Scalable Visualization and Interactive Analysis using Massive Data Streams

Valerio Pascucci[1, 3], Peer-Timo Bremer[1, 2], Attila Gyulassy[1], Giorgio Scorzelli[1], Cameron Christensen[1], Brian Summa[1] and Sidharth Kumar[1],

[1]*University of Utah*
[2]*Lawrence Livermore National Laboratory*
[3]*Pacific Northwest National Laboratory*

Abstract.
Historically, data creation and storage has always outpaced the infrastructure for its movement and utilization. This trend is increasing now more than ever, with the ever growing size of scientific simulations, increased resolution of sensors, and large mosaic images. Effective exploration of massive scientific models demands the combination of data management, analysis, and visualization techniques, working together in an interactive setting. The ViSUS application framework has been designed as an environment that allows the interactive exploration and analysis of massive scientific models in a cache-oblivious, hardware-agnostic manner, enabling processing and visualization of possibly geographically distributed data using many kinds of devices and platforms.

For general purpose feature segmentation and exploration we discuss a new paradigm based on topological analysis. This approach enables the extraction of summaries of features present in the data through abstract models that are orders of magnitude smaller than the raw data, providing enough information to support general queries and perform a wide range of analyses without access to the original data.

Keywords. Visualization, data analysis, topological data analysis, Parallel I/O

1. Introduction

In this supercomputing era, with the continued surge of computing resources, scientists are simulating newer and more complex phenomena. These simulations often generate enormously large datasets that are exceedingly difficult to manage and hence require more advanced data management, analysis and visualization techniques to gain new scientific insights. Developing such techniques involves addressing a number of major challenges, such as the real-time management of massive data and the quantitative analysis of scientific features of unprecedented complexity.

In this chapter we focus on two key components that aid the visualization and analysis process:

1. the ViSUS software framework, designed with the primary philosophy that the visualization of massive data need not be tied to specialized hardware or infrastructure, and

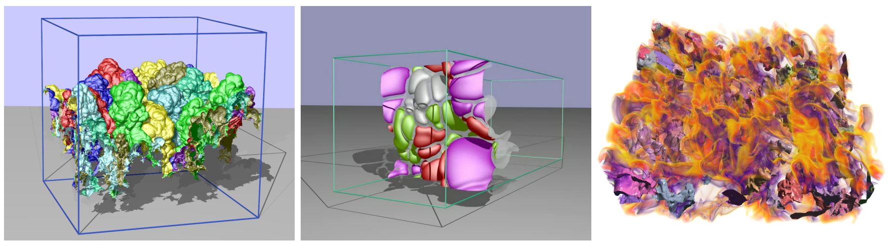


Figure 1. Examples of simulations producing routinely massive amounts of data. Rayleigh-Taylor hydrodynamic instabilities simulate mixing fluids (left). A combustion simulation of low-swirl hydrogen flames (middle). Simulation of non-premixed hydrogen flames with extinction and re-ignition phenomena (right).

2. robust topological analysis techniques that provide the ability to detect and quantify features in data at multiple scales.

ViSUS is a scalable data analysis and visualization framework for processing large scale scientific data with high performance selective queries. It combines an efficient data model with progressive streaming techniques to allow interactive processing rates on a variety of computing devices ranging from handheld devices, like an iPhone, to simple workstations, to the I/O of parallel supercomputers. In other words, ViSUS provides a visualization framework for large data which is designed to be lightweight, highly scalable and run on a wide range of platforms or hardware. The ViSUS infrastructure is designed to support a wide variety of applications all from the same code base. In section 2 we describe the framework and demonstrate a wide range of applications supported by ViSUS (also illustrated in Figure 7).

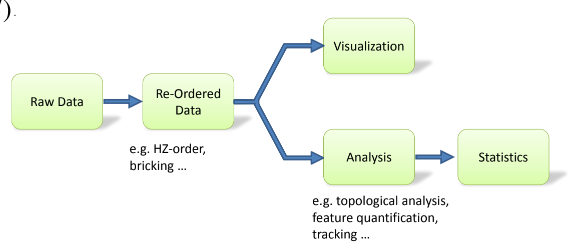

Raw Data
Re-Ordered Data
e.g. HZ-order, bricking ...
Visualization
Analysis
Statistics
e.g. topological analysis, feature quantification, tracking ...

Figure 2.: Hierarchical z-ordered data layout (integral part of ViSUS) and topological analysis components in the context of an advanced interactive visualization and analysis pipeline.

We also present a complementary approach, where the data is first reduced using representations that maintain the important features for subsequent analysis. Topological techniques based on Morse theory are useful in this scenario, since they represent a large feature space, can be computed robustly, and inherently encode features at multiple scales. Due to the combinatorial nature of this framework, we can implement the core constructs of Morse theory without the approximations and instabilities of classical numerical techniques. The inherent robustness of the combinatorial algorithms allows us to address the high complexity of the feature extraction problem for high resolution scientific data. Our approach has enabled the successful quantitative analysis (see Figure 1) of several massively parallel simulations including the study of turbulent hydrodynamic instabilities, porous material under stress and failure, the energy transport of eddies in ocean data used for climate modeling, and lifted flames that lead to clean energy production.

To provide context, we highlight these two components in a typical visualization and analysis pipeline in Figure 2. We assume that raw data from simulations is available

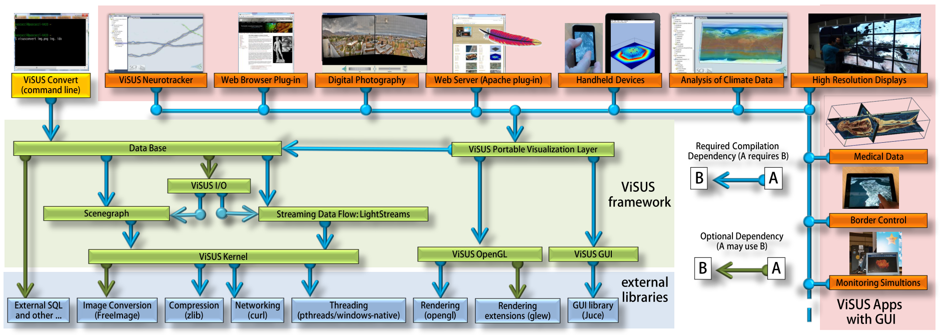


Figure 3. The ViSUS application framework. Arrows denote external and internal dependencies of the main software components. Additionally we show the relationship with several applications that have been successfully developed using this framework.

as real-valued, regular samples of space-time. Due to the large size of datasets, we emphasize that the data samples cannot all be loaded into main memory. As a result it is not feasible to use standard implementations of visualization and analysis algorithms on commodity hardware.

2. ViSUS

This section is devoted to the description of the scalability principles that are the basis of the ViSUS application framework and how they can be used in practical applications, both in scientific visualization and other domains such as exploration of geospatial models or digital photography. The components can be grouped into three major categories.

1. A lightweight and fast out-of-core data management framework using multi-resolution space filling curves. This allows the organization of information in an order that exploits the cache hierarchies of any modern data storage architectures and delivers the most pertinent information to the user sooner.
2. A dataflow framework that allows data to be processed during movement. Processing massive datasets in their entirety is a long and expensive operation that hinders interactive exploration. By designing new streaming algorithms that function within the ViSUS framework, data can be processed on-the-fly.
3. A portable visualization layer which was designed to scale from mobile devices to powerwall displays using the same code base.

2.1. ViSUS Software Architecture

In this section we will detail the three major components of ViSUS and how they are used to achieve a fast, scalable, and highly portable data processing and visualization environment. Figure 3 provides a diagram of the architecture of the ViSUS application framework.

Data Access Layer The ViSUS data access layer is a key component allowing immediate, efficient data pipeline processing that otherwise would be stalled by traditional

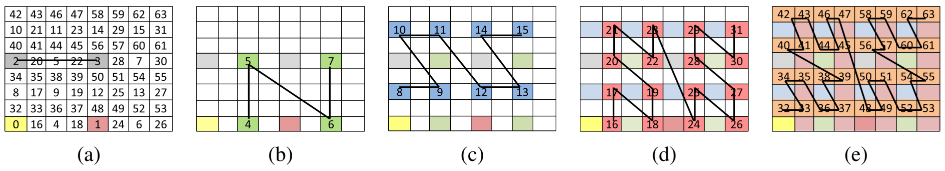


Figure 4. HZ ordering (indicated by box labels and curves) at various levels for an 8x8 dataset. (a) levels 0, 1, and 2. (b)-(e) levels 3-6, respectively.

system I/O cost. The ViSUS I/O component (and its generalized database component), in particular are focused on enabling the effective deployment of out-of-core and data streaming algorithms. Out-of-core computing [24] specifically addresses the issues of data layout restructuring and algorithm redesign. These are necessary to achieve data access patterns having minimal performance degradation with external memory storage. Algorithmic approaches in this area also yield valuable techniques for parallel and distributed computing. In this environment, one must typically address the similar issues of balancing processing time with the time required for data access and movement amongst elements of a distributed or parallel application.

The solution to the out-of-core processing problem is typically divided into two parts: (1) algorithm analysis, to understand data access patterns and, when possible, redesign to maximize data locality; (2) storage of data in secondary memory using a layout consistent with the access patterns of the algorithm, amortizing the cost of individual I/O operations over several memory access operations.

To achieve real-time rates for visualization and/or analysis of extreme scale data, one would commonly seek some form of adaptive level of detail and/or data streaming. By traversing simulation data hierarchically from the coarse to the fine resolutions and progressively updating output data structures derived from this data, one can provide a framework that allows for real-time access of the simulation data that performs well, even for extreme scale data. Many of the interaction parameters, such as display viewpoint, are determined at run time by users and therefore precomputing these levels of detail optimized for specific queries is infeasible. Therefore, to maintain efficiency, a storage data layout must satisfy two general requirements: (i) the input hierarchy is traversed from coarse to fine and level by level, so that data in the same level of resolution is accessed at the same time, and (ii) within each resolution level, the regions that are in close geometric proximity are stored as much as possible in close memory locations and also traversed at the same time.

Space filling curves [22] have been used successfully to develop a static indexing scheme that generates a data layout satisfying both of the above requirements for hierarchical traversal (Figure 4). The data access layer of ViSUS employs a hierarchical variant of a Lebesgue space filling curve [17]. The data layout of this curve is commonly referred to as *HZ order* in the literature. This data access layer has three key features that make it particularly attractive. First, the order of the data is independent of the out-of-core block structure, so that its use in different settings (e.g. local disk access or transmission over a network) does not require any large data reorganization. Second, conversion from the Z-order indexing [12] used in classical database approaches to the ViSUS HZ-order indexing scheme can be implemented with a simple sequence of bit-string manipulations. Third, since there is no data replication, ViSUS avoids the performance penalties asso-

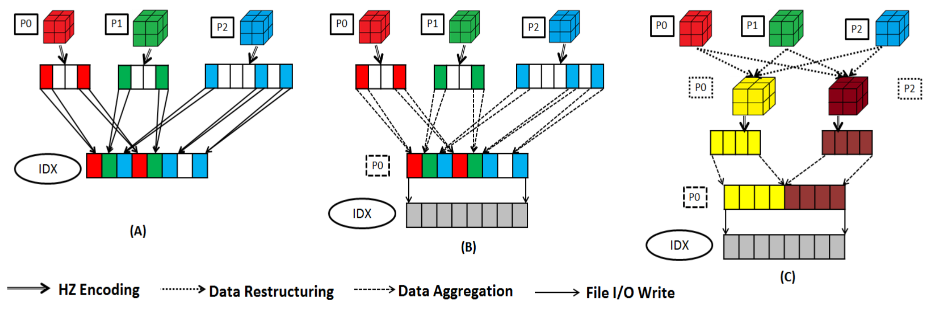


Figure 5. Parallel I/O strategies: (a) Naive approach where each process writes its data in the same file, (b) alternative approach where contiguous data segments are passed to an intermediate aggregator that writes to disk, (c) Three-Phase I/O : Data is first restructured, maintaining the multi-dimensional layout, then aggregated and written to the disk.

ciated with guaranteeing consistency especially for dynamic updates, and does not incur the increased storage requirements associated with most other hierarchical and out-of-core schemes.

Beyond the theoretical interest in developing hierarchical indexing schemes for *n*-dimensional space filling curves, ViSUS targets practical applications in out-of-core data analysis and visualization and has been successfully used for direct streaming and real-time remote monitoring of large scale simulations during their executions on IBM BG/L supercomputers at LLNL [16] as well as on Hopper supercomputer at NERSC [11]. The multi-resolution data model used in ViSUS allows adjusting the quality of the visualization depending on the communication speed and on the performance of the local workstation. Owing to the extremely scalable nature of this approach, the same code base is used for a large set of applications while exploiting a wide range of available devices from large power-wall displays to workstations, laptop computers or handheld devices such as the iPhone. For details on related work, algorithmic analysis, and experimental results see [17].

Parallel I/O for Large Scale Simulations The multi-resolution data layout of ViSUS discussed above is a progressive, linear format and therefore has a write routine that is inherently serial. During the execution of large scale simulations, it would be ideal for each node in the simulation to be able to write its piece of the domain data directly into this layout. Therefore a parallel write strategy must be employed. Figure 5 illustrates different possible parallel strategies that have been considered. As shown in Figure 5 (A), each process can naively write its own data directly to the proper location in a unique underlying binary file. This is inefficient due to the discontinuous access of sparse buffers within memory as well as the large number of small granular, concurrent accesses to the same file. Moreover, as the data gets large, it becomes disadvantageous to store the entire dataset as a single large file and typically the entire dataset is partitioned into a series of smaller more manageable pieces. This disjointness can be an advantage to a parallel write routine. As each simulation process produces a portion of the data, it can store its piece of the overall dataset locally and pass the data on to an aggregator process.

These aggregator processes can be used to gather the individual pieces and composite the entire dataset. Figure 5 (B) shows this strategy, where each process transmits a contiguous data segment to an intermediate aggregator. Once the aggregator's buffer is filled, the data is written to disk using a single large I/O operation. This method still suffered due to discontinuous memory access owing to sparse data buffers. Therefore we adopted a three-phase I/O (Figure 5 (C)) technique to solve this problem by first restructuring the data in their multi-dimensional layout forming dense and contiguous memory buffers, then following with data aggregation and writes to the disk. This strategy has been shown to exhibit good throughput performance and weak scaling for S3D combustion simulation applications when compared to the standard Fortran I/O benchmark (on Hopper) [11]. S3D weak scaling results for Hopper supercomputer are shown in Figure 6 (Top). In each run, S3D writes out 20 time-steps wherein each process contributes a 30^3 block of double-precision data consisting of 4 variables; pressure, temperature, velocity (3 samples), and species (11 samples). we varied the number of processes from 1,024 to 65,536, thus varying the amount of data generated per time-step from 3.29 GiB to 210.93 GiB. From Figure 6 we see, compared with other file formats PnetCDF and Fortran I/O, our parallel implementation (Parallel ViSUS) appears to lag in the lower process count ranges ($\leq$ 8192); but as the number of processes increases it outperforms the other file formats. This lag in performance for the lower process counts can be attributed to lack in aggregators, which can be adjusted by varying some tunable parameters. Comparing performance numbers, we see at process count 65,536 our implementation achieves a throughput of around 19.84 GiB/sec which is approximately three times that of Fortran I/O (7.2 GiB/sec) and PnetCDF (6.5 GiB/sec). Similar S3D weak scaling results for Intrepid BG/P supercomputer are shown in Figure 6 (Bottom). At the time of these experiments, the Intrepid file system was nearly full (95% capacity) which is believed to have seriously degraded I/O performance for all experiments. While each of the output methods showed scaling, none of the output methods approached the expected throughput of Intrepid at scale. As evident from the figure performance of our parallel file format is more than other file formats. Here, Fortran I/O fails to scale on Intrepid because of high degree of metadata contention and serialization involved in creating all the file in the same subdirectory. Recent

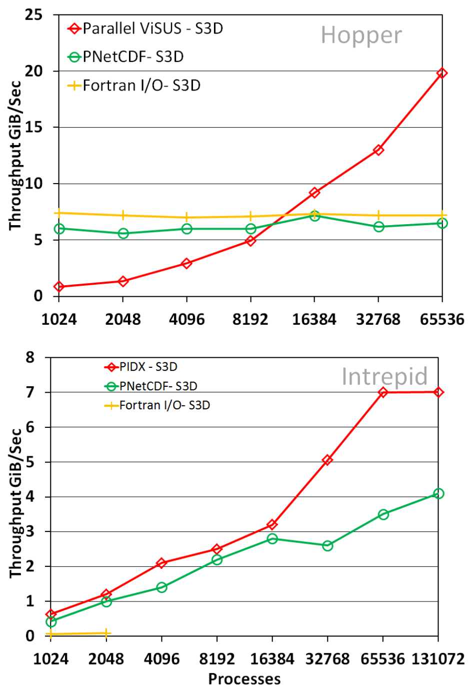


Figure 6.: Hopper (Top) and Intrepid (Bottom) results for weak scaling (per-process block size 30^3) of different I/O mechanisms: Parallel ViSUS, Fortran I/O, and PnetCDF with S3D.

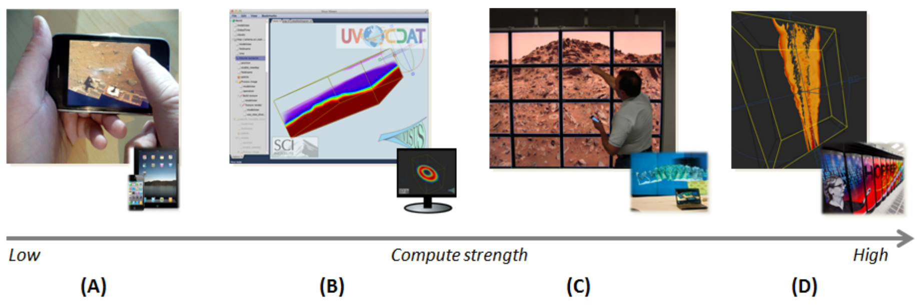


Figure 7. A wide range of platform and hardware supported by ViSUS ranging from (A) Hand-held devices (B) Desktops/Laptops (C)Clusters/power-walls, (D) supercomputers (parallel implementation of ViSUS). Also shown here the same application and visualization of a Mars panorama running on an iPhone mobile device (A) and a powerwall display (C) (data courtesy of NASA).

results[1] have shown empirically how this strategy scales well for a large number of nodes while enabling real-time monitoring of high resolution simulations (see Section 2.2).

Portable Visualization Layer - ViSUS AppKit. The visualization component of ViSUS was built with the philosophy that a single code base can be designed to run on a variety of platforms and hardware ranging from mobile devices to powerwall displays. To enable this portability, the the basic rendering routines were designed to be OpenGL ES compatible. This is a limited subset of OpenGL used primarily for mobile devices. More advanced rendering routines can be enabled if hardware support is available. In this way, the data visualization can scale in quality depending on the available hardware. Beyond the display of the data, the underlying GUI library can hinder portability to multiple devices. Therefore ViSUS AppKit provides an abstract GUI interface that currently supports both the Qt [2] and Juce [3] libraries providing lightweight support for mobile platforms such as iOS and Android in addition to the major desktop operating systems. ViSUS provides a generic viewer which contains standard visualizations such as slicing, volume rendering and isosurfacing. Similarly to the example LightStream modules, these routines can be expanded through a well-defined API. Additionally, the base system can display 2D and 3D time- varying data. As mentioned above, each of these visualizations can operate on the end result of a LightStream dataflow. The system considers a 2D dataset as a special case of a slice renderer and therefore the same code base is used to render both 2D and 3D data. The above design decisions allow the same code base to be used on multiple platforms seamlessly for data of arbitrary dimensions. Figure 7 shows a wide variety of platform that ViSUS supports.

ViSUS Dataflow.

Even simple manipulations can be very expensive when applied to each variable in a large scale dataset. Instead, it would be ideal to process the data based on need by pushing data through a processing pipeline as the user interacts with different portions of the data. With the ViSUS multi-resolution layout different regions of the data can be efficiently accessed at varying resolutions. Therefore, different compute modules can

[1]Execution on the Hopper system at NERSC and BG/P intrepid system at Argonne National Lab.
[2]http://qt.digia.com
[3]http://www.rawmaterialsoftware.com

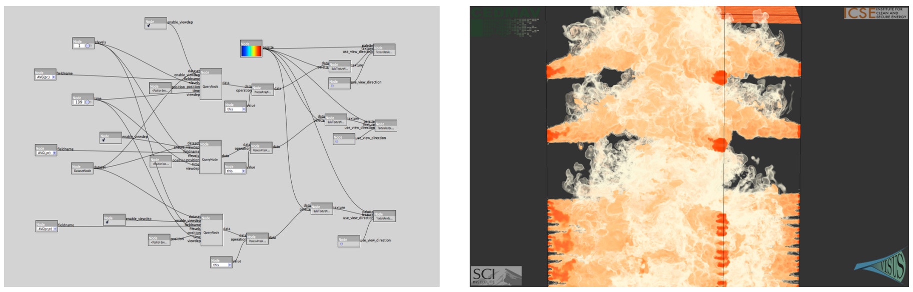


Figure 8. The ViSUS Dataflow used for analysis and visualization of a 3D combustion simulation. (left) Several dataflow modules chained together to provide a light and flexible stream processing capability. (right) One visualization that is the result from this dataflow.

be implemented using progressive algorithms to operate on this data stream. Operations such as binning, clustering, or rescaling are trivial to implement on this hierarchy given some known statistics on the data, such as the function value range, etc. These operators can be applied to the data stream as-is while the data is moving to the user, progressively refining the operation as more data arrives. More complex operations can also be reformulated to work well using the hierarchy. For instance, using the direct ViSUS layout for 2-dimensional image data produces a hierarchy which is identical to a sub-sampled image pyramid on the data. Moreover, as data is requested progressively, the transfer will traverse this pyramid in a coarse-to-fine manner. Techniques such as gradient-domain image editing can be reformulated to use this progressive stream and produce visually acceptable solutions [23,18,19]. These adaptive, progressive solutions allow the user to explore a full resolution solution as if it were fully available, without the expense of the full computation.

The ViSUS Dataflow facilitates this stream processing model by providing definable modules within a dataflow framework with a well understood API. Figure 8 gives an example of a dataflow for the analysis and visualization of a scientific simulation. This particular example is the dataflow for a Uintah combustion simulation used by the Institute for Clean and Secure Energy (ICSE) at the University of Utah. Each of the dataflow module provides streaming capability through input and output data ports which can be used in a variety of data transfer/sharing modes. In this way, groups of modules can be chained to provide complex processing operations as the data is transferred from the initial source to the final data analysis and visualization stages. User demands and interactions, typically drive this data flow. A variety of "standard" modules, such as data differencing (for change detection), content based image clustering (for feature detection), volume rendering and simple topological analysis, are part of the base system. These can be used as templates by new developers for their own progressive streaming data processing modules.

2.2. *Applications*

The upper and right portions of the infrastructure diagram in Figure 3 show that ViSUS has the versatility to be used in a wide range of applications. Below we highlight a representative subset of these applications. The general philosophy behind ViSUS applications is the development of task-driven tools on top of a robust cache-oblivious, operating-system-independent streaming framework.

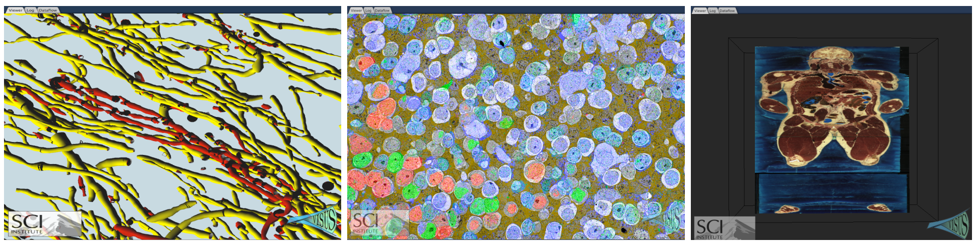


Figure 9. The ViSUS application framework visualizing and processing medical imagery. (left) The NeuroTracker application providing the segmentation of neurons from extremely high resolution Confocal Fluorescence Microscopy brain imagery. This data courtesy of the Center for Integrated Neuroscience and Human Behavior at the Brain Institute, University of Utah. (middle) An application for the interactive exploration of an electron microscopy image of a slice of a rabbit retina. This dataset is courtesy of the MarcLab at the University of Utah. (right) A 3D slicing example using the Visible Male dataset.

NeuroTracker and Other Medical Applications.

Built on the ViSUS framework, NeuroTracker is an application that targets the segmentation of neurons from extremely high resolution Confocal Fluorescence Microscopy brain imagery [4]. NeuroTracker's core data processing uses the ViSUS I/O library to get fast access to brain imaging data combined with multi-resolution topological analysis used to seed the segmentation of neurons. The Voxelscooping image segmentation algorithm [21] is used to extract filamentary structures from the topological seeds. NeuroTracker is based on the ViSUS application framework including ViSUS streaming dataflow and GUI components.

Figure 9 also highlights two additional medical imaging applications. In Figure 9 (center), an example of the ViSUS framework is used for interactive exploration of an electron microscopy image of a slice of a rabbit retina. In all, this 3D dataset composed of 141 slices is over 5.5 terapixels in size[5]. Figure 9 (right) is a 3D data slicing example of the Visible Male[6] dataset comprised of over 4.6 billion color voxels.

Web-server and plug-in In addition to the traditional viewer, ViSUS has been extended to support a client-server model. The ViSUS server can be used as a standalone application or a web server plugin module. The ViSUS server uses HTTP (a stateless protocol) in order to support many clients. A traditional client/server infrastructure, where the client establishes and maintains a stable connection to the server, can only handle a limited number of clients robustly. Using HTTP, the ViSUS server can scale to thousands of connections. The ViSUS client keeps a number (normally 48) of connections alive in a pool using the "keep-alive" option of HTTP. The use of lossy or lossless compression is configurable by the user. For example, ViSUS supports JPEG and EXR for lossy compression of byte and float data respectively. The ViSUS server is an open client/server architecture. Therefore it is possible to port the plugin to any web server which supports a C++ module (i.e. apache, IIS). The ViSUS client can be enabled to cache data to local memory or to disk. In this way, a client can minimize transfer time by referencing data already sent, as well as having the ability to work offline if the server becomes un-

[4]Data is courtesy of the Center for Integrated Neuroscience and Human Behavior at the Brain Institute, University of Utah (http://brain.utah.edu/)

[5]Data is courtesy of the MarcLab at the University of Utah (http://prometheus.med.utah.edu/~marclab/)

[6]http://www.nlm.nih.gov/research/visible/visible_human.html

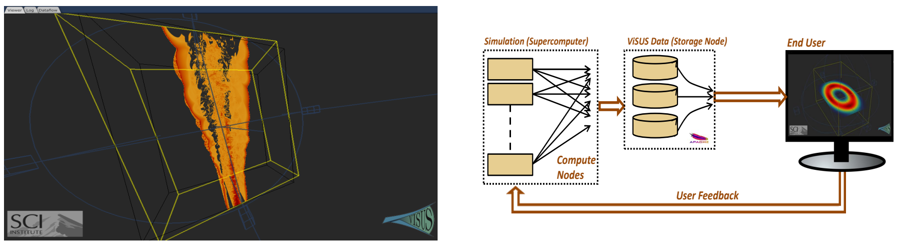


Figure 10. (left) Remote visualization and monitoring of simulations with ViSUS. An S3D combustion simulation visualized from a desktop in the SCI Institute (Salt Lake City, Utah) during its execution on the HOPPER 2 high performance computing platform in Lawrence Berkeley National Laboratory (Berkeley, California). (right) A generic visualization pipeline for real-time remote monitoring of simulation data from supercomputers using ViSUS.

reachable. The ViSUS portable visualization framework (Appkit) also has the ability to be compiled as a Google Chrome, Microsoft Internet Explorer, or Mozilla Firefox web browser plugin. This allows a ViSUS framework based viewer to be easily integrated into web visualization portals.

Real-Time Simulation Monitoring

Ideally, a user-scientist would like to view a simulation as it is computed in order to steer or correct the simulation as unforeseen events arise. Simulation data is often very large. For instance, a single field of a time-step from the S3D combustion simulation in Figure 10 (left) is approximately 128 GB in size. In the time needed to transfer this single time-step, the user-scientist would have lost any chance for significant steering/correction of an ongoing simulation or at least take the opportunity to save computing resources by early termination of a job that is not useful anymore. By using the parallel ViSUS data format in simulation checkpointing [9,10,11], we can link this data directly with an Apache server using a ViSUS plug-in running on a node of the cluster system. By doing this, user-scientists can visualize simulation data as checkpoints are reached. ViSUS can handle missing or partial data, therefore the data can be visualized even as it is being written to disk by the system.

ViSUS's support for a wide-variety of clients (a cross-platform standalone application, a web browser plug-in, or an iOS application for the iPad or iPhone) allows the application scientist to monitor a simulation as it is produced from practically any system that is available without any need to transfer the data off the computing cluster. As mentioned above, Figure 10 (left) is an S3D large-scale combustion simulation visualized remotely from an HPC platform[7]. We have used this approach for direct streaming and real-time remote monitoring of early large-scale simulations. Figure 10 (right) illustrates the generic visualization pipeline deployed to achieve real-time remote monitoring of large scale simulation data using ViSUS. This work continues its evolution towards the deployment of high performance tools for in-situ and post-processing data management and analysis for the software and hardware resources of the future including exascale DOE platforms of the next decade.

Remote Climate Analysis and Visualization The ViSUS application framework has been used in the climate modeling community to visualize climate change simulations

[7]Data is courtesy of Jackie Chen at Sandia National Laboratories, Combustion Research Facility

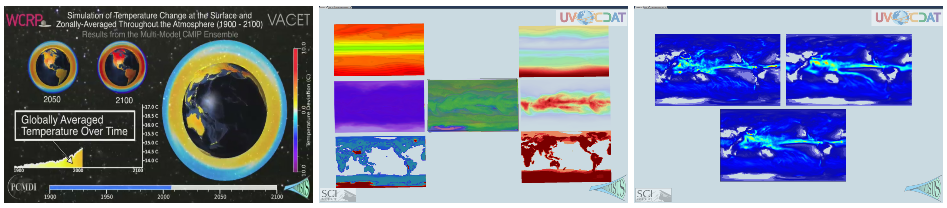


Figure 11. Climate visualization with ViSUS.(left) The ViSUS framework providing a visualization for a temperature change ensemble simulation for the Earth's surface for the December 2009 climate summit meeting in Copenhagen. (center) Two datasets of different spatial resolution showing global surface precipitation. They are dynamically resampled and blended with ViSUS using an arbitrary user-specified operation (average). (right) Climate simulation showing global temperature and precipitation from sixteen different climate models of differing spatial resolution, dynamically resampled and blended with ViSUS using an arbitrary user-specified operation (standard deviation).

comprising many species over a large number of time steps. This type of data provides the opportunity to both build high quality data analysis routines and challenge the performance of the data management infrastructure. Figure 11 (left) shows ViSUS rendering 10TB of data used to present findings regarding the Earth's temperature change based on historical and projected simulation data at the December 2009 climate summit meeting in Copenhagen (Denmark). This work showed the possibility of transferring, transforming, analyzing, and rendering a large dataset on geographically distributed computing resources [8]. Figure 11 (center) we see the ViSUS framework providing a visualization of global precipitation that dynamically resamples datasets of different spatial dimensions and blends them accoring to a user-specified operation (average). Figure 11 (right) the ViSUS framework providing visualization of climate data showing global temperature and precipitation from sixteen different climate models of differing spatial resolution, dynamically resampled and blended according to a user-specified operation (standard deviation). The ability to dynamically combine heterogenous datasets can help bring novel insights into the dynamics of global carbon cycle, atmospheric chemistry, land and ocean ecological processes and their coupling with climate.

Panorama Multiscale Processing and Viewer The ViSUS framework along with the Lightstream Dataflows can be used for real-time, large panorama processing and visualization. Figure 12 (left) provides a visualization of the original picture data for a panorama of Salt lake City, which is comprised of over 600 individual images for a total image mosaic size of 3.2 gigapixel. As mentioned in Section 2.1, the Lightstream Dataflow can be used to operate on the multi-resolution data as the panorama is being viewed and provide an approximate gradient domain solution [23, 18,19] for each viewpoint. In this way, a user can explore the panorama as if the full gradient domain solution was available without it ever being computed in full. This preview is shown in Figure 12 (right).

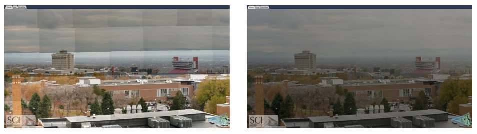


Figure 12.: (left) The color shift between images in a panorama mosaic. (right) An application using a Lightstream dataflow to provide approximate gradient domain solution as a user interacts with the data.

3. Streaming Feature-Based Analysis for Exploration of Large Scale Turbulent Combustion

Traditional techniques for data analysis typically compute global statistics, or focus on a set of features determined by particular parameters or thresholds. With the increasing resolution of simulations, scientists are becoming more interested in relatively smaller scale phenomena. Instead of concentrating only on global statistical analysis or visualization, the focus is often on highly localized *features* corresponding to some region of interest, e.g. material core lines [7], extinction regions [13], or burning cells [4]. Traditionally, such features are analyzed by extracting them in an (often costly) post-processing step and subsequently characterizing them through various statistics, e.g. number of features, size, and/or conditional statistics. However, the feature definition typically depends on multiple parameters and thresholds which depending on the application area are picked more or less *ad hoc*. This introduces problems in the analysis, as the outcomes might depend heavily on the choice of input parameters, yet few input parameters are ever explored due to computational costs. This problem becomes even more pronounced as data is increasing in size, since the cost (time/energy) for evaluating the results for any given parameter set increases proportionally. Instead, we present alternative techniques to represent entire feature families for a wide range of parameters based on topological feature definitions. These techniques encode a wide range of threshold- and gradient-based features in a parameter independent manner. Furthermore, various additional attributes can be computed for all features, such as characteristic length scales [2] and descriptive statistics [4]. In the following, we first describe some theoretical concepts of Morse theory which are used to define and represent features, and subsequently discuss an application where the concepts introduced are used to identify burning cells in a simulated turbulent combustion experiment [4].

3.1. Use Case: Extracting Burning Cells in Turbulent Combustion

In combustion sciences, detailed simulations are used to support the fundamental basis behind the interpretation of critical laser-based flame diagnostic approaches. For a detailed discussion on basic combustion theory we refer the reader to the book by Williams [25] and for the theory and numerical modeling of flames including turbulence to Poinsot and Veynante [20]. We focus this study on advanced ultra-lean premixed combustion systems, see Bell et al. [1] for a discussion of recent research in simulation of lean premixed combustion, and our ultimate goal is to: (i) Augment and validate laser-based diagnostics; (ii) Assess the underlying assumptions in their interpretation; and (iii) Aid the development of models to characterize the salient behavior of these flame systems.

Low-swirl injectors are emerging as an important new combustion technology. In particular, such devices can support a lean hydrogen-air flame that has the potential to dramatically reduce pollutant emissions in transportation systems and turbines designed for stationary power generation. However, hydrogen flames are highly susceptible to various fluid-dynamical and combustion instabilities, making them difficult to design and optimize. Due to these instabilities, the flame tends to arrange itself naturally in localized cells of intense burning that are separated by regions of complete flame extinction.

Existing approaches to analyze the dynamics of flames, including most standard experimental diagnostic techniques, assume that the flame is a connected interface that separates the cold fuel from hot combustion products. In cellular hydrogen-air flames, many of the basic definitions break down—there is no connected interface between the fuel and products, and in fact there is no concrete notion of a "progress variable" that can be used to normalize the progress of the combustion reactions through the flame. As a consequence, development of models for cellular flames requires a new paradigm of flame analysis.

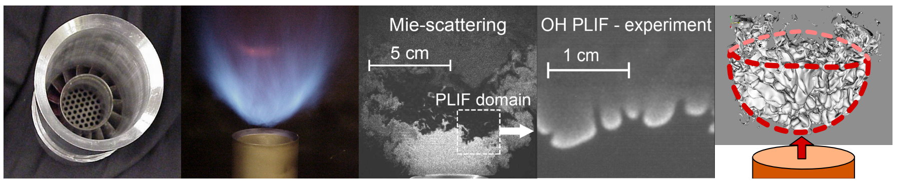


Figure 13. (a) Photo of a typical laboratory low-swirl nozzle. (b) Photo of a lean premixed CH_4 low-swirl flame. (c) Experimental Mie scattering image of a lean premixed H_2 flame. (d) PLIF data imaging the OH concentration in a lean premixed H_2 flame. (e) Rendering of the burning cells of the SwirlH2 simulation data. The cells form a bowl shaped structure with the arrow indicating the direction of the fuel stream.

Fig. 13a shows the detail of a low-swirl nozzle. The annular vanes inside the nozzle throat generate a swirling component in the fuel stream. Above the nozzle the resulting flow-divergence provides a quasi-steady aerodynamic mechanism to anchor a turbulent flame. Fig. 13b illustrates such a flame for a lean premixed CH_4-air mixture (the illustration shows a methane flame since H_2 flames do not emit light in the visible spectrum). Figs. 13c,13d show typical experimental data from laboratory low-swirl, lean H_2-air flames. Such data is used to extract the mean location and geometrical structure of instantaneous flame profiles. The images indicate highly wrinkled flame surfaces that respond in a complex way to turbulent structures and cellular patterns in the inlet flow-field.

To identify features in the simulation, we investigate merge trees computed on the rate of fuel consumption. The cellular regions of intense burning are identified by thresholding the local consumption rate. The process may be regarded as a generalized subsetting strategy, whereby subregions of a computational result may be sampled, explored and categorized in terms of a volume of space with an arbitrary application-specific definition for its boundary. Since we are interested in regions of high fuel consumption we have identified merge trees which encode the topology of super-level sets as appropriate data structure.

The computational model used to generate the simulation results explored in this study incorporates a detailed description of the chemical kinetics and molecular transport, thus enabling a detailed investigation of the interaction between the turbulent flow field and the combustion chemistry. The data is saved for 332 snapshots in time, at a resolution of 1024^3, representing about 25 cm^3 in space. Each snapshot is roughly 12–20 Gigabytes in size totaling a combined 8.4 Terabytes of raw data. The large size of this data makes each analysis pass expensive. Our approach is to compute a *meta-segmentation* of the domain, *i.e.*, a representation that encodes many possible segmentations for many threshold values that can be finalized during interactive exploration. The topology-based data structures and algorithms we use are rooted in Morse theory [14,15]. We use hierarchical merge trees as a meta-segmentation, and present an approach for their computation in a streaming manner. In the following we briefly review the necessary theoretical background.

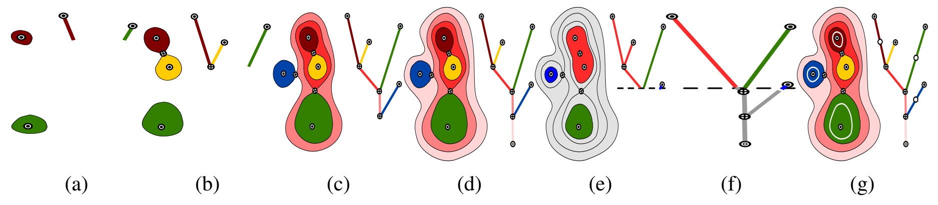


Figure 14. (a)-(d) Constructing a merge tree and corresponding segmentation by recording the merging of contours as the function value is swept top-to-bottom through the function range. (e) The segmentation for a particular threshold can be constructed by cutting the merge tree at the threshold, ignoring all pieces below the threshold and treating each remaining (sub-)tree as a cell. (f) The segmentation of (e) constructed by simplifying all saddles above the threshold. (g) The merge tree of (d) augmented by splitting all arcs spanning more than a given range.

3.2. Morse Theory and Merge Trees

Given a smooth simply connected manifold $\mathbb{M} \subset \mathbb{R}^n$ and a function $f : \mathbb{M} \rightarrow \mathbb{R}$ the *level set* $L(s)$ of f at isovalue s is defined as the collection of all points on $\mathbb{R}$ with function value equal to s: $L(s) = \{p \in \mathbb{M} | f(p) = s\}$. A connected component of a level set is called a *contour*. Similarly, we define *super-level sets* $LS(s) = \{p \in \mathbb{M} | f(p) \geq s\}$ as the collection of points with function value greater or equal to s and *super-contours* as their connected components. The *merge tree* of f represents the merging of super-contours as the isovalue s is swept top-to-bottom through the range of f, see Figure 14(a). Each time the isovalue passes a maximum a new super-contour is created and a new leaf appears in the tree. As the function value is lowered, super-contours merge, and this is represented in the tree as the joining of two branches. Departing slightly from standard Morse theory we will call a point p in a merge tree *regular* if it has valence two and *critical* otherwise. Note that, this definition of critical points ignores points at which contours split or change genus as well as local minima as these do not affect the structure of a merge tree. Consequently, there exist only three types of critical points: maxima with no higher neighbors, saddles with two or more higher neighbors, and the global minimum with no lower neighbors.

Each branch in the merge tree corresponds to a neighboring set of contours and therefore branches represent subsets of $\mathbb{M}$. In the application we will discuss, we are interested in regions of high fuel-consumption rate and we use sub-trees above a given threshold to define burning cells. Given a threshold, t, for the fuel consumption rate, f, we determine the corresponding burning cells by (conceptually) cutting the merge tree of f at t creating a forest of trees. Each individual tree represents one connected burning cell, see Figure 14(e). In practice, rather than cutting the merge tree and traversing sub-trees the same information is stored more efficiently as a simplification sequence. A merge tree is simplified by successively merging leaf branches with their sibling branch. We order these simplifications by decreasing function value of the merge saddles and store the resulting simplification sequence. A merge tree along with its simplification sequence is called a *hierarchical merge tree*. In this framework burning cells at threshold t are defined as sets of all leaf branches with function value greater than or equal to t of the tree simplified to t, see Figure 14(f).

3.3. Streaming Computation of Hierarchical Merge Trees

The large data size necessitates a specially adapted algorithm that operates in a streaming fashion [4]. In general, streaming algorithms are attractive for large scale data processing due to their low memory foot print, high performance, and their ability to avoid temporary files and also to reduce file I/O. The streaming merge tree algorithm presented here provides several additional advantages. By representing the input data as a stream of vertices and edges it naturally supports culling of vertices outside a given function range of interest (in this case very low function values for example). Furthermore, we can perform an on-the-fly simplification that significantly reduces the size of the merge trees. Finally, a streaming approach is more flexible with respect to input formats and interpolation schemes. We assume our input consists of a stream of vertices, edges between vertices, and finalization flags, where a vertex v must appear before the first edge that references v and the finalization tag of v appears after the last edge that uses v.

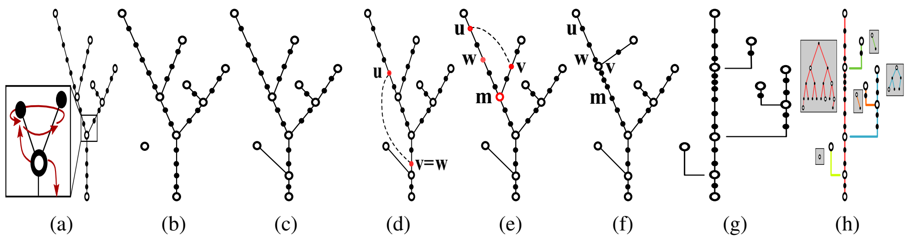


Figure 15. Streaming merge tree construction: (a) Intermediate merge tree with the zoom-in showing the pointers used to maintain the data structure in red. Each vertex stores a pointer to its sibling, creating a linked list, as well as a pointer to one of its parents and a pointer to its child; (b) Merge tree of (a) after the addition of a new vertex; (c) The tree of (b) after attaching the lonely vertex with an edge; (d) Adding the dotted edge from u to v does not change the merge tree as v is already a descendant of u; (e) Adding the dotted edge from u to v creates a (temporary) loop which is immediately closed by merging the sub-branches $w-m$ and $v-m$. This creates a new saddle at v and changes m to a regular vertex shown in (f); (g) A branch decomposition of the tree in (f). (h) The branch decomposition of (g) where each branch stores a balanced search tree of its vertices.

Algorithm. In the following discussion we use the nomenclature of vertices and links for the dynamically changing merge tree and nodes and arcs for elements of the final tree, which for distinction we will call merge graph. Note that vertices of the dynamic merge tree have a natural one-to-one correspondence to vertices of the input stream. The algorithm is based on three atomic operations corresponding to the three input elements: *CreateVertex* is called each time a vertex appears in the input stream; *AddEdge* is called for each edge in the input; and *FinalizeVertex* is called for each finalization flag. We illustrate the algorithm using the examples shown in Fig. 15. At all times we maintain a merge tree consisting of all (unfinalized) vertices seen so far. Each vertex stores its links using three pointers: A *down* pointer to its (unique) lower vertex; An *up* pointer to one of its higher vertices; and A *next* pointer to one of its siblings, see Fig. 15(a). The up and next pointers form a linked list of higher vertices allowing to completely traverse the tree. *CreateVertex* creates a new isolated vertex, see Fig. 15(b). *AddEdge* connects two vertices (u,v) of the existing tree and wlg. we assume $f(u) > f(v)$. With respect to the merge tree structure, each edge (u,v) provides one key piece of information about f: The existence of (u,v) guarantees that u's contour at $f(u)$ must evolve (potentially through merging) into v's contour at $f(v)$ and thus v must be a descendant of u in the tree. In the following we check this *descendant* property and if necessary adapt the merge tree to ensure its validity. First, we find the lowest descendant w of u such that $f(w) \geq f(v)$. If $w = v$ then the current tree already fulfills the descendant property and it remains unchanged , see Figs. 15(c) and 15(d). However, if $w \neq v$ the current tree does not fulfill the descendant property and must be modified. Another indication of this violation is that adding the new edge to the tree would create a loop which cannot exist in a merge tree, see Fig. 15(e). To restore a correct merge tree that reflects the new descendant information we perform a merge sort of both branches starting at w and v respectively until we find the first common descendant m of u and v, see Fig. 15(f). In this merge sort step, all other branches originating from a saddle are preserved. The merge sort closes the (illegal) loop and creates the correct merge tree. The pseudo code of the *AddEdge* algorithm is shown in Fig. 16

As vertices become finalized we remove them from the tree. However, care must be taken that this early removal does not impact the outcome of the computation. For example, a finalized saddle may become regular through global changes in the tree, see below. In particular, critical points of the merge tree must be preserved as they may form the nodes of the merge graph. In this context it is important to understand how the local neighborhood of a vertex relates to it being critical. Here, we are interested in the connected components of the *upper link* [6] of a vertex v. Here the upper-link is defined as all vertices v_i connected to v with $f(v_i) > f(v)$ and the edges between them. By definition, a saddle must have more than one component in its upper link, while the upper link of the global minimum is a topological sphere of dimension $n-1$. However, the reverse is not true: A vertex can have multiple components in its upper link yet not be a (merge-tree) saddle (its upper link component may belong to contours that merged ear-

```
AddEdge(Vertex $u$, Vertex $v$, MergeTree $T$, Function $f$)
    if $f(u) < f(v)$
        Swap($u,v$)
    endif
    $w = u$
    while f($T$.GetChild($w$)) $\geq$ f($v$) // Find the lowest child $w$
        $w = T$.GetChild($w$)              // of $u$ such that $f(w) \geq f(v)$
    endwhile
    if $w \neq v$    // If $v$ is not a direct descendant of $u$
        $T$.MergeSort($w,v$) // Close the loop $w,v,m$ (see Fig, 15(e))
    endif
```

Figure 16.: Pseudo-code for the *AddEdge* function to update a merge tree T to include the edge (u,v)

```
FinalizeVertex(Vertex $v$, MergeTree $T$, MergeGraph $Final$)
    $T$.MarkAsFinalized($v$)
    $c = T$.GetChild($v$)
    if $T$.IsRegular($v$)
        $T$.BypassVertex($v$) // Link all parents of $v$ to its child $c$
        $T$.RemoveVertex($v$)
    endif
    if $T$.IsCritical($c$) AND $T$.IsFinalized($c$)
        forall $p \in T$.GetParents($c$)
            if $T$.IsCritical($p$) AND $T$.IsFinalized($p$)
                    AND $T$.GetParents($p$) == $\emptyset$
                $Final$.AddArc($c,p$)
                $T$.RemoveEdge($c,p$)
                $T$.RemoveVertex($p$)
                endif
            endif
        endfor
        if $T$.GetChild($c$) == $\emptyset$ AND $T$.GetParents($c$) == $\emptyset$
            $T$.RemoveVertex($c$)
        endif
    endif
```

Figure 17.: Pseudo-code for the *FinalizeVertex* function to update a merge tree T after finalizing vertex v.

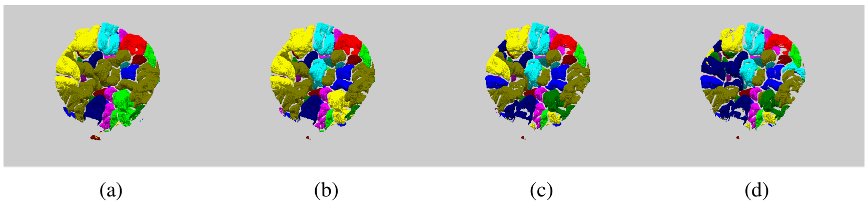


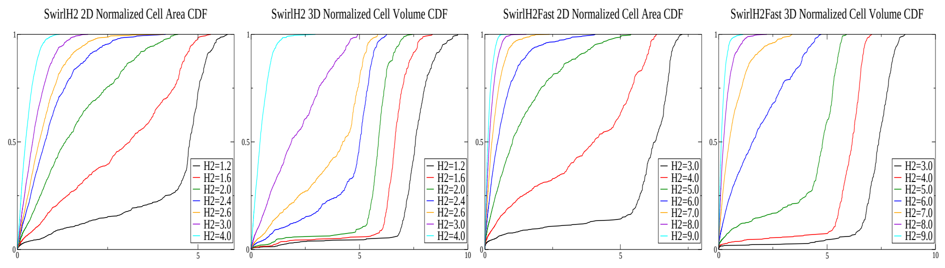
Figure 18. Burning cells in the center of the SwirlH2 data at time step 1500 randomly colored using 4.0 (a), 5.0 (b), 6.0 (c), and 7.0 (d) as fuel consumption threshold respectively.

lier); Similarly, the upper link of a lo-
cal minimum is a topological sphere yet it is not critical with respect to the merge tree. This relationship becomes important when deciding which finalized vertices can be removed safely.

Finalizing a vertex v indicates that its entire local neighborhood has been processed. Thus, a finalized regular vertex will remain regular as more edges are added and can be removed without impacting the tree. Similarly, a finalized maximum will remain a leaf of the tree and thus will always correspond to a node of the merge graph. A finalized saddle, however, may become regular as its branches can be merged through additional edges not incident to the vertex itself. Finally, the global minimum of all vertices seen so far may become regular or a saddle when further vertices or edges are added. Nevertheless, a branch of the merge tree which consists only of a finalized leaf and a finalized saddle/minimum is guaranteed to be an arc of the merge graph (none of the contours it represents can be changed anymore). These conditions result in the algorithm shown in Fig. 17. When finalizing a vertex v we first check whether it is a regular vertex and if so remove it from the tree. We then determine whether this finalization has created a leaf edge between two finalized critical vertices. All such edges must be arcs in the merge graph and are added alongside their nodes. This procedure peels finalized branches from top to bottom from the merge tree and adds them to the merge graph. Finally, we check whether only the global minimum is left and remove it if necessary.

3.4. Results

The hierarchical merge trees represent a meta-segmentation that can be explored interactively obtain reconstructions of the burning cells interactively. For example, Fig. 18 shows the segmentation of the burning cells for two different thresholds of fuel consumption. The effective compression by only storing the meta-segmentation was about two orders of magnitude. The ability to move between threshold parameters quickly allows parameter studies, such as illustrated in Fig. 19, where cumulative density functions (CDFs) of several statistics are shown for the burning cells. A traditional analysis of this data would require reprocessing the entire 8.4 terabytes of data 168 times, and instead, each new parameter study can be done in seconds.



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

Figure 19. Weighted cumulative density functions of the distributions of cell sizes for various thresholds for the surface based analysis of [3] (a),(c) and the volumetric analysis presented here (b),(d) for the SwirlH2 and SwirlH2Fast data set respectively. Unlike the idealized flames of [5] the distributions for lower thresholds are exponential indicating few large cells rather than the expected many small cells.

4. Conclusion

Interactive visualization and data exploration are an indispensable part of hypothesis testing and knowledge discovery. Like most fields discussed in this book, visualization faces substantial scientific data management challenges that are the result of growth in size and complexity of the data being produced by simulations and collected from experiments. In this chapter we have indicated some effective directions in scientific data management that play a unique role in interactive data exploration.

With data of massive scale, it is often useful to perform a multiresolution analysis, working first with a smaller, coarser version of the data, then progressively refining the analysis as interesting features are revealed. We saw with ViSUS that a space-filling curve model has proven to be highly efficient for interactive analysis of massive data. Topological methods for multi-scale, quantitative feature detection and analysis have also been demonstrated to be highly effective by providing intermediate concise descriptions of the data. By providing a higher level abstraction, they enable scientists to explore feature definitions interactively even if the raw data is prohibitively large in size.

References

[1] J. Bell, M. Day, A. Almgren, M. Lijewski, C. Rendleman, R. Cheng, and I. Shepherd. Simulation of lean premixed turbulent combustion. *SciDAC 2006 (Journal of Physics: Conference Series)*, 46:1–15, 2006.

[2] J.C. Bennett, V. Krishnamoorthy, S. Liu, R.W. Grout, E.R. Hawkes, J.H. Chen, J. Shepherd, V. Pascucci, and P.-T. Bremer. Feature-based statistical analysis of combustion simulation data. In *IEEE Transactions on Visualization and Computer Graphics, Proceedings of the 2011 IEEE Visualization Conference*, volume 17, pages 1822–1831, 2011.

[3] P.-T. Bremer, G. Weber, V. Pascucci, M. Day, and J. Bell. Analyzing and tracking burning structures in lean premixed hydrogen flames. *IEEE Transactions on Visualization and Computer Graphics*, 16(2):248–260, 2010.

[4] P.-T. Bremer, G. Weber, J. Tierny, V. Pascucci, M. Day, and J. B. Bell. Interactive exploration and analysis of large scale simulations using topology-based data segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 17(99), 2010.

[5] M. Day, J. Bell, P.-T. Bremer, V. Pascucci, V. Beckner, and M. Lijewski. Turbulence effects on cellular burning structures in lean premixed hydrogen flames. *Combustion and Flame*, 156:1035–1045, 2009.

[6] H. Edelsbrunner, J. Harer, and A. Zomorodian. Hierarchical Morse complexes for piecewise linear 2-manifolds. In *Proceedings of the seventeenth annual symposium on Computational geometry*, SCG '01, pages 70–79, New York, NY, USA, 2001. ACM.

[7] A. Gyulassy, M. Duchaineau, V. Natarajan, V. Pascucci, E. Bringa, A. Higginbotham, and B. Hamann. Topologically Clean Distance Fields. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1432–1439, November/December 2007.

[8] Rajkumar Kettimuthu and Others. Lessons learned from moving earth system grid data sets over a 20 gbps wide-area network. In *SC*, pages 194–198. ACM, Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010).

[9] S. Kumar, V. Pascucci, V. Vishwanath, P. Carns, R. Latham, T. Peterka, M. Papka, and R. Ross. Towards parallel access of multi-dimensional, multiresolution scientific data. In *Proceedings of 2010 Petascale Data Storage Workshop*, November 2010.

[10] S. Kumar, V. Vishwanath, P. Carns, B. Summa, G. Scorzelli, V. Pascucci, R. Ross, J. Chen, H. Kolla, and R. Grout. Pidx: Efficient parallel i/o for multi-resolution multi-dimensional scientific datasets. In *Proceedings of IEEE Cluster 2011*, September 2011.

[11] Sidharth Kumar, Venkatram Vishwanath, Philip Carns, Joshua A. Levine, Robert Latham, Giorgio Scorzelli, Hemanth Kolla, Ray Grout, Robert Ross, Michael E. Papka, Jacqueline Chen, and Valerio Pascucci. Efficient data restructuring and aggregation for i/o acceleration in pidx. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 50:1–50:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[12] J. K. Lawder and P. J. H. King. Using space-filling curves for multi-dimensional indexing. *Lecture Notes in Computer Science*, 1832:20, 2000.

[13] A. Mascarenhas, R. W. Grout, P.-T. Bremer, E. R. Hawkes, V. Pascucci, and J.H. Chen. *Topological feature extraction for comparison of terascale combustion simulation data*, pages 229–240. Mathematics and Visualization. Springer, 2011.

[14] Y. Matsumoto. *An Introduction to Morse Theory. Translated from Japanese by K. Hudson and M. Saito*. American Mathematical Society, 2002.

[15] J. W. Milnor. *Morse Theory*. Princeton Univ. Press, New Jersey, USA, 1963.

[16] V. Pascucci, D. E. Laney, R. J. Frank, G. Scorzelli, L. Linsen, B. Hamann, and F. Gygi. Real-time monitoring of large scientific simulations. In *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, pages 194–198, New York, NY, USA, 2003. ACM.

[17] Valerio Pascucci and Randall J. Frank. Global static indexing for real-time exploration of very large regular grids. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 2–2, New York, NY, USA, 2001. ACM Press.

[18] Sujin Philip, Brian Summa, Peer-Timo Bremer, and Valerio Pascucci. Parallel Gradient Domain Processing of Massive Images. In Torsten Kuhlen, Renato Pajarola, and Kun Zhou, editors, *Eurographics Symposium on Parallel Graphics and Visualization*, pages 11–19, Llandudno, Wales, UK, 2011. Eurographics Association.

[19] Sujin Philip, Brian Summa, Valerio Pascucci, and Peer-Timo Bremer. Hybrid cpu-gpu solver for gradient domain processing of massive images. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 244 –251, dec. 2011.

[20] Thierry Poinsot and Denis Veynante. *Theoretical and Numerical Combustion*. R.T. Edwards, second edition, 2005.

[21] Alfredo Rodriguez, Douglas B. Ehlenberger, Patrick R. Hof, and Susan L. Wearne. Three-dimensional neuron tracing by voxel scooping. *Journal of Neuroscience Methods*, 184(1):169 – 175, 2009.

[22] Hans Sagan. *Space-Filling Curves*. Springer-Verlag, New York, NY, 1994.

[23] B. Summa, G. Scorzelli, M. Jiang, P.-T. Bremer, and V. Pascucci. Interactive editing of massive imagery made simple: turning Atlanta into Atlantis. *ACM Trans. Graph.*, 30:7:1–7:13, April 2011.

[24] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, March 2000.

[25] Forman A. Williams. *Combustion Theory*. Westview Press, second edition, 1994.