 + A1 A8 OP1 V2 + A5 A7 OP1 V3 +
        A4 A8 OP1 V3 + A2 A7 OP1 V4 + A5 A7 OP1 V4 + A1 A8 OP1 V4 + A4 A8 OP1 V4 -
        2 A7 A8 OP1 V4 - A2 A7 OP1 V5 - A1 A8 OP1 V5 - A5 A7 OP1 V6 - A4 A8 OP1 V6)) +
 qx (-A10 A9 OP4 V1 - A11 A9 OP4 V1 - A10 A3 OP4 (-V1 + V2) - A11 A3 OP4 (-V1 + V2) -
    A10 A6 OP4 (-V1 + V3) - A11 A6 OP4 (-V1 + V3) + A10 A9 OP1 V4 +
    A11 A9 OP1 V4 + A10 A3 OP1 (-V4 + V5) + A11 A3 OP1 (-V4 + V5) +
    A10 A6 OP1 (-V4 + V6) + A11 A6 OP1 (-V4 + V6) +
    qz (A1 A9 OP1 V1 + A4 A9 OP1 V1 - 2 A7 A9 OP1 V1 - A1 A9 OP1 V2 - A3 A7 OP1 (-V1 + V2) -
        A4 A9 OP1 V3 - A6 A7 OP1 (-V1 + V3) - A1 A9 OP1 V4 - A4 A9 OP1 V4 + 2 A7 A9 OP1 V4 +
        A1 A9 OP1 V5 + A3 A7 OP1 (-V4 + V5) + A4 A9 OP1 V6 + A6 A7 OP1 (-V4 + V6)) +
    qy (-A2 A9 OP1 V1 - A5 A9 OP1 V1 + 2 A8 A9 OP1 V1 + A2 A9 OP1 V2 + A3 A8 OP1 (-V1 + V2) +
        A5 A9 OP1 V3 + A6 A8 OP1 (-V1 + V3) + A2 A9 OP1 V4 + A5 A9 OP1 V4 - 2 A8 A9 OP1 V4 -
        A2 A9 OP1 V5 - A3 A8 OP1 (-V4 + V5) - A5 A9 OP1 V6 - A6 A8 OP1 (-V4 + V6))) /.
(*/.*) {(A1 A10 OP4 V1 + A1 A11 OP4 V1 + A10 A4 OP4 V1 + A11 A4 OP4 V1 - A10 A7 OP4 V1 -
    A11 A7 OP4 V1 - A1 A10 OP4 V2 - A1 A11 OP4 V2 - A10 A4 OP4 V3 - A11 A4 OP4 V3 -
    A1 A10 OP1 V4 - A1 A11 OP1 V4 - A10 A4 OP1 V4 - A11 A4 OP1 V4 + A10 A7 OP1 V4 +
    A11 A7 OP1 V4 + A1 A10 OP1 V5 + A1 A11 OP1 V5 + A10 A4 OP1 V6 + A11 A4 OP1 V6) → C1,
 (A1 A7 OP1 V1 + A4 A7 OP1 V1 - A7^2 OP1 V1 - A1 A7 OP1 V2 - A4 A7 OP1 V3 -
    A1 A7 OP1 V4 - A4 A7 OP1 V4 + A7^2 OP1 V4 + A1 A7 OP1 V5 + A4 A7 OP1 V6) → C2,
 (A2 A8 OP1 V1 + A5 A8 OP1 V1 - A8^2 OP1 V1 - A2 A8 OP1 V2 - A5 A8 OP1 V3 -
    A2 A8 OP1 V4 - A5 A8 OP1 V4 + A8^2 OP1 V4 + A2 A8 OP1 V5 + A5 A8 OP1 V6) → C3,
 (-A9^2 OP1 V1 - A3 A9 OP1 (-V1 + V2) - A6 A9 OP1 (-V1 + V3) + A9^2 OP1 V4 +
    A3 A9 OP1 (-V4 + V5) + A6 A9 OP1 (-V4 + V6)) → C4,
 (-A2 A7 OP1 V1 - A5 A7 OP1 V1 - A1 A8 OP1 V1 - A4 A8 OP1 V1 + 2 A7 A8 OP1 V1 +
    A2 A7 OP1 V2 + A1 A8 OP1 V2 + A5 A7 OP1 V3 + A4 A8 OP1 V3 + A2 A7 OP1 V4 +
    A5 A7 OP1 V4 + A1 A8 OP1 V4 + A4 A8 OP1 V4 - 2 A7 A8 OP1 V4 -
    A2 A7 OP1 V5 - A1 A8 OP1 V5 - A5 A7 OP1 V6 - A4 A8 OP1 V6) → C5,
-A10 A9 OP4 V1 - A11 A9 OP4 V1 - A10 A3 OP4 (-V1 + V2) - A11 A3 OP4 (-V1 + V2) -
  A10 A6 OP4 (-V1 + V3) - A11 A6 OP4 (-V1 + V3) + A10 A9 OP1 V4 +
  A11 A9 OP1 V4 + A10 A3 OP1 (-V4 + V5) + A11 A3 OP1 (-V4 + V5) +
  A10 A6 OP1 (-V4 + V6) + A11 A6 OP1 (-V4 + V6) → C6}
```

```
C4 qx^2 + C3 qy^2 + C1 qz + C2 qz^2 +
 qy (C5 qz - A10 A2 OP4 V1 - A11 A2 OP4 V1 - A10 A5 OP4 V1 - A11 A5 OP4 V1 + A10 A8 OP4 V1 +
     A11 A8 OP4 V1 + A10 A2 OP4 V2 + A11 A2 OP4 V2 + A10 A5 OP4 V3 + A11 A5 OP4 V3 +
     A10 A2 OP1 V4 + A11 A2 OP1 V4 + A10 A5 OP1 V4 + A11 A5 OP1 V4 - A10 A8 OP1 V4 -
     A11 A8 OP1 V4 - A10 A2 OP1 V5 - A11 A2 OP1 V5 - A10 A5 OP1 V6 - A11 A5 OP1 V6) +
 qx (C6 + qz (A1 A9 OP1 V1 + A4 A9 OP1 V1 - 2 A7 A9 OP1 V1 - A1 A9 OP1 V2 - A3 A7 OP1 (-V1 + V2) -
         A4 A9 OP1 V3 - A6 A7 OP1 (-V1 + V3) - A1 A9 OP1 V4 - A4 A9 OP1 V4 + 2 A7 A9 OP1 V4 +
         A1 A9 OP1 V5 + A3 A7 OP1 (-V4 + V5) + A4 A9 OP1 V6 + A6 A7 OP1 (-V4 + V6)) +
     qy (-A2 A9 OP1 V1 - A5 A9 OP1 V1 + 2 A8 A9 OP1 V1 + A2 A9 OP1 V2 + A3 A8 OP1 (-V1 + V2) +
         A5 A9 OP1 V3 + A6 A8 OP1 (-V1 + V3) + A2 A9 OP1 V4 + A5 A9 OP1 V4 - 2 A8 A9 OP1 V4 -
         A2 A9 OP1 V5 - A3 A8 OP1 (-V4 + V5) - A5 A9 OP1 V6 - A6 A8 OP1 (-V4 + V6)))
```

```
(*Continue from above to substitute lengthy coefficient with short symbol*)
```

```
C4 qx^2 + C3 qy^2 + C1 qz + C2 qz^2 +
  qy (C5 qz - A10 A2 OP4 V1 - A11 A2 OP4 V1 - A10 A5 OP4 V1 - A11 A5 OP4 V1 + A10 A8 OP4 V1 +
      A11 A8 OP4 V1 + A10 A2 OP4 V2 + A11 A2 OP4 V2 + A10 A5 OP4 V3 + A11 A5 OP4 V3 +
      A10 A2 OP1 V4 + A11 A2 OP1 V4 + A10 A5 OP1 V4 + A11 A5 OP1 V4 - A10 A8 OP1 V4 -
      A11 A8 OP1 V4 - A10 A2 OP1 V5 - A11 A2 OP1 V5 - A10 A5 OP1 V6 - A11 A5 OP1 V6) + qx (C6 +
      qz (A1 A9 OP1 V1 + A4 A9 OP1 V1 - 2 A7 A9 OP1 V1 - A1 A9 OP1 V2 - A3 A7 OP1 (-V1 + V2) -
          A4 A9 OP1 V3 - A6 A7 OP1 (-V1 + V3) - A1 A9 OP1 V4 - A4 A9 OP1 V4 + 2 A7 A9 OP1 V4 +
          A1 A9 OP1 V5 + A3 A7 OP1 (-V4 + V5) + A4 A9 OP1 V6 + A6 A7 OP1 (-V4 + V6)) +
      qy (-A2 A9 OP1 V1 - A5 A9 OP1 V1 + 2 A8 A9 OP1 V1 + A2 A9 OP1 V2 + A3 A8 OP1 (-V1 + V2) +
          A5 A9 OP1 V3 + A6 A8 OP1 (-V1 + V3) + A2 A9 OP1 V4 + A5 A9 OP1 V4 - 2 A8 A9 OP1 V4 -
          A2 A9 OP1 V5 - A3 A8 OP1 (-V4 + V5) - A5 A9 OP1 V6 - A6 A8 OP1 (-V4 + V6))) /.
 {(-A2 A9 OP1 V1 - A5 A9 OP1 V1 + 2 A8 A9 OP1 V1 + A2 A9 OP1 V2 + A3 A8 OP1 (-V1 + V2) +
      A5 A9 OP1 V3 + A6 A8 OP1 (-V1 + V3) + A2 A9 OP1 V4 + A5 A9 OP1 V4 - 2 A8 A9 OP1 V4 -
      A2 A9 OP1 V5 - A3 A8 OP1 (-V4 + V5) - A5 A9 OP1 V6 - A6 A8 OP1 (-V4 + V6)) → C7,
  (A1 A9 OP1 V1 + A4 A9 OP1 V1 - 2 A7 A9 OP1 V1 - A1 A9 OP1 V2 - A3 A7 OP1 (-V1 + V2) -
      A4 A9 OP1 V3 - A6 A7 OP1 (-V1 + V3) - A1 A9 OP1 V4 - A4 A9 OP1 V4 + 2 A7 A9 OP1 V4 +
      A1 A9 OP1 V5 + A3 A7 OP1 (-V4 + V5) + A4 A9 OP1 V6 + A6 A7 OP1 (-V4 + V6)) → C8,
  -A10 A2 OP4 V1 - A11 A2 OP4 V1 - A10 A5 OP4 V1 - A11 A5 OP4 V1 + A10 A8 OP4 V1 +
    A11 A8 OP4 V1 + A10 A2 OP4 V2 + A11 A2 OP4 V2 + A10 A5 OP4 V3 + A11 A5 OP4 V3 +
    A10 A2 OP1 V4 + A11 A2 OP1 V4 + A10 A5 OP1 V4 + A11 A5 OP1 V4 - A10 A8 OP1 V4 -
    A11 A8 OP1 V4 - A10 A2 OP1 V5 - A11 A2 OP1 V5 - A10 A5 OP1 V6 - A11 A5 OP1 V6 → C9}
```

```
C4 qx^2 + C3 qy^2 + C1 qz + C2 qz^2 + qy (C9 + C5 qz) + qx (C6 + C7 qy + C8 qz)
```

```
(*Finally, we have reached the Equation 3.10 in the dissertation*)
```

```
sm3 = (C4 qx^2 + C3 qy^2 + C1 qz + C2 qz^2 + qy (C9 + C5 qz) + qx (C6 + C7 qy + C8 qz)) /
    ((A10 + A11) (OP1 - OP4) (A9 qx - A8 qy + A7 qz));
```

```
sm3
```

$$\left(C4\ qx^2 + C3\ qy^2 + C1\ qz + C2\ qz^2 + qy\ (C9 + C5\ qz) + qx\ (C6 + C7\ qy + C8\ qz)\right) /$$
$$((A10 + A11)\ (OP1 - OP4)\ (A9\ qx - A8\ qy + A7\ qz))$$

```
sm3 /. (1 / ((A10 + A11) (OP1 - OP4))) → C0
```

$$\left(C0\ \left(C4\ qx^2 + C3\ qy^2 + C1\ qz + C2\ qz^2 + qy\ (C9 + C5\ qz) + qx\ (C6 + C7\ qy + C8\ qz)\right)\right) /$$
$$(A9\ qx - A8\ qy + A7\ qz)$$

```
(* Next, we move to derive Equation 3.12*)
```

```
Clear[t]
```

```
(*Parametric expression of qx, qy,
and qz using ray equation(o+t*d). The (x0,y0,z0) is the eye position*)
```

```
qx = x0 + λ * dx; qy = y0 + λ * dy; qz = z0 + λ * dz;
```

```
(*nominator of sm3*)
Collect[C0 (C4 qx^2 + C3 qy^2 + C1 qz + C2 qz^2 + qy (C9 + C5 qz) + qx (C6 + C7 qy + C8 qz)), λ]
```

$$C0\ \left(C6\ x0 + C4\ x0^2 + C9\ y0 + C7\ x0\ y0 + C3\ y0^2 + C1\ z0 + C8\ x0\ z0 + C5\ y0\ z0 + C2\ z0^2\right) +$$
$$C0\ (C6\ dx + C9\ dy + C1\ dz + 2\ C4\ dx\ x0 + C7\ dy\ x0 + C8\ dz\ x0 +$$
$$C7\ dx\ y0 + 2\ C3\ dy\ y0 + C5\ dz\ y0 + C8\ dx\ z0 + C5\ dy\ z0 + 2\ C2\ dz\ z0)\ \lambda +$$
$$C0\ \left(C4\ dx^2 + C7\ dx\ dy + C3\ dy^2 + C8\ dx\ dz + C5\ dy\ dz + C2\ dz^2\right)\ \lambda^2$$

```
(*denominator of sm3*)
Collect[A9 qx - A8 qy + A7 qz, λ]
```

$$A9\ x0 - A8\ y0 + A7\ z0 + (A9\ dx - A8\ dy + A7\ dz)\ \lambda$$

```
(A9 x0 - A8 y0 + A7 z0) + (A9 dx - A8 dy + A7 dz) λ /. {(A9 dx - A8 dy + A7 dz) → b3}
```

```
A9 x0 - A8 y0 + A7 z0 + b3 λ /. {(A9 x0 - A8 y0 + A7 z0) → b4}
```

$$b4 + b3\ \lambda$$

```
b4 + b3 λ
```

```
C0 (C6 x0 + C4 x0^2 + C9 y0 + C7 x0 y0 + C3 y0^2 + C1 z0 + C8 x0 z0 + C5 y0 z0 + C2 z0^2) +
  C0 (C6 dx + C9 dy + C1 dz + 2 C4 dx x0 + C7 dy x0 + C8 dz x0 +
      C7 dx y0 + 2 C3 dy y0 + C5 dz y0 + C8 dx z0 + C5 dy z0 + 2 C2 dz z0) λ +
  C0 (C4 dx^2 + C7 dx dy + C3 dy^2 + C8 dx dz + C5 dy dz + C2 dz^2) λ^2 /.
 {C0 (C4 dx^2 + C7 dx dy + C3 dy^2 + C8 dx dz + C5 dy dz + C2 dz^2) → b0,
  C0 (C6 dx + C9 dy + C1 dz + 2 C4 dx x0 + C7 dy x0 + C8 dz x0 + C7 dx y0 +
      2 C3 dy y0 + C5 dz y0 + C8 dx z0 + C5 dy z0 + 2 C2 dz z0) → b1,
  C0 (C6 x0 + C4 x0^2 + C9 y0 + C7 x0 y0 + C3 y0^2 + C1 z0 + C8 x0 z0 + C5 y0 z0 + C2 z0^2) → b2}
  (*nominator of sm3*)
```

```
b2 + b1 λ + b0 λ^2
```

```
(* Finally, we have Equation 3.12 in the dissertation*)
```

```
sm3 = (b2 + b1 λ + b0 λ^2) / (b4 + b3 λ)
```

REFERENCES

[1] Oak ridge leadership computing facility. http://www.olcf.ornl.gov/computing-resources/jaguar/.

[2] Hasan Abbasi, Greg Eisenhauer, Matthew Wolf, Karsten Schwan, and Scott Klasky. Just in time: adding value to the io pipelines of high performance applications with JIT-Staging. In *Proceedings of the 20th international symposium on High performance distributed computing*, pages 27–36. ACM, 2011.

[3] Hasan Abbasi, Matthew Wolf, Greg Eisenhauer, Scott Klasky, Karsten Schwan, and Fang Zheng. Datastager: scalable data staging services for petascale applications. *Cluster Computing*, 13(3):277–290, 2010.

[4] David J. Abel and John. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics, and Image Processing*, 24(1):1–13, 1983.

[5] Gerardo Abrugia and Maria Francesca Carfora. Semilagrangian advection on a spherical geodesic grid. *International Journal for Numerical Methods in Fluids*, 55:127–142, 2007.

[6] Radhakrishna Achanta, Appu Shaji, Kevin Smith, Aurelien Lucchi, Pascal Fua, and Sabine Susstrunk. Slic superpixels compared to state-of-the-art superpixel methods. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(11):2274–2282, 2012.

[7] Hussein Aluie and Susan Kurien. Joint downscale fluxes of energy and potential enstrophy in rotating stratified boussinesq flows. *EPL (Europhysics Letters)*, 96(4):44006, 2011.

[8] Janine Bennett, Richard Cook, Nelson Max, Deborah May, and Peter Williams. Parallelizing a high accuracy hardware-assisted volume renderer for meshes with arbitrary polyhedral. In *Proceedings of PVG*, pages 150–156, 2001.

[9] Janine C Bennett, Hasan Abbasi, Peer-Timo Bremer, Ray Grout, Attila Gyulassy, Tong Jin, Scott Klasky, Hemanth Kolla, Manish Parashar, Valerio Pascucci, Philippe Pebay, David Thompson, Hongfeng Yu, Fan Zhang, and Jacqueline Chen. Combining in-situ and in-transit processing to enable extreme-scale scientific analysis. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 49. IEEE Computer Society Press, 2012.

[10] Odemir Martinez Bruno and Luciano da Fontoura Costa. A parallel implementation of exact euclidean distance transform based on exact dilations. *Microprocessors and Microsystems*, 28(3):107–113, 2004.

[11] Chungang Chen, Juzhong Bin, and Feng Xiao. A global multimoment constrained finite-volume scheme for advection transport on the hexagonal geodesic grid. *Monthly Weather Review*, (140):941955., 2012.

[12] Jacqueline H Chen, Alok Choudhary, B De Supinski, M DeVries, ER Hawkes, Scott Klasky, WK Liao, Kwan-Liu Ma, J Mellor-Crummey, Norbert Podhorszki, Ramanan Sankaran, S Shende, and Chun Sang Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Computational Science & Discovery*, 2(1):015001, 2009.

[13] Arthur C Clarke. Extra-terrestrial relays: Can rocket stations give world-wide radio coverage? *Wireless World*, page 306, October 1945.

[14] Carlos D Correa, Robert Hero, and Kwan-Liu Ma. A comparison of gradient estimation methods for volume rendering on unstructured meshes. *Visualization and Computer Graphics, IEEE Transactions on*, 17(3):305–319, 2011.

[15] Jeffrey A Daily, Karen L Schuchardt, and Bruce J Palmer. Efficient extraction of regional subsets from massive climate datasets using parallel IO. In *American Geophysical Union, Fall Meeting 2010*, pages IN41A–1360, 2010.

[16] Ciprian Docan, Manish Parashar, Julian Cummings, and Scott Klasky. Moving the code to the data-dynamic code deployment using activespaces. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 758–769. IEEE, 2011.

[17] Ciprian Docan, Manish Parashar, and Scott Klasky. Dataspaces: An interaction and coordination framework for coupled simulation workflows. volume 15, pages 163–181. Springer, 2012.

[18] Ricardo Fabbri, Luciano Da F. Costa, Julio C. Torelli, and Odemir M. Bruno. 2D Euclidean distance transform algorithms: A comparative survey. *ACM Computing Surveys*, 40(1):2:1–2:44, 2008.

[19] Nathan Fabian, Kenneth Moreland, David Thompson, Andrew C Bauer, Pat Marion, Berk Geveci, Michel Rasquin, and Kenneth E Jan. The paraview coprocessing library: A scalable, general purpose in situ visualization library. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 89–96. IEEE, 2011.

[20] Nathaniel Fout, Kwan-Liu Ma, and James Ahrens. Time-varying, multivariate volume data reduction. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 1224–1230. ACM, 2005.

[21] Irene Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, 1982.

[22] Michael P Garrity. Raytracing irregular volume data. *ACM SIGGRAPH Computer Graphics*, 24(5):35–40, 1990.

[23] Peter R. Gent, Gokhan Danabasoglu, Leo J. Donner, Marika M. Holland, Elizabeth C. Hunke, Steve R. Jayne, David M. Lawrence, Richard B. Neale, Philip J. Rasch, Mariana Vertenstein, Patrick H. Worley, Zong-Liang Yang, , and Minghua Zhang. The community climate system model version 4. *Journal of Climate*, 24(19):4973–4991, 2011.

[24] Stefan Guthe and Wolfgang Straßer. Real-time decompression and visualization of animated volume data. In *Visualization, 2001. VIS'01. Proceedings*, pages 349–572. IEEE, 2001.

[25] Attila Gyulassy, Mark Duchaineau, Vijay Natarajan, Valerio Pascucci, Eduardo Bringa, Andrew Higginbotham, and Bernd Hamann. Topologically clean distance fields. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1432–1439, November 2007.

[26] Kai Hormann and Michael S Floater. Mean value coordinates for arbitrary planar polygons. *ACM Transactions on Graphics (TOG)*, 25(4):1424–1441, 2006.

[27] Tom C. Johnson and Judy M. Vance. The use of the voxmap pointshell method of collision detection in virtual assembly methods planning. In *Proceedings of ASME 2001 Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, 2001.

[28] Mark W. Jones, J. Andreas Bæentzen, and Milos Sramek. 3D distance fields: A survey of techniques and applications. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):581–599, 2006.

[29] Martin Kraus and Thomas Ertl. Cell-projection of cyclic meshes. In *Visualization, 2001. VIS'01. Proceedings*, pages 215–559. IEEE, 2001.

[30] Sriram Lakshminarasimhan, John Jenkins, Isha Arkatkar, Zhenhuan Gong, Hemanth Kolla, Seung-Hoe Ku, Stephane Ethier, Jackie Chen, C. S. Chang, Scott Klasky, Robert Latham, Robert Ross, and Nagiza F. Samatova. ISABELA-QA: Query-driven analytics with ISABELA-compressed extreme-scale scientific data. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 31:1–31:11, 2011.

[31] Daniel Laney, Martin Bertram, Mark Duchaineau, and Nelson Max. Multiresolution distance volumes for progressive surface compression. In *3D Data Processing Visualization and Transmission, 2002. Proceedings. First International Symposium on*, pages 470–479. IEEE, 2002.

[32] Nick Leaf, Venkatram Vishwanath, Joseph Insley, Mark Hereld, Michael Papka, and Kwan-Liu Ma. Efficient parallel volume rendering of large-scale adaptive mesh refinement data. In *Proceedings of the IEEE Symposium on Large-Scale Data Analysis and Visualization*, 2013.

[33] Yu-Hua Lee, Shi-Jinn Horng, and J Seltzer. Parallel computation of the euclidean distance transform on a three-dimensional image array. *Parallel and Distributed Systems, IEEE Transactions on*, 14(3):203–212, 2003.

[34] John M. Levesque, Ramanan Sankaran, and Ray Grout. Hybridizing S3D into an exascale application using OpenACC: An approach for moving to multi-petaflops and beyond. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 15:1–15:11, 2012.

[35] Alex Levinshtein, Adrian Stere, Kiriakos N Kutulakos, David J Fleet, Sven J Dickinson, and Kaleem Siddiqi. Turbopixels: Fast superpixels using geometric flows. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 31(12):2290–2297, 2009.

[36] Dawn Levy. Exascale supercomputing advances climate research. *SciDAC Review*, (16):20–31, 2010.

[37] F. D. Libera and F. Gosen. Using b-trees to solve geographic range queries. *The Computer Journal*, 29(2):176–180, 1986.

[38] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *ACM siggraph computer graphics*, volume 21, pages 163–169. ACM, 1987.

[39] Eric B Lum, Kwan Liu Ma, and John Clyne. Texture hardware assisted rendering of time-varying volume data. In *Proceedings of the conference on Visualization'01*, pages 263–270. IEEE Computer Society, 2001.

[40] Eric B Lum, Brett Wilson, and Kwan-Liu Ma. High-quality lighting and efficient preintegration for volume rendering. In *Proceedings of the Sixth Joint Eurographics-IEEE TCVG conference on Visualization*, pages 25–34. Eurographics Association, 2004.

[41] Kwan-Liu Ma. Large-scale data visualization. *Computer Graphics and Applications, IEEE*, 21(4):22–23, 2001.

[42] Kwan-Liu Ma and Thomas W Crockett. A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. In *Parallel Rendering, 1997. PRS 97. Proceedings. IEEE Symposium on*, pages 95–104. IEEE, 1997.

[43] Kwan-Liu Ma, James S. Painter, Charles D. Hansen, and Michael F. Krogh. Parallel volume rendering using binary-swap compositing. *IEEE Comput. Graph. Appl.*, 14(4):59–68, July 1994.

[44] Detlev Majewski, DöRte Liermann, Peter Prohl, Bodo Ritter, Michael Buchhold, Thomas Hanisch, Gerhard Paul, and Werner Wergen. The operational global icosahedral-hexagonal gridpoint model GME: description and high-resolution tests. *Monthly Weather Review*, (130):319–338, 2002.

[45] Stéphane Marchesin and Guillaume Colin de Verdière. High-quality, semi-analytical volume rendering for amr data. *IEE TVCG*, 15(6):1611–1618, 2009.

[46] MATLAB. MATLAB R2014a Documentation, 2014.

[47] Nelson Max. Optical models for direct volume rendering. *Visualization and Computer Graphics, IEEE Transactions on*, 1(2):99–108, 1995.

[48] Nelson Max, Pat Hanrahan, and Roger Crawfis. Area and volume coherence for efficient visualization of 3d scalar functions. *Computer Graphics*, 24(5):27–33, 1990.

[49] André Maximo, Saulo Ribeiro, Cristiana Bentes, Antonio AF Oliveira, and Ricardo C Farias. Memory efficient gpu-based ray casting for unstructured volume rendering. In *Volume Graphics*, pages 155–162, 2008.

[50] Rashmi Mittal, H. C. Upadhyaya, and Om P. Sharma. Notes and correspondence on near-diffusion-free advection over spherical geodesic grids. *Monthly Weather Review*, (135):4214–4225, 2007.

[51] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Comput. Graph. Appl.*, 14(4):23–32, July 1994.

[52] Kenneth Moreland and Edward Angel. A fast high accuracy volume renderer for unstructured data. In *Proceedings of the IEEE Symposium on Volume Visualization and Graphics*, pages 9–16, 2004.

[53] Kenneth Moreland, Ron Oldfield, Pat Marion, Sebastien Jourdain, Norbert Podhorszki, Venkatram Vishwanath, Nathan Fabian, Ciprian Docan, Manish Parashar, Mark Hereld, et al. Examples of in transit visualization. In *Proceedings of the 2nd international workshop on Petascal data analytics: challenges and opportunities*, pages 1–6. ACM, 2011.

[54] Philipp Muigg, Markus Hadwiger, Helmut Doleisch, and Eduard Gröller. Interactive volume visualization of general polyhedral grids. *IEEE TVCG*, 17(12):2115–2124, 2011.

[55] Philipp Muigg, Markus Hadwiger, Helmut Doleisch, and Helwig Hauser. Scalable hybrid unstructured and structured grid raycasting. *IEEE TVCG*, 13(6):1592–1599, 2007.

[56] Ulrich Neumann. Communication costs for parallel volume-rendering algorithms. *IEEE Comput. Graph. Appl.*, 14(4):49–58, July 1994.

[57] Ulrich Neumann. Communication costs for parallel volume-rendering algorithms. *IEEE Computer Graphics and Applications*, 14(4):49–58, 1994.

[58] Michael B. Nielsen, Ola Nilsson, Andreas Söderström, and Ken Museth. Out-of-core and compressed level set methods. *ACM Trans. Graph.*, 26(4), October 2007.

[59] Bruce Palmer, Annette Koontz, Karen Schuchardt, Ross Heikes, and David Randall. Efficient data io for a parallel global cloud resolving model. *Environmental Modelling & Software*, 26(12):1725–1735, 2011.

[60] Steven Parker, Michael Parker, Yarden Livnat, Peter-Pike Sloan, Charles Hansen, and Peter Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(3):238–250, July 1999.

[61] Tom Peterka, David Goodell, Robert Ross, Han-Wei Shen, and Rajeev Thakur. A configurable algorithm for parallel image-compositing applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 4:1–4:10, 2009.

[62] Tom Peterka, Robert B. Ross, Han-Wei Shen, Kwan-Liu Ma, Wes Kendall, and Hongfeng Yu. Parallel visualization on leadership computing resources. In *Journal of Physics: Conference Series*, volume 180, page 012088. IOP Publishing, 2009.

[63] David A. Randall, Todd D. Ringler, Ross P. Heikes, Phil Jones, and John Baumgardner. Climate modeling with spherical geodesic grids. *Computing in Science & Engineering*, 4(5):32–41, 2002.

[64] Todd Ringler, Mark Petersen, Robert L Higdon, Doug Jacobsen, Philip W Jones, and Mathew Maltrud. A multi-resolution approach to global ocean modeling. *Ocean Modelling*, 69:211–232, 2013.

[65] Stefan Röttger and Thomas Ertl. A two-step approach for interactive pre-integrated volume rendering of unstructured grids. In *Proceedings of the IEEE Symposium on Volume Visualization and Graphics*, pages 23–28, 2002.

[66] Stefan Röttger, Martin Kraus, and Thomas Ertl. Hardwareaccelerated volume and isosurface rendering based on cell projection. In *Proceedings of IEEE Visualization*, pages 109–116, 2000.

[67] Robert Sadourny, Akio Arakawa, and Yale Mintz. Integration of the nondivergent barotropic vorticity equation with an icosahedral-hexagonal grid for the sphere 1. *Monthly Weather Review*, 96(6):351–356, 1968.

[68] Hanan Samet. Distance transform for images represented by quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 4(3):298–303, 1982.

[69] Jens Schneider and Rüdiger Westermann. Compression domain volume rendering. In *Visualization, 2003. VIS 2003. IEEE*, pages 293–300. IEEE, 2003.

[70] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *Proceedings of VecPar*, pages 1–25, 2010.

[71] Han-Wei Shen, Ling-Jen Chiang, and Kwan-Liu Ma. A fast volume rendering algorithm for time-varying fields using a time-space partitioning (TSP) tree. In *Proceedings of the conference on Visualization'99: celebrating ten years*, pages 371–377. IEEE Computer Society Press, 1999.

[72] Han-Wei Shen and Christopher R Johnson. Differential volume rendering: A fast volume visualization technique for flow animation. In *Proceedings of the conference on Visualization'94*, pages 180–187. IEEE Computer Society Press, 1994.

[73] Peter Shirley and Allan Tuchman. *A polygonal approximation to direct scalar volume rendering*, volume 24. ACM, 1990.

[74] Bong-Soo Sohn, Chandrajit Bajaj, and Vinay Siddavanahalli. Feature based volumetric video compression for interactive playback. In *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 89–96. IEEE Press, 2002.

[75] Clifford M. Stein, Barry G. Becker, and Nelson Max. Sorting and hardware assisted rendering for volume visualization. In *Proceedings of VolVis*, pages 83–89, 1994.

[76] John Strain. Fast tree-based redistancing for level set computations. *Journal of Computational Physics*, 152(2):664–686, 1999.

[77] Waldo Tobler and Zi-tan Chen. A quadtree for global information storage. *Geographical Analysis*, 18(4):360–371, 1986.

[78] Tiankai Tu, Hongfeng Yu, Leonardo Ramirez-Guzman, Jacobo Bielak, Omar Ghattas, Kwan-Liu Ma, and David R. O'Hallaron. From mesh generation to scientific visualization: an end-to-end approach to parallel supercomputing. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, 2006.

[79] Andrea Vedaldi and Stefano Soatto. Quick shift and kernel methods for mode seeking. In *Computer Vision–ECCV 2008*, pages 705–718. Springer, 2008.

[80] Venkatram Vishwanath, Mark Hereld, and Michael E Papka. Toward simulation-time data analysis and i/o acceleration on leadership-class systems. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*. IEEE, 2011.

[81] Chaoli Wang, Hongfeng Yu, and Kwan-Liu Ma. Application-driven compression for visualizing large-scale time-varying data. *IEEE Computer Graphics and Applications*, 30:59–69, 2010.

[82] Pan Wang, Zhiquan Cheng, Ralph Martin, Huahai Liu, Xun Cai, and Sikun Li. Numa-aware image compositing on multi-gpu platform. *The Visual Computer*, 29(6-8):639–649, 2013.

[83] Zachary Wartell, William Ribarsky, and Larry Hodges. Efficient ray intersection for visualization and navigation of global terrain using spheroidal height-augmented quadtrees. In *Data Visualization '99*, Eurographics, pages 213–223, 1999.

[84] Jishang Wei, Hongfeng Yu, Ray W Grout, Jacqueline H Chen, and Kwan-Liu Ma. Visual analysis of particle behaviors to understand combustion simulations. *Computer Graphics and Applications, IEEE*, 32(1):22–33, 2012.

[85] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *Proceedings of IEEE Visualization*, pages 333–340, 2003.

[86] Manfred Weiler, Martin Kraus, Markus Merz, and Thomas Ertl. Hardware-based view-independent cell projection. *IEEE TVCG*, 9(2):163–175, 2003.

[87] Manfred Weiler, Paula N Mallón, Martin Kraus, and Thomas Ertl. Texture-encoded tetrahedral strips. In *Volume Visualization and Graphics, 2004 IEEE Symposium on*, pages 71–78. IEEE, 2004.

[88] Hilary Weller, Henry G. Weller, and Aimé Fournier. Voronoi, delaunay, and block-structured mesh refinement for solution of the shallow-water equations on the sphere. *Monthly Weather Review*, (137):4208–4224, 2009.

[89] Wikipedia. Chebyshev distance — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Chebyshev_distance, 2015. [Online; accessed 09-Jan-2015].

[90] Wikipedia. Taxicab geometry — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Taxicab_geometry, 2015.

[91] Wikipedia. Visualization (computer graphics) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Visualization_(computer_graphics), 2016. [Online; accessed 01-Feb-2016].

[92] Jane Wilhelms and Allen Van Gelder. Octrees for faster isosurface generation. *ACM Trans. Graph.*, 11(3):201–227, 1992.

[93] Jane Wilhelms and Allen Van Gelder. Multi-dimensional trees for controlled volume rendering and compression. In *Proceedings of the 1994 symposium on Volume visualization*, pages 27–34. ACM, 1994.

[94] Peter L. Williams. Visibility-ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, 1992.

[95] David L. Williamson. Integration of the barotropic vorticity equation on a spherical geodesic grid. *Tellus*, (20):642–653, 1968.

[96] Jinrong Xie, Franz Sauer, and Kwan-Liu Ma. Fast uncertainty-driven large-scale volume feature extraction on desktop PCs. In *Large Data Analysis and Visualization (LDAV), 2015 IEEE 5th Symposium on*, pages 17–24. IEEE, 2015.

[97] Jinrong Xie, Hongfeng Yu, and Kwan-Liu Ma. Interactive ray casting of geodesic grids. In *Computer Graphics Forum*, volume 32, pages 481–490. Wiley Online Library, 2013.

[98] Jinrong Xie, Hongfeng Yu, and Kwan-Liu Ma. Visualizing large 3d geodesic grid data with massively distributed GPUs. In *Large Data Analysis and Visualization (LDAV), 2014 IEEE 4th Symposium on*, pages 3–10. IEEE, 2014.

[99] Hongfeng Yu, Kwan-Liu Ma, and Joel Welling. A parallel visualization pipeline for terascale earthquake simulations. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, 2004.

[100] Hongfeng Yu, Chaoli Wang, Ray W Grout, Jacqueline H Chen, and Kwan-Liu Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, (3):45–57, 2010.

[101] Hongfeng Yu, Chaoli Wang, Ray W. Grout, Jacqueline H. Chen, and Kwan-Liu Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, 30(3):45–57, 2010.

[102] Hongfeng Yu, Chaoli Wang, and Kwan-Liu Ma. Massively parallel volume rendering using 2-3 swap image compositing. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 48:1–48:11, 2008.

[103] Hongfeng Yu, Jinrong Xie, Kwan-Liu Ma, Hemanth Kolla, and Jacqueline Chen. Scalable parallel distance field construction for large-scale applications.

[104] Fang Zheng, Hasan Abbasi, Ciprian Docan, Jay Lofstead, Qing Liu, Scott Klasky, Manish Parashar, Norbert Podhorszki, Karsten Schwan, and Matthew Wolf. Predata–preparatory data analytics on peta-scale machines. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.