Title
Scientific Data Management Challenges in High Performance Visual Data Analysis

Authors
Bethel, E. Wes
Childs, Hank
Mascarenhas, Ajith
et al.

Publication Date
2009-06-01

Contents

List of Tables

List of Figures

Chapter 7

Scientific Data Management Challenges in High Performance Visual Data Analysis

E. Wes Bethel and Prabhat
High Performance Computing Research Department,
Lawrence Berkeley National Laboratory
Berkeley, California, USA, 94720.

Hank Childs
Computing Applications and Research
Lawrence Livermore National Laboratory
7000 East Avenue, Livermore, Ca, 94550.

Ajith Mascarenhas
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
7000 East Avenue, Livermore, CA 94550.

Valerio Pascucci
Scientific Computing and Imaging (SCI) Institute
School of Computing
University of Utah
72 S Central Campus Drive 3750 WEB Salt Lake City, UT, 84112

Abstract Visualization is a highly data intensive science: visualization algorithms take as input vast amounts of data produced by simulations or experiments, and then transform that data into imagery. It turns out, as we shall explore in this chapter, that visualization reveals a somewhat different view of scientific data management challenges than are examined elsewhere

in this book. For example, a data ordering and storage layout that works well for saving data from memory to disk may not be the best thing for subsequent visual data analysis algorithms.

This chapter will present four broad topic areas under this general rubric: (1) a view of SDM-related issues from the perspective of implementing a production-quality, parallel capable visual data analysis infrastructure; (2) novel data storage formats for multi-resolution, streaming data movement, access and use by post-processing tools; (3) data models, formats and APIs for performing efficient I/O for both simulations and post-processing tools, discussion of issues and previous work in this space; (4) how combining state-of-the-art techniques from scientific data management and visualization enables visual data analysis of truly massive datasets.

7.1 Introduction

Scientific visualization, which is the transformation of abstract data into readily comprehensible images, and visual data analysis/analytics, which combines visualization with analysis, play a central role in the modern scientific process. We use the umbrella term "visualization" to refer to this broad set of investigatory techniques aimed at enabling knowledge discovery – gaining insight – from large, complex collections of scientific data.

The term "scientific visualization" was coined in 1987 in a landmark report [21], which said: "Visualization is a method of computing. It transforms the symbolic into the geometric, enabling researchers to observe their simulations and computations. Visualization offers a method for seeing the unseen. It enriches the process of scientific discovery and fosters profound and unexpected insights. In many fields, it is already revolutionizing the way scientists do science...The goal of visualization is to leverage existing scientific methods by providing new scientific insight through visual methods."

While the term was coined in 1987, the art and science of visualization dates back hundreds of years to DaVinci's illustrations, or even earlier, to Cicero's written account of an early orrery constructed by the Greek philosopher Posidonious to exhibit the diurnal motions of the sun, moon and five known planets[1]. A relatively recent overview in [16] provides a good survey of the field's breadth and depth: algorithms for visualizing scalar, vector and tensor fields, geometric modeling, virtual environments for visualization, large data visualization, perceptual and cognitive issues in visualization, visualization software and frameworks, software architecture, and so forth.

[1]See http://en.wikipedia.org/wiki/Orrery

Visualization is a very data intensive science: visualization algorithms take as input vast amounts of data produced by simulation or experiment, and then transform that data into imagery. It turns out, as we shall explore in this chapter, that visualization reveals a somewhat different view of scientific data management challenges than are examined elsewhere in this book. For example, a data ordering and storage layout that works well for saving data from memory to disk may not be the best thing for subsequent visual data analysis algorithms.

This chapter will present four broad topics areas under this general rubric: (1) a view of SDM-related issues from the perspective of implementing a production-quality, parallel capable visual data analysis infrastructure; (2) novel data storage formats for multi-resolution, streaming data movement, access and use by post-processing tools; (3) data models, formats and APIs for performing efficient I/O for both simulations and post-processing tools, discussion of issues and previous work in this space; (4) how combining state-of-the-art techniques from scientific data management and visualization enables visual data analysis of truly massive datasets.

7.2 Production-level, Parallel Visualization Tool Perspective on SDM

A production-level, parallel visualization tool is a robust program that is used by a potentially large population of users to perform diverse visualizations and analyses, normally on data from many different types of file formats and with varying types of data models. As such, these tools are somewhat different from scientific simulation codes in that:

- They "unify" many different data models. For example, they support many mesh types, field types, and various centerings for those fields (point-centered, cell-centered, etc).
- They are not the originators of the semantics placed on the data. Therefore, the meaning of each of the arrays of data must somehow be provided to the visualization tool.
- When run in a parallel environment, visualization tools are expected to adapt to available resources (e.g. number of processors) and partition the data for processing in a way that achieves good load balance.

In the following subsections, we describe how these visualization tools use the data from scientific simulations. In particular, we will discuss:

- How a production-level, parallel visualization tool loads data, processes it, and produces results.

- How a production-level, parallel visualization tool can optimize its data management and processing with the presence of metadata.
- The importance of data semantics from the perspective of a production-level, parallel visualization tool.

7.2.1 How Data is Processed

The three major parallelized, production-level visualization tools – EnSight [9], VisIt [20], and ParaView [17] – all employ similar strategies. They use a client-server design, where the client provides a user interface on the ser's desktop, and the server runs where the data is located, which is assumed to have resources for parallel processing. The general data management strategy for the parallel server can essentially be described as a "Scatter-Gather" algorithm. The process can be characterized in three steps:

1. I/O (scatter): load data (in parallel) onto the server.
2. Processing: employ visualization and analysis algorithms; transform the data to geometry.
3. Rendering (gather): transform the geometry into images.

7.2.1.1 I/O

Since visualization is a data intensive endeavor, I/O is frequently the slowest and most expensive part of the entire visualization pipeline. As such, it is advantageous to parallelize the data loading. A typical design pattern is for each processor of the parallel server to read a portion of the input data set, which is the mechanism that "scatters" the data set across each of the processors. The key question during the I/O phase is how to assign portions of the input data set to the processors of the server. We simplify the discussion below, by assuming that the data is being read from disk, i.e. *not* being processed in situ as part of a single program with the simulation code.

When the visualization server processes a portion of the data set, the input data set must be partitioned and distributed across the server's processors. When the simulation outputs data, it may impose restrictions on data partitioning. There are two typical scenarios:

1. The underlying I/O infrastructure only supports a partitioning scheme fixed by the simulation when the file(s) were created. Most of the time, this scenario corresponds to having one atomic chunk of data for each processor[2]. Examples of I/O libraries of this type are Silo [30] and Exodus [31]. Other examples include file-per-processor output, which

[2]Atomic in the sense that partial reads of the chunk of data are not possible.

may be a good way to achieve I/O performance for the simulation's data-write phase, but has undesirable consequences for processing tools that read their data. Those processing tools are then forced to reconcile between the simulation's degree of parallelism and their own. For example, the simulation may decompose a three-dimensional space into one thousand pieces, but the processing tool may be running with only five processors. In this case, the processing tool must find a way to partition the pieces across its processors, either by combining all of the pieces on a given processor into one large piece or by respecting piece layout and supporting multiple pieces per processor.

2. The underlying I/O infrastructure supports re-partitioning during read. Most of the time, this scenario corresponds to having all of the data in one large file, with the I/O infrastructure supporting operations like hyperslab reads, collective I/O, and so forth. Examples of formats that can repartition data in this manner are ViSUS [26], SAF [22], and HDF5 [13].

These two scenarios are well supported by the major, parallel, production visualization tools, although the scenarios are supported differently in terms of how the tools do parallel partitioning. For the first case (imposed partitioning), each subset of the partition normally consists of "domains", where each domain consists of the portion operated on by a single processor. In this case, the visualization tool distributes the domains across its processors. For the second case (adaptive partitioning), the visualization tool forms its own partition of the data set by having each processor read in a unique piece. In both cases, it is important that each processor has an approximately equal amount of data to read, which correlates strongly with work to be performed in subsequent stages. From an SDM perspective, the summary is that both ways of writing data are acceptable.

7.2.1.2 Processing

The modern parallel visualization tools all use a data flow network processing design [1, 32, 36]. Data flow networks have base types of *data objects* and *components* (sometimes called process objects). The components can be *filters*, *sources*, or *sinks*. Filters have an input and an output, both of which are data objects. Sources have only data object outputs, while sinks have only data object inputs. A *pipeline* is an ordered collection of components. Each pipeline has a source (typically a file reader) followed by one or more filters (for example slicing or contouring algorithms) followed by a sink (typically a rendering algorithm). When a pipeline is executed, data comes from the source and flows from filter to filter until it reaches the sink. There are many variations on this general design that include caching, how the execution takes place (push versus pull), multiplicity in terms of sources and sinks, feedback loops in the filters, reusing arrays from data object to data object to reduce

memory footprint, and optimizations like parallel-pipelined operation so that different stages of the pipeline may operate concurrently.

When the client asks the server to perform some operations on a data object, each processor of the server sets up an identical data flow network. They only differ in the portion of the data set that they process. The majority of visualization operations are "embarrassingly parallel" – the processing can occur in parallel with no communication between the parallel processes. For these operations, the only concern is artifacts that can occur along the boundaries of a chunk. For example, a stencil-based algorithm that is run in parallel may require data from adjacent grid points that are owned by another processor. The typical way to resolve this problem is using redundant data located at the boundary, which is often referred to as *ghost data*.

7.2.1.3 Rendering and Remote/Distributed Visualization

Within the context of visualization software architectures, the majority of SDM-related concerns reside in I/O and processing stages. The later stage of visualization – rendering, where visualization results (geometry, 3D volumes, etc.) are transformed into images – has its own unique set of visualization-centric SDM-related issues.

As context, remote and distributed visualization applications can use one of three general types of architectures as shown in Figure 7.1. A discussion of the relative performance and usability merits of these different configurations is presented in [33]. The important point here, within the context of SDM-related issues, is that moving data across machine boundaries can be a non-trivial task. The data might be raw data, as in the desktop-only configuration; it might be geometric output produced by visualization tools, as in the "cluster isosurface" configuration; or it might be raw image pixels, as in the "cluster render" configuration. Unlike "traditional" data movement applications (e.g., ftp and its variants), the visualization use model often dictates which pipeline partitioning will work best given a particular problem size and set of machines/networks. For instance, if maximizing rendering interactivity of static data is the desired target, then one of the configurations that uses desktop graphics hardware for rendering is the best choice, assuming the problem will fit onto the desktop machine. If maximizing throughput is the objective, e.g., cycling through large, time-varying data, then the configuration where data I/O and processing is performed on a parallel machine and image sent to the remote viewer is the best choice. The trend we see in high performance visualization for many problem domains is more towards this latter configuration, which exhibits favorable scaling characteristics as data sizes grow larger.

In the case when all rendering occurs on the desktop, all data representation and data transfer issues are encapsulated inside the graphics library (though there may be substantial SDM issues to consider in the visualization pipeline prior to the rendering stage). In the case where rendering occurs on

one machine and image pixels are transmitted to one or more remote machines for viewing, several interrelated issues appear: security (authorization and authentication), compression (lossless vs. lossy), efficient data movement (lossless vs. lossy, multistreamed, multicast), data formats and models for the pixel data.

For this latter issue, a widely adopted approach is the Remote Framebuffer Protocol (RFB), which is part of the popular Virtual Network Computing (VNC) client/server application for remote desktop access [29]. Here, the remote client connects to a central server where the data-intensive application is run; the resulting imagery is "harvested" by the VNC server, encoded into RFB format, and transmitted to the VNC client for display via a standard TCP connection. In its native form, VNC is not capable of capturing image pixels created by graphics hardware. Other recent work rectifies this shortcoming, as well as provides a solution layered atop the RFB protocol to capture and deliver image pixels produced by hardware-accelerated, distributed memory rendering infrastructure [28]. In both cases, the rendering infrastructure, particularly the image capture and remote delivery, is transparent to the visualization application.

7.2.2 How Metadata Can Enable Optimizations

In this section, we give an example that motivates how the presence of metadata[3] can lead to extensive optimizations for the visualization tool. I/O is the most expensive portion of a pipeline execution for almost every operation a visualization tool performs. We can reduce I/O and processing load by reading only the domains that are relevant to any given pipeline operation. This performance gain propagates through the pipeline, since the domains not read in do not have to be processed downstream.

Consider the example of slicing a three-dimensional data set by a plane (Figure 7.2). In this case, most of the domains will not intersect the plane – loading and processing those domains that do not intersect the slice plane is wasted effort. By using metadata, we can dramatically reduce both I/O and processing load: we can limit I/O and processing only to those domains needed to complete the task at hand. For example, if the slice filter had access to the spatial extents for each domain, it could calculate the list of domains whose bounding boxes intersects the slice and only process that list (note that false positives can potentially be generated by considering only the bounding box).

The performance gains one can realize from metadata optimizations can be extensive. From a theoretical perspective, if D is the total number of domains, then the number of domains intersected by the slice is typically $O(D^{2/3})$. Us-

[3]We define metadata as data about the total data set to be processed, whose size is small relative to the total data set itself

ing this fact, we observe that we might expect an order-of-magnitude improvement in performance by using metadata to optimize visualization processing. From a practical perspective, we ran some performance experiments to show exactly how much performance gain can result from metadata optimizations.

Table 7.1 presents the results of the study where we run a pair of visualization algorithms – slicing, and isocontouring – and measure the I/O and processing costs with- and without-metadata configurations. In the slicing case, we use spatial metadata to limit the subsets of data that are loaded to only those that intersect the slice plane. In the isocontouring case, we use metadata describing the data range for each block to limit I/O and processing only to those blocks containing data ranges of interest – those that intersect the isosurface.

The data for this study was produced by a Rayleigh-Taylor Instability simulation, which models fluid instability between heavy fluid and light fluid. The simulation was performed on a 1152x1152x1152 rectilinear grid, for a total of more than one and a half billion elements. The data was decomposed into 729 domains, with each domain containing more than two million elements. All timings were taken on a cluster of 1.4GHz Intel Itanium2 processors, each with access to two gigabytes of memory.

The processing time includes the time to read in a data set from disk, perform operations to it, and prepare it for rendering. Rendering was not included because it can be highly dependent on screen size. We note that using spatial metadata typically yields a consistent performance improvement, whereas performance gains resulting from metadata about fields defined on the mesh (e.g. pressure, density, etc.) can be highly problem specific. To illustrate this effect, we show results from running the contouring algorithm on simulation data from both early and late timesteps. In earlier timesteps, the fluids have not mixed much, so the shape of the contour approximates a planar slice and does not intersect many domains. In the later timestep, the fluid has undergone substantial mixing, and the contour has much greater surface area due to folding, so it intersects many more data domains.

In summary, the presence of metadata can improve performance to the point of being interactive for certain algorithms. And interactivity is widely regarded to be a key component for scientific discovery.

7.2.3 Data Models and Semantics

Visualization tools devote a substantial amount of code to representing data (i.e. data structures), importing data, and translating it into those data structures. Anecdotal evidence suggests that as much as 80% of any given visualization application is dedicated to these very activities. The large amount of "SDM-related code" typically comes from the fact that various simulation tools have many different ways to represent their data and the visualization often must support them all. By way of example, the VisIt visualization tool devotes approximately forty thousand lines of code to various data structures.

		Processing time (sec)		Data processed (MB)	
Algorithm	Processors	Without Metadata	With Metadata	Without Metadata	With Metadata
Slicing	32	25.3	3.2	6,375.6	708.4
Contouring (early time)	32	41.1	5.8	6,375.6	708.4
Contouring (late time)	32	185.0	97.2	6,375.6	3,948.0

Table 7.1: Performance results of visualization processing – slicing, and isocontouring – with and without metadata optimization. For slicing and early-time isocontouring, we see an order-of-magnitude performance gain resulting from the metadata optimization. For the late-time isocontouring, the performance gain is still substantial, but not as profound due to the fact that late-stage isocontour is more complex and spans more blocks of data.

This does not include the portion of the data model that is incorporated from a third party library (the Visualization ToolKit), which in fact forms the core of the data model (the portion for mesh and field representations). In addition, VisIt has over eighty separate file format readers, each of which ranges from eight hundred lines of code to twelve thousand lines of code. Approximately 150,000 lines of code in VisIt is devoted to file format readers.

In addition to a basic data model for representing standard mesh types (e.g. rectilinear, curvilinear, unstructured, adaptive mesh refinement (AMR), and point meshes) and fields (e.g. scalars, vectors, and tensors), production visualization tools must understand many types of metadata about the data set to perform certain operations. We list a subset of this metadata to give a feel for how deep the visualization tool must go:

- For each array in the file that corresponds to data that should be visualized, the tool must understand what this array is and how it should be interpreted. For example, the visualization tool must understand that an array in a file labeled "den" is in fact a scalar field defined on a mesh. It is often not necessary to know that "den" is actually the density field, but may be necessary to know if the field values are explicit (i.e. density of some material per unit of volume) or implicit (i.e. density is directly related to the volume of the cell).

- Which cells in the mesh, if any, are ghost cells.

- If the mesh is hierarchical, as is the case with Adaptive Mesh Refinement (AMR) meshes, then the visualization tool must understand metadata describing how the patches of the mesh nest from coarse to fine resolution.

- If the mesh is from a multi-domain data set, the metadata defines how the domains abut. This information is required for many operations, such as computation of ghost data.
- Metadata for optimizations, such as the per-domain bounding boxes discussed in the previous section.
- Metadata about temporal characteristics, such as the simulation time and cycle identifier.
- Information such as volume fractions for Eulerian calculations, including maintaining sparse matrix structures for efficient representation of this information.

7.2.4 A Real World Production Parallel Visualization Tool

For concreteness, we describe the architecture and operation of a specific, production-quality, parallel visualization, VisIt, which implements many of the concepts described in the previous sections. In terms of I/O, it supports both imposed and adaptive partitioning (see Section 7.2.1.1). It also caches all I/O, which is frequently the dominant portion of execution time. In terms of processing, it uses a pipeline design, although its user interface does not expose pipeline constructs to users. VisIt's pipeline design makes full use of *contracts*, which enables the components of a data flow network to specify and communicate optimizations to achieve higher levels of performance efficiency (see Section 7.2.2). Many filters utilize metadata to limit the amount of data being processed and many file format readers produce metadata, such as per-domain bounding boxes. VisIt uses two approaches for rendering. First, surfaces with a relatively small number of geometric primitives (e.g., triangles, line segments, etc.) are sent to the client and rendered locally using the local desktop's graphics hardware. Surfaces with a relatively large amount of geometric primitives remain on the server where VisIt renders them in parallel, then the VisIt server sends the resulting imagery to the remote client. VisIt automatically decides which rendering approach to use based on the number of geometric primitives, but this decision can be overridden by users. In terms of VisIt's data model, VisIt processes all of the mesh types and field types described in Section 7.2.3. It also pays special attention to preserving information about data layout and ordering, so that users can ask debugging type questions such as, "what is the value of the 110th element in the 41st domain?" This design represents a large amount of effort. VisIt has over two hundred filters, twenty different ways to render data, and ninety different file readers, adding up to over one and a half million lines of C++ code.

Summarizing the entire section, data representation issues and designs of file formats are a critical issue for visualization tools. First, the visualization tool needs to be aware of most of the data that the simulation code itself is aware of, simply because much of that information is directly visualized or needed for

proper visualization. Second, additional metadata can enable optimizations and greatly improve the performance of a visualization tool. Third, data layout issues, such as the way data can be partitioned for parallelization, are very important and can have a profound impact on end-to-end performance and usability.

7.3 Multi-resolution Data Layout for Large Scale Data Analysis

In recent years, computational scientists with access to powerful supercomputers have successfully simulated fundamental physical processes with the goal of shedding new light on our understanding of nature. Such simulations often produce massive amounts of data: grids of size 1024^3 to 4096^3 at multiple timesteps and dozens of variables per grid point are not uncommon. This data must be visualized and analyzed to verify and validate the underlying model, to understand the phenomenon in detail, and to develop new insights into fundamental physics. Both data visualization and data analysis are vibrant research areas and much effort is being spent on developing advanced, new techniques to process the massive amounts of data produced by scientists. In this section, we describe a multi-resolution data layout, which provides the ability for quick access to data at varying levels of resolution, from coarse to fine.

7.3.1 Background

To provide context, we highlight these two components in a typical visualization and analysis pipeline shown in Figure 7.3. We assume that raw data from simulations is available as real-valued, regular samples of space-time. Due to the large size of datasets, we emphasize that all data samples cannot all be loaded into main memory at once; it is not feasible to use standard implementations of visualization and analysis algorithms on these large datasets.

Re-ordering this raw data into a suitable multi-resolution data layout can improve the efficiency of both visualization and analysis. Multi-resolution layouts enable interactive visualization by allowing the user to first load the data at a coarse level, then progressively refine by adding more samples to obtain a more detailed view. Classical schemes, e.g., those based on bricking or chunking, do not readily support the type of data access required for progressive or multiresolution techniques. In the following, we describe our hierarchical z-order data layout scheme. It builds on the coherent layout provided by the z-order space filling curve by incorporating a coarse-to-fine hierarchy on the

ordering. Our system is very simple to implement and has been used as a core technology and applied to a variety of visualization algorithms, such as slicing, isosurfacing, and volume rendering on massive amounts of scientific simulation data.

A multi-resolution data layout is a key technology that plays a central role in advanced analysis algorithms that go beyond simple images or movies. Topological analysis is one such technique that is useful for providing a deeper understanding of scientific phenomena. In this type of application, the analysis takes the form of defining, detecting, and quantifying features in data. Features can and do exist at multiple scales in data. For this reason, an efficient, multi-resolution data layout and model is an integral part of high-performance implementations of such algorithms. For more information on state-of-the art topological analysis, see see [6–8,10,11,25], and for application of such techniques to the analysis of simulation data, including hydrodynamic instability, and comparative analysis, see [15, 18, 23].

7.3.2 Hierarchical Indexing for Out-of-Core Access to Multi-Resolution Data

Out-of-core computing [37] specifically addresses the issues of algorithm redesign and data layout restructuring that are necessary to enable data access patterns having minimal out-of-core processing performance degradation. Research in this area is also valuable in parallel and distributed computing, where one has to deal with the similar issue 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.

In the case of hierarchical visualization algorithms for volumetric data, the 3D input hierarchy is traversed from a coarse grid to the fine grid levels to build derived geometric models having adaptive levels of detail. The shape of the output models are then modified dynamically with incremental updates of their level of detail. The parameters that govern this continuous modification of the output geometry are dependent on runtime user interaction, making it impossible to determine, *a priori*, what levels of detail will be constructed. For example, parameters can be external, such as the viewpoint of the current display window or internal, such as the isovalue of a contour or the position of a slice plane. The general structure of the access pattern can be summarized into two main points: (1) 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 (2) within each level of resolution, 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.

In this section, we describe a static indexing scheme that induces a data layout satisfying both requirements (1) and (2) for the hierarchical traversal of n-dimensional regular grids. The scheme 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 [19] used in classical database approaches to the new indexing scheme can be implemented with a simple sequence of bit-string manipulations, making it appealing for a possible hardware implementation. Third, since there is no data replication, we avoid the performance penalties associated with dynamic updates as well as increased storage requirements typically associated with most hierarchical and out-of-core schemes.

Beyond the theoretical interest in developing hierarchical indexing schemes for n-dimensional space filling curves, our approach targets practical applications in out-of-core visualization algorithms. For details on related work, algorithmic analysis, and experimental results see [27].

7.3.2.1 Hierarchical Subsampling Framework

This section discusses the general framework for an efficient definition of a hierarchy over the samples of a dataset.

Consider a set S of n elements decomposed into a hierarchy $\mathcal{H}$ of k levels of resolution $\mathcal{H} = \{S_0, S_1, \ldots, S_{k-1}\}$ such that:

$$S_0 \subset S_1 \subset \cdots \subset S_{k-1} = S$$

where S_i is said to be coarser than S_j if $i < j$. The order of the elements in S is defined by a cardinality function $I : S \rightarrow \{0 \ldots n-1\}$. This means that the following identity always holds:

$$S[I(s)] \equiv s$$

where square brackets are used to index an element in a set.

One can define a derived sequence $\mathcal{H}'$ of sets S'_i as follows:

$$S'_i = S_i \backslash S_{i-1} \qquad i = 0, \ldots, k-1$$

where formally $S_{-1} = \emptyset$. The sequence $\mathcal{H}' = \{S'_0, S'_1, \ldots, S'_{k-1}\}$ is a partitioning of S. A derived cardinality function $I' : S \rightarrow \{0 \ldots n-1\}$ can be defined on the basis of the following two properties:

- $\forall s, t \in S'_i : I'(s) < I'(t) \Leftrightarrow I(s) < I(t)$;
- $\forall s \in S'_i, \forall t \in S'_j : i < j \Rightarrow I'(s) < I'(t)$.

If the original function I has strong locality properties when restricted to any level of resolution S_i, then the cardinality function I' generates the desired global index for hierarchical and out-of-core traversal. The scheme has strong locality if elements with close indices are also close in geometric position. These locality properties are well studied in [24].

The construction of function I' can be achieved as follows: (i) determine the number of elements in each derived set S'_i and (ii) determine a cardinality function $I''_i = I'|_{S'_i}$ restriction of I' to each set S'_i. In particular, if c_i is the number of elements of S'_i, one can predetermine the starting index of the elements in a given level of resolution by building the sequence of constants $C_0, \ldots, C_{k-1}$ with

$$C_i = \sum_{j=0}^{i-1} c_j. \tag{7.1}$$

Next, one must determine a set of local cardinality functions $I''_i : S'_i \rightarrow \{0 \ldots c_i - 1\}$ so that:

$$\forall s \in S'_i : I'(s) = C_i + I''_i(s). \tag{7.2}$$

The computation of the constants C_i can be performed in a preprocessing stage so that the computation of I' is reduced to the following two steps:

- given s determine its level of resolution i (that is the i such that $s \in S'_i$);
- compute $I''_i(s)$ and add it to C_i.

These two steps must be performed very efficiently as they will be executed repeatedly at run time. The following section reports a practical realization of this scheme for rectilinear cube grids in any dimension.

7.3.2.2 Binary Trees and the Lebesgue Space Filling Curve

This section reports the details on how to derive from the Z-order space filling curve the local cardinality functions I''_i for a binary tree hierarchy in any dimension and its remapping to the new index I'.

Indexing the Lebesgue Space Filling Curve. The Lebesgue space filling curve, also called Z-order space filling curve for its shape in the 2D case, is depicted in figure 7.4(a-e). The Z-order space filling curve can be defined inductively by a base Z shape of size 1 (figure 7.4a), that is by the vertices of a square of side 1 that are connected along a "Z" pattern. Such vertices can then be replaced each by a Z shape of size $\frac{1}{2}$ as in Figure 7.4(b). The vertices obtained in this way are then replaced by Z shapes of size $\frac{1}{4}$ as in Figure 7.4(c), and so on. In general, the i^{th} level of resolution is defined as the curve obtained by replacing the vertices of the $(i-1)^{th}$ level of resolution with Z shapes of size $\frac{1}{2^i}$. The 3D version of this space filling curve has the same hierarchical structure with the only difference being that the basic Z shape is

Level of Tree	0	1	2		3				4							
Z-order index (2 levels)	0	1														
Z-order index (3 levels)	0	2	1	3												
Z-order index (4 levels)	0	4	2	6	1	3	5	7								
Z-order index (5 levels)	0	8	4	12	2	6	10	14	1	3	5	7	9	11	13	15
hierarchical index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Table 7.2: Structure of the hierarchical indexing scheme for binary tree combined with the order defined by the Lebesgue space filling curve.

replaced by a connected pair of Z shapes lying on the opposite faces of a cube as shown in Figure 7.4(f). Figure 7.4(f-j) show five successive refinements of the 3D Lebesgue space filling curve. The d-dimensional version of the space filling curve has also the same hierarchical structure, where the basic shape (the Z of the 2D case) is defined as a connected pair of $(d-1)$-dimensional basic shapes lying on the opposite faces of a d-dimensional cube.

The property that makes the Lebesgue's space filling curve particularly attractive is the easy conversion from the d indices of a d-dimensional matrix to the 1D index along the curve. If one element e has d-dimensional reference $(i_1, \ldots, i_d)$, its 1D reference is built by interleaving the bits of the binary representations of the indices $i_1, \ldots, i_d$. In particular if i_j is represented by the string of h bits "$b_j^1 b_j^2 \cdots b_j^h$" (with $j = 1, \ldots, d$) then the 1D reference I of e is represented by the string of hd bits $I =$ "$b_1^1 b_2^1 \cdots b_d^1 b_1^2 b_2^2 \cdots b_d^2 \cdots b_1^h b_2^h \cdots b_d^h$".

The 1D order can be structured in a binary tree by considering elements of level i, those that have the last i bits all equal to 0. This yields a hierarchy where each level of resolution has twice as many points as the previous level. From a geometric point of view this means that the density of the points in the d-dimensional grid is doubled alternating along each coordinate axis. Figure 7.5 shows the binary hierarchy in the 2D case where the resolution of the space filling curve is doubled alternately along the x and y axis. The coarsest level (a) is a single point, the second level (b) has two points , the third level (c) has four points (forming the Z shape), and so on.

Index Remapping. The cardinality function discussed in Section 7.3.2.1 for the binary tree case has the structure shown in Table 7.2. For example, the element of index 0 is always at the top of the tree (level 0) for any granularity. If the index as 5 granularity levels (the 4th row in the table), node 8 is at the second level the tree; the nodes 4 and 12 are at the next level of the tree; the node 4 has nodes 2 and 6 below it at the next level of the tree, while the node 12 has 10 and 14 below it at the next level of the tree. The last level has nodes 1 and 3 below 2, 5 and 7 below 6 and so on. Note that this is a general structure suitable for out-of-core storage of static binary trees. It is independent of the dimension d of the grid of points or of the Z-order space filling curve.

The structure of the binary tree defined on the Z-order space filling curve allows one to easily determine the three elements necessary for the computation of the cardinality. They are: (i) the level i of an element, (ii) the constants C_i of equation (7.1) and (iii) the local indices I_i''.

i - if the binary tree hierarchy has k levels then the element of Z-order index j in the Z-order belongs to the level $k-h$, where h is the number of trailing zeros in the binary representation of j;

C_i - the total number of elements in the levels coarser than i, with $i > 0$, is $C_i = 2^{i-1}$ with $C_0 = 0$;

I_i'' - if an element has index j and belongs to the set S_i' then $\frac{j}{2^{k-i}}$ must be an odd number, by definition of i. Its local index is then:

$$I_i''(j) = \left\lfloor \frac{j}{2^{k-i+1}} \right\rfloor .$$

The computation of the local index I_i'' can be explained easily by looking at the bottom right part of Table 7.2 where the sequence of indices (1, 3, 5, 7, 9, 11, 13, 15) needs to be remapped to the local index (0, 1, 2, 3, 4, 5, 6, 7). The original sequence is made of a consecutive series of odd numbers. A right shift of one bit (or rounded division by two) turns them into the desired index.

These three elements can be put together to build an efficient algorithm that computes the hierarchical index $I'(s) = C_i + I_i''(s)$ in the two steps shown in the diagram of Figure 7.6:

1. Set the bit in position $k+1$ to 1;
2. Shift to the right until a 1 comes out of the bit-string.

This algorithm could have a very simple and efficient hardware implementation. The software C++ version can be implemented as follows:

```
inline adhocindex remap(register adhocindex i){
  i |= last_bit_mask;  // set leftmost one
  i /= i&-i;           // remove trailing  zeros
  return (i>>1);       // remove rightmost one
}
```

This code would work only on machines with two's complement representation of numbers. In a more portable version, one needs to replace `i /= i&-i` with `i /= i&((~i)+1)`.

Figure 7.7 shows the data layout obtained for a 2D matrix when its elements are reordered following the index I'. The data is stored in this order and divided into blocks of constant size. The 2D image of such decomposition has the first block corresponding to the coarsest level of resolution of the data. The subsequent blocks correspond to finer and finer resolution data, which is distributed more and more locally.

7.3.2.3 Performance

In this section, we describe experimental results for a simple, fundamental visualization technique: orthogonal slicing of a 3D rectilinear grid. Slices can be at different resolutions to allow interactivity: as the user manipulates the slice parameters, we compute and display a coarse resolution slice, then refine it progressively. We compare our layout with two common array layouts: row major, and $h \times h \times h$ brick decomposition.

Data I/O Requirements. As we shall see, the amount of data required to be read from disk varies substantially from one array layout to another. By way of example, consider the case of an 8GB dataset (a 2048^3 mesh of `unsigned char` data values). An orthogonal slice of this mesh consists of 2048×2048, or 4194304 points/bytes. In this example, disk pages are 32KB in size (see Figure 7.8(a)). For the brick decomposition case, one would use $32 \times 32 \times 32$ blocks of 32KB for the entire dataset. The data loaded from disk for a slice is 32 times larger than the output, or 128MB bytes. As the subsampling increases up to a value of 32 (one sample out of 32), the amount of data loaded does not decrease because each $32 \times 32 \times 32$ brick needs to be completely loaded. At lower subsampling rates, the data overhead remains the same: the data loaded is 32768 times larger than the data needed. In the binary tree with Z-order remapping, the data layout is equivalent to a KD-tree, constructing the same subdivision as an octree. For a 2D slice, the KD-tree mapping is equivalent to a quadtree layout. The data loaded is grouped into blocks along the hierarchy that gives an overhead factor in number of blocks of $1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} + \cdots < 2$ (as for one added to a geometric series), while each block is 32KB.

Tests with Memory Mapped Files. A series of basic tests were performed to verify the performance of the approach using a general purpose paging system. The out-of-core component of the scheme was implemented simply by mapping a 1D array of data to a file on disk using the `mmap` function. In this way, the I/O layer is implemented by the operating system virtual memory subsystem, paging in and out a portion of the data array as needed. No multi-threaded component is used to avoid blocking the application while retrieving the data. The blocks of data defined by the system are typically 4KB. Figure 7.8(b) shows performance tests executed on a Pentium III Laptop. The proposed scheme shows the best scalability in performance. The brick decomposition scheme with 16^3 chunks of regular grids shows the next best performance. The (i, j, k) row-major storage scheme has the worst performance because of its dependency on the slicing direction: best for (j, k) plane slices and worst for (j, i) plane slices. Figure 7.8(c) shows the performance results for a test on a larger, 8GB dataset, run on an SGI Octane. The results are similar.

The hierarchical Z-order is our layout of choice for efficient data management for visualizing and analyzing large scale scientific simulation data. In the next section, we describe some of the fundamental mathematical techniques

that we use for analyzing such datasets.

7.4 File Formats for High-Performance Visualization and Analytics

The notion of data models and formats is a central focal point in the nexus between producers and consumers of data. As indicated earlier in Section 7.2, production visualization applications are expected to be able to read and process a large number of different formats. Many of these formats are built atop underlying I/O libraries, like HDF5 and netCDF. Even with well-established I/O libraries, which implement an on-disk data format for storing and retrieving arrays of data, there exist several issues that perennially affect producers and consumers of data. One is the fact that the APIs for these I/O libraries can be complex: they provide a great deal of functionality, which is exposed through an API. Another is that it is possible to "misuse" the I/O library in a way that will result in less-than-optimal I/O performance. Yet another is that semantic conventions are not consistent across and within disciplines. The existence of well-established I/O libraries is of huge benefit to developers, for these technologies accelerate development by avoiding "reinventing the wheel." However, they can be complex to use, and they don't solve the semantic gap problem that exists between producers and consumers of data.

This section addresses these topics from a visualization-centric perspective. In our work, we often are the ones who end up having to reconcile semantic as well as format discrepancies between simulation, experiment and visualization technologies. As it does not seem practical for a panacea solution that will solve these problems for all disciplines, we have adopted a "bottom-up" approach that focuses on addressing these problems for communities and for specific new capabilities at the crossover point between the fields of visualization and scientific data management.

The first subsection below presents H5part, which is a high-level API that provides a solution to the semantic gap for use in computational accelerator modeling. Concurrently, H5part is engineered to provide good I/O performance in a way that is reasonably "immune to misuse." It encapsulates the complexity of an underlying I/O library, thereby providing advanced data I/O capabilities to both simulation and visualization developers. The relative simplicity of its API lowers the developer cost of taking advantage of such technology. Next, we present another high-level API aimed at encapsulating both data I/O and index/query capabilities. Again, our motivation is to encapsulate the complexity of both these technologies. Finally, we present some examples of how these technologies are used in practice.

7.4.1 H5Part

7.4.1.1 Motivation

Modern science – both computational and experimental – is typically performed by a number of researchers from different organizations who collectively collaborate on a challenging research problem. In the case of particle accelerator modeling, different groups collaborate on different aspects of modeling the entire beamline. One group works on modeling injection of particles into the beam, another works on modeling magnetic confinement and beam focusing along the beamline, while yet another works on modeling the impact of the energized particles with the target. Each of these different models is studied with one or more different simulation codes. One "grand challenge" in accelerator modeling is to simulate the entire beamline, from injection to impact. This ambitious project is being approached by the development and use of a number of different codes, each of which models a different portion of the beamline[4]. Accurate end-to-end modeling of accelerators is crucial to create an optimized design prior to building the multi-billion dollar instrument.

Unfortunately, there has historically been little coordination between these code teams: the individual codes often output particle and field data in different, incompatible formats. Worse yet, some of these codes output multiple terabytes of simulation data in ASCII format because it is "easy" to do so. Many of these codes use serial I/O; e.g., ASCII, HDF4, etc. As a result, there is data format incompatibility between different modeling stages, I/O is slow when performed in serial, and is wasteful of space in the case of ASCII formats. Of more concern is the impediment to scientific progress that results from data format incompatibility. It becomes extremely difficult to compare results from different simulations that model the same part of the beamline due to different data formats (e.g, different units of measure, different coordinate systems, different data layouts, and so forth).

H5part[5] was motivated by the desire of the accelerator modeling community to address these problems. The vision is to have a community-centric data format and API that would enable codes from all processing stages to: (1) read/write particle and field data in a single format; (2) allow legacy codes – simulations, visualization and analysis tools – to quickly take advantage of modern parallel I/O capabilities on HPC platforms; (3) accelerate software development and engineering in the area of I/O for accelerator modeling codes; (4) facilitate code-to-code and code-to-experiment data comparison and analysis; (5) enable the accelerator modeling community to more easily share data, simulation and analysis code; and (6) facilitate migrating towards community standards for data formats and analysis/visualization. The intended result is

[4]See `http://compass.fnal.gov` for an example of a large, community-based accelerator modeling project with exactly these objectives.

[5]We coined the term H5part for the API to reflect use of HDF5 for particle-based data sets common in high energy physics.

to accelerate scientific discovery by reducing software development time, by fostering "best practices" in data management within the particle accelerator modeling community, and to improve the I/O efficiency of simulation and analysis tools.

H5Part is a very simple data storage schema and provides an API that simplifies the reading/writing of the data to the HDF5 file format [13]. An important foundation for a stable visualization and data analysis environment is a stable and portable file storage format and its associated APIs. The presence of a "common file storage format," including associated APIs, fosters a fundamental level of interoperability across the project's software infrastructure. It also ensures that key data analysis capabilities are present during the earliest phases of the software development effort. The H5Part file format and APIs enable disparate research groups with different simulation implementations to transparently share datasets and data analysis tools. For instance, the common file format enables groups that depend on completely different simulation implementations to share data analysis tools.

H5Part is built on top of HDF5 (Hierarchical Data Format). HDF5 offers a number of advantages: it is a self-describing machine-independent binary file format that supports scalable parallel I/O performance for MPI codes on a variety of supercomputing systems, and works equally well on laptop computers. HDF5 is available for C, C++, and Fortran codes. The primary disadvantage of HDF5 is in the complexity of the API. Because of the rich set of functionality in HDF5, it can be challenging for domain scientists to write out a simple 1D array of data using raw HDF5 calls. Worse, doing parallel I/O is further complicated by the variety of data layout and I/O tuning options available in HDF5.

By restricting the usage scenario to particle accelerator data, H5Part encapsulates much of the complexity of HDF5 to present a simple interface for data I/O to accelerator scientists that is much easier to use than the HDF5 API. Compared to code that calls HDF5 directly, code that calls H5part is more terse, less complex, and easier to maintain. For example, code that uses H5part need to make any HDF5 calls that set up organization for data groups inside the HDF5 file since it encapsulates such functionality. The internal layout of data groups inside the HDF5/H5part file has proven to be effective for both efficiently writing data from simulation code as well as for efficiently reading data, either serially or in parallel, into visual data analysis tools.

7.4.1.2 File Organization and API

The H5part file storage format uses HDF5 for the low-level file storage and a simple API to provide a high-level interface to that file format. A programmer can use the H5Part API to access the data files or to write directly to the file format using some simple conventions for organizing and naming the objects stored in the file.

In order to store particle data in the HDF5 file format, we have formalized

the hierarchical arrangement of the datasets and naming conventions for the groups and associated datasets. The H5Part API formally encodes these conventions in order to provide a simple and uniform way to access these files from C, C++, and Fortran codes. The API makes it easy to write very portable data adaptors for visualization tools in order to expand the number of tools available to access the data. Users may write their own HDF5-based interface for reading and writing the file format, or may use the *h5ls* and *h5dump* command-line utilities, which are included with the HDF5 distribution, to display the organization and contents of H5Part files. The standards offered by the sample API are completely independent of the standard for organizing data within the file. The file format supports the storage of multiple timesteps of datasets that contain multiple fields.

The data model for particle data allows storing multiple timesteps where each timestep can contain several datasets of the same length. Typical particle data consists of the three-dimensional Cartesian positions of particles (x, y, z) as well as the corresponding three-dimensional momenta (px, py, pz). These six variables are stored as six HDF5 datasets. The type of the dataset can be either integer or real. H5Part also allows storing attribute information for the file and timesteps.

Table 7.3 presents sample H5Part code for storing particle data with two timesteps. The resulting HDF5 file with two timesteps is shown in Table 7.4. These examples show the simplicity of an application that uses the H5Part API to write or read H5part files. One point is that there is basically a one-line difference between serial and parallel code. Another is that the H5Part application is much simpler than an HDF5-only counterpart: this example code need not worry about setting up data groups inside HDF5; that task is performed inside the H5Part library.

7.4.1.3 Parallel I/O

A näive approach to writing data from a parallel program is to write one file per processor. While this approach is simple to implement and very efficient on most cluster filesystems, it leads to file management difficulties when the data needs to be analyzed. One must either recombine these separate files into a single file or create unwieldy user-interfaces that allows a data analysis application to read from a directory full of files instead of just one file. An arguably better approach is to provide the means for a parallel application to write data into a single file from all PEs, which is known as *collective I/O*. Collective I/O performance is typically (but not always) lower than that of writing one file per processor, but it makes data management much simpler after the program has finished. No additional recombininations steps are required to make the file accessible by visualization tools or for restarting a simulation using a different number of processors.

Parallel HDF5 uses MPI-I/O for its low-level implementation. The mechanics of using MPI-I/O are hidden from the user by the H5Part API (the

```
if(serial)
   handle=H5PartOpenFile(filename, mode);
else
   handle=H5PartOpenFileParallel(filename, mode, mpi_comm);
H5PartSetNumParticles(handle, num_particles);
loop(step=1,2)
   // compute data
   H5PartSetStep(handle, step);
   H5PartWriteDataFloat64(handle,"px",data_px);
   H5PartWriteDataFloat64(handle,"py",data_py);
   H5PartWriteDataFloat64(handle,"pz",data_pz);
   H5PartWriteDataFloat64(handle, "x",data_x);
   H5PartWriteDataFloat64(handle, "y",data_y);
   H5PartWriteDataFloat64(handle, "z",data_z);
H5PartCloseFile(handle);
```

Table 7.3: Sample H5Part code to write multiple fields from a time-varying simulation to a single file.

```
GROUP "/" {
   GROUP "Step#0" {
      DATASET "px" {
         DATATYPE H5T_IEEE_F64LE
         DATASPACE SIMPLE { ( 1000 ) / ( 1000 ) }
      }
      DATASET "py" {
         DATATYPE H5T_IEEE_F64LE
         DATASPACE SIMPLE { ( 1000 ) / ( 1000 ) }
      }
      DATASET "pz" {
         DATATYPE H5T_IEEE_F64LE
         DATASPACE SIMPLE { ( 1000 ) / ( 1000 ) }
      }
      DATASET "x" {
         DATATYPE H5T_IEEE_F64LE
         DATASPACE SIMPLE { ( 1000 ) / ( 1000 ) }
      }
      DATASET "y" {
         DATATYPE H5T_IEEE_F64LE
         DATASPACE SIMPLE { ( 1000 ) / ( 1000 ) }
      }
      DATASET "z" {
         DATATYPE H5T_IEEE_F64LE
         DATASPACE SIMPLE { ( 1000 ) / ( 1000 ) }
      }
   }
   GROUP "Step#1" {
      ...information for 6 datasets...
   }
}
```

Table 7.4: Contents of the H5Part file generated in Table 7.3.

code looks nearly identical to reading/writing the data from a serial program). While the performance is not always as good as writing one-file per-processor, we have shown that writing files with Parallel HDF5 is consistently faster than writing the data in raw/native binary using the MPI-I/O library [2]. This efficiency is made possible through sophisticated HDF5 tuning directives, which are transparent to the parallel application, that control data alignment and caching within the HDF5 layer. Therefore, we argue that it would be difficult to match HDF5 performance even using a home-grown binary file format.

7.4.2 HDF5_FastQuery

Large scale scientific data is often stored in scientific data formats like FITS, netCDF and HDF. These storage formats are of particular interest to the scientific user community since they provide multi-dimensional storage and retrieval capabilities. However, one of the drawbacks of these storage formats is that they do not support the ability to extract subsets of data that meet multidimensional, compound range conditions. Such multidimensional range conditions are often the basis for defining "features of interest," which are the focus of scientific inquiry and study.

HDF5_FastQuery [12] is a high-level API that provides the ability to perform multi-dimensional indexing and searching on large HDF5 files. It leverages an efficient bitmap indexing technology called "FastBit" [14, 38, 39] (described in Chapter 10) that has been widely used in the database community. Bitmap indices are especially well suited for interactive exploration of large-scale read-only data. Storing the bitmap indices into the HDF5 file has the following advantages: a) Significant performance speedup of accessing subsets of multi-dimensional data and b) portability of the indices across multiple computer platforms. The HDF5_FastQuery API simplifies the execution of queries on HDF5 files for general scientific applications and data analysis. The design is flexible enough to accommodate the use of arbitrary indexing technology for semantic range queries.

HDF5_FastQuery provides an interface to support semantic indexing for HDF5 via a query API. HDF5-FastQuery allows users to efficiently generate complex selections on HDF5 datasets using compound range queries like $(energy > 10^5)$ AND $(70 < pressure < 90)$ and retrieve only the subset of data elements that meet the query conditions. The FastBit technology generates the compressed bitmap indices that accelerate searches on HDF5 datasets, as well as the raw indices (the compressed bitmap indices), which are stored together with the datasets in an HDF5 file. Compared with other indexing schemes, compressed bitmap indices are compact and very well suited for searching over multi-dimensional data even for arbitrarily complex combinations of range conditions.

7.4.2.1 Functionality

HDF5 supports slab and hyper-slab selections of n-dimensional datasets. HDF5_FastQuery extends the HDF5 selection mechanism to allow subset selection based upon arbitrary range conditions on the data values contained in the datasets using the bitmap indices. As a result, HDF5_FastQuery supports fast execution of searches based upon compound queries that span multiple datasets. The API also allows us to seamlessly integrate the FastBit query mechanism for data selection with HDF5s standard hyper-slab selection mechanism. Using the HDF5_FastQuery API, one can quickly select subsets of data from an HDF5 file using text-string queries.

The bitmap indices are created and stored through a single call to the HDF5-FastQuery API. The storage of these indices uses separate arrays in the same file as the datasets they refer to and are opaque to the general HDF5 functions. It is important to note that all such indices must be built before any queries are posed to the API. Once the bitmap indices have been built and stored in the data file, queries are posed to the API as a text-string such as $(temperature > 1000)$ AND $(70 < pressure < 90)$, where the names specified in the range query correspond to the names of the datasets in the HDF5 file. The HDF5_FastQuery interface uses the stored bitmap indices that correspond to the specified dataset to accelerate the selection of elements in the datasets that meet the search criteria. An accelerated query on the contents of a dataset requires only small portions of the compressed bitmap indices to be read into memory, so extremely large datasets can be searched with little memory overhead. The query engine then generates an HDF5 selection that is used to read only the elements from the dataset that are specified by the query string.

7.4.2.2 Architectural Layout

In this section, we present a high-level view of the HDF5-FastQuery architectural layout. We begin by defining relevant terms used throughout the architectural layout as well as the HDF5-FastQuery API.

Groups are the logical way to organize data in an HDF5 file. We use the term group or grouping to refer to this logical structuring. These groups act as containers of various types of metadata, which in our approach are specific to a given dataset. Note that these groups may be assigned type information (float, int, string etc.) to uniquely describe these datasets.

Variables vs. Attributes. The properties assigned to a specific group (i.e. group metadata) are called attributes or group attributes. For all datasets, the specific physical properties that the dataset quantizes (density, pressure, helicity etc.) will be referred to as dataset variables. To organize a given multivariate dataset consisting of a discrete range of time steps, a division is made between the raw data and the attributes that describe the data. This division is represented in the architectural layout by the separation and formation of two classes of groups: the *TimeStep* groups for the raw data, and

the *VariableDescriptor* groups for the metadata used to describe the dataset variables.

For the dataset variables, one *VariableDescriptor* group is created for each variable (pressure, velocity etc.). The metadata saved under these groups usually includes the size of the data set; name of the dataset variable; coordinate system used in the dataset (spherical, Cartesian etc.); the schema (structured, unstructured, etc); Centering (cell centered, vertex centered, edge centered etc.) and number of coordinates which must exist per centering element (each vertex, each face etc.).

The various *VariableDescriptor* groups are then organized under one TOC (table of contents) group that retains common global information about the file's variables (the names of all variables, bitmap indices metadata information). For the raw datasets, a unique *TimeStep* group is created for each time step in the discrete time range. Under each *TimeStep* group exists one HDF5 dataset that contains the raw data for a given variable at that time step. The bitmap dataset corresponding to the variable is also stored under the same *TimeStep* group.

This division between data and metadata is essential for the primary reason that variable metadata for a given dataset is relevant and accurate across all time steps for that dataset variable (there is no need to store redundant metadata). Figure 7.9 illustrates the HDF5-FastQuery architectural layout.

7.4.3 Status and Sample Applications

H5part research and development is an active, international collaborative effort involving researchers from high performance computing and accelerator modeling. It has evolved from focusing on I/O for particle-based datasets to include support for block-structured fields and unstructured meshes. It is an open source project, and source code may be downloaded from `https://codeforge.lbl.gov/projects/h5part`. Several accelerator modeling codes use H5part for writing data in parallel, and H5part readers have been created and integrated into visual data analysis applications in widespread use by the accelerator modeling community (see Figures 7.10 and 7.11).

Recent efforts have focused on merging the capabilities of HDF5-FastQuery, which provides a high-level API interface to advanced index/query capabilities, into H5part. The objective here is to encapsulate the complexity of index/query APIs and to integrate that capability into what appears to the application as an I/O layer. This work has proven very useful in enabling rapid visual data exploration in several projects, including advanced accelerator design using laser wakefield acceleration (see Figure 7.12). Here, the tightly integrated index/query and I/O provides the fundamental scientific data management infrastructure needed to implement query-driven visualization: a visual interface enables rapid exploration of high-level data characteristics, enables visual specification of "interesting" data subsets, which are then quickly extracted from a very large dataset and used by downstream visual

data analysis tools. The index/query capability has been implemented inside the H5part layer to run in parallel to take advantage of high-performance computing platforms to accelerate I/O and range-based subsetting to support interactive exploration.

7.5 Query-Driven Visualization

The term "query-driven visualization" (QDV) refers to the process of limiting visual data analysis processing only to "data of interest [35]." In brief, QDV is about using software machinery combined with flexible and highly useful interfaces to help reduce the amount of information that needs to be analyzed. The basis for the reduction varies from domain to domain, but boils down to "what subset of the large dataset is really of interest for the problem being studied." This notion is closely related to that of "feature detection and analysis," where "features" can be thought of as subsets of a larger population that exhibit some characteristics that are either intrinsic to individuals within the population (e.g., data points where there is high pressure and high velocity) or that are defined as relations between individuals within the population (e.g, the temperature gradient changes sign at a given data point).

QDV is one approach to visual data analysis of problems that are of massive scale in size. Other common approaches focus on increasing capacity of the visualization processing pipeline through increasing levels of parallelism to scale up existing techniques to accommodate larger data sizes. While effective in the primary objective – increase capacity to accommodate larger problem sizes – there is a fundamental problem with these approaches: they don't necessarily increase the likelihood of scientific insight. By processing more data and creating an image that is more complex, such an approach can actually impede scientific understanding.

Let's examine the first question a bit more closely. First, let's assume that we're operating on a gigabyte-sized dataset (10^9 data points), and we're displaying the results on a monitor that has, say, 2 million pixels ($2 * 10^6$ pixels). For the sake of discussion, let's assume we're going to create and display an isosurface of this dataset. Studies have shown that on the order of about $N^{2/3}$ grid cells in a dataset of size N^3 will contain any given isosurface [3]. In our own work, we have found this estimate to be somewhat low – our results have shown the number to be closer to $N^{0.8}$ for N^3 data. Also, we have found an average of about 2.4 triangles per grid cell will result from the isocontouring algorithm [5]. If we use these two figures as lower and upper bounds, then for our gigabyte-sized dataset, we can reasonably expect on the order of between about 2.1 and 40 million triangles for many isocontouring levels. At a display resolution of about 2 million pixels, the result is a depth

complexity – the number of objects at each pixel along all depths – of between one and twenty.

With increasing depth complexity come at least two types of problems. First, more information is "hidden from view." In other words, the nearest object at each pixel hides all the other objects that are further away. Second, if we do use a form of visualization and rendering that supports transparency – so that we can, in principle, see all the objects along all depths at each pixel – we are assuming that a human observer will be capable of distinguishing among the objects in depth. At best, this latter assumption does not always hold true, and at worst, we are virtually guaranteed the viewer will not be able to gain any meaningful information from the visual information overload.

If we scale up our dataset from gigabyte (10^9) to terabyte (10^{12}), we then can expect on the order of between 199 million and 9.5 billion triangles representing a depth complexity ranging between about 80 and 4700, respectively. Regardless of which estimate of the number of triangles we use, we end up drawing the same conclusion: depth complexity, and correspondingly scene complexity and human cognitive workload all grow at a rate that is a function of the size of the source data. Even if we are able to somehow display all those triangles, we would be placing an incredibly difficult burden on the user. They will be facing the impossible task of trying to visually locate "smaller needles in a larger haystack."

The multi-faceted approach we're adopting takes square aim at the fundamental objective: help scientific researchers more quickly and efficiently do science. In one view, one primary tactical approach that seems promising is to help focus user attention on easily consumable images from the large data collection. We do not have enough space in this chapter to cover all aspects in this regard. Instead, we provide a few details about a couple of especially interesting challenge areas.

7.5.1 Implementing Query-Driven Visualization

In principle, the QDV idea is conceptually quite simple: restrict visualization and analysis processing only to data of interest. In practice, implementing this capability can be quite a challenge. Our approach is to take advantage of the state-of-the-art index/query technology mentioned earlier in this chapter, FastBit, and use it as the basis for data subsetting in query-driven visualization applications. In principle, any type of data subsetting technology can be used to provide the same functionality. In practice, FastBit has proven to be very efficient and effective at problems of scale.

Some of our early work in this space focused on comparing the performance of FastBit as the basis for index/query in QDV applications with some of the "industry standard" algorithms for isosurface computation [35]. In computing isosurfaces, one must first find all the grid cells that contain the surface of interest, then for each such cell, compute the isosurface that intersects the cell. Our approach, which leverages FastBit, shows a performance gain of between

25% to 300% over the best isocontouring algorithms created by the visualization community. Of greater significance is the fact that our approach extends to $n-$dimensional queries (i.e., queries of n different variables), whereas the indexing structures created for use in isocontouring are applicable only to univariate queries. This approach was demonstrated on datasets created by a combustion simulation (see Figure 7.13, which shows three different query conditions performed on a single dataset. See [35] for more details.).

7.5.2 Case Study – Network Traffic Analysis

While our earlier work established the viability of the approach, particularly when compared to the best search algorithms from the visualization community, more recent work extends and applies these techniques to a "hero-sized" problem. In this application, our objective is to perform interactive visual data analysis of one year's worth of network connection data. The case study in this work focuses on rapid drill-down using multiresolution histograms computed by FastBit for the purposes of identifying the existence of a distributed network scan attack, then for identifying the set of hosts participating in the attack. The results of that study (see [4,34]) show that this approach performs up to four orders of magnitude faster than conventional techniques commonly used in the field of network traffic analysis.

The basic use model for this application, which is shown pictorially in Figure 7.14, is as follows: first, compute and display a histogram of traffic levels at a coarse granularity (per-day over a 365-day period). Histogram display is augmented with statistical analysis to help highlight anomalous behavior. Next, through a visual user interface, "drill into" the data by allowing the user to specify a temporal window of finer resolution. FastBit computes a new histogram over the specified temporal window and at finer resolution, allowing more details of the data to emerge. This process repeats until coherent temporal patterns of the attack begin to emerge. Once the attack signature is identified and confirmed to be a network scan (see Figure 7.15), FastBit can quickly locate and return the network traffic records that contain information about hosts participating in the attack (see [34] for more details).

In this work, FastBit was extended to support the rapid creation of multidimensional, conditional histograms. The implementation and performance study was conducted on a parallel, shared-memory platform. The results of the study show that forensic investigation of such massive datasets can be conducted in an interactive fashion. Previous approaches would require hours or days to conduct a similar investigation. The significance of this work is that it shows the potential of coupling state-of-the-art scientific data management technology with visual data analysis technology to tackle problems of scale in a manner that virtually guarantees scientific insight and knowledge discovery.

7.6 Summary and Conclusion

Visualization, which is the transformation of abstract data into images, plays a central role in virtually all fields of scientific endeavor. It is an indispensable part of hypothesis testing and knowledge discovery. Like most other fields discussed in this book, visualization faces substantial scientific data management challenges that are the result of growth in size and complexity of data being produced by simulations and collected from experiments. Our objective in this chapter has been to reveal some of the scientific data management issues, challenges, and solutions that are somewhat unique to the field of visualization.

Production visualization applications, namely those that run on large-scale parallel machines, can derive great benefit from close attention to scientific data management issues. Staple operations, like slicing and isosurfacing, can be vastly accelerated by taking advantage of meta-data. In some cases, the simulation or experiment produces such meta-data as part of the data production process. In other cases, we must generate that data ourselves. A significant fraction of the code in these production visualization applications is dedicated to scientific data management: they must support a plethora of input data formats, and therefore, they contain a number of data loader modules; they must create an internal data structure that is suitable for use by a potentially large collection of visualization, analysis and rendering modules, all of which may potentially run in parallel on shared or distributed memory machines. An open problem is one of data models and semantics, where meaning is assigned to arrays of data stored in data files.

With data of massive scale, it is often useful to perform a multiresolution analysis, working first with a smaller, coarser version of data, then progressively refine the analysis as interesting features are revealed. We saw that a space-filling curve model has proven to be highly efficient for interactive analysis of massive data. However, such a data model and layout is unlikely to be output directly from a simulation. This is a good example of how a data model and layout that works very well for multiresolution analysis is unlikely to be used by simulations for output. Multiresolution, quantitative feature detection and analysis methods were demonstrated on two different datasets from the field of turbulent mixing. This quantitative analysis approach is very useful for enabling scientific knowledge discovery by focusing on features rather than on the machinery for creating potentially incomprehensible images of large-scale scientific data.

A significant barrier faced by many computational and experimental science projects is the complexity of using state-of-the art technology from scientific data management. We discussed an approach for encapsulating complexity in the form of a high-level API for data storage and retrieval that lowers the entry point for using such technology. This concept was also applied to index/query

technology. We have applied these concepts to multiple application areas to produce results showing that visual data analysis, as a field, can benefit from a close collaboration with the field of scientific data management. The benefit is improved performance for many data intensive operations, like data I/O and data subsetting, as well as the potential to conceive and create completely new, paradigm-changing approaches to solve the problem of scientific knowledge discovery for massive, complex datasets.

7.7 Acknowledgments

This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231 through the Scientific Discovery through Advanced Computing (SciDAC) program's Visualization and Analytics Center for Enabling Technologies (VACET). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] Greg Abram and Lloyd A. Treinish. An extended data-flow architecture for data analysis and visualization. Research report RC 20001 (88338), IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, February 1995.

[2] Andreas Adelmann, Achim Gsell, B. Oswald, T. Schietinger, E. Wes Bethel, John Shalf, Cristina Siegerist, and Kurt Stockinger. Progress on H5Part: A Portable High Performance Parallel Data Interface for Electromagnetic Simulations. In *Particle Accelerator Conference PAC07 25–29 June 2007, Albuquerque NM*, 2007. http://vis.lbl.gov/Publications/2007/LBNL-63042.pdf.

[3] Chandrajit L. Bajaj, Valerio Pascucci, and Daniel R. Schikore. Fast isocontouring for improved interactivity. In *VVS '96: Proceedings of the 1996 symposium on Volume visualization*, pages 39–ff., Piscataway, NJ, USA, 1996. IEEE Press.

[4] E. Wes Bethel, Scott Campbell, Eli Dart, Kurt Stockinger, and Kesheng Wu. Accelerating Network Traffic Analysis Using Query-Driven Visualization. In *Proceedings of 2006 IEEE Symposium on Visual Analytics Science and Technology*, pages 115–122. IEEE Computer Society Press, October 2006. LBNL-59891.

[5] Ian Bowman, John Shalf, Kwan-Liu Ma, and E. Wes Bethel. Performance Modeling for 3D Visualization in a Heterogenous Computing Environment. Technical Report LBNL-56977, Lawrence Berkeley National Laboratory, 2004.

[6] P.-T. Bremer, H. Edelsbrunner, B. Hamann, and V. Pascucci. A multiresolution data structure for two-dimensional Morse-Smale functions. In G. Turk, J. J. van Wijk, and R. Moorhead, editors, *Proc. IEEE Visualization '03*, pages 139–146, Los Alamitos California, 2003. IEEE, IEEE Computer Society Press.

[7] P.-T. Bremer, H. Edelsbrunner, B. Hamann, and V. Pascucci. A topological hierarchy for functions on triangulated surfaces. *IEEE Trans. on Visualization and Computer Graphics*, 10(4):385–396, 2004.

[8] K. Cole-McLaughlin, H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci. Loops in reeb graphs of 2-manifolds. In *Proceedings of the 19th Annual Symposium on Computational Geometry*, pages 344–350. ACM Press, 2003.

[9] Computational Engineering International, Inc. *Ensight Visualization Software*, 2008. `http://www.ensight.com/`.

[10] H. Edelsbrunner, J. Harer, and A. Zomorodian. Hierarchical Morse-Smale complexes for piecewise linear 2-manifolds. *Discrete Comput. Geom.*, 30:87–107, 2003.

[11] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 454, Washington, DC, USA, 2000. IEEE Computer Society.

[12] Luke Gosink, John Shalf, Kurt Stockinger, Kesheng Wu, and E. Wes Bethel. HDF5-FastQuery: Accelerating Complex Queries on HDF Datasets using Fast Bitmap Indices. In *Proceedings of the 18th International Conference on Scientific and Statistical Database Management*. IEEE Computer Society Press, July 2006. LBNL-59602.

[13] HDF Group. Hierarchical Data Format – HDF5, 2008. `http://www.hdfgroup.com`.

[14] LBNL Scientific Data Management Research Group. FastBit: An Efficient Compressed Bitmap Index Technology, 2008. `http://sdm.lbl.gov/fastbit/`.

[15] 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.

[16] Christopher Johnson and Charles Hansen (eds.). *Visualization Handbook*. Academic Press, Inc., Orlando, FL, USA, 2004.

[17] Kitware, Inc. and Los Alamos National Laboratory and Sandia National Laboratory and CSimSoft. *ParaView Visualization Software*, 2008. `http://www.paraview.org/`.

[18] D. Laney, P. T. Bremer, A. Mascarenhas, P. Miller, and V. Pascucci. Understanding the structure of the turbulent mixing layer in hydrodynamic instabilities. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1053–1060, 2006.

[19] J K Lawder and P J H King. Using space-filling curves for multi-dimensional indexing. In *Lecture Notes in Computer Science*, pages 20–35, 2000.

[20] Lawrence Livermore National Laboratory. *VisIt Visualization Software*, 2008. `http://www.llnl.gov/visit/`.

[21] B. McCormick, T. Defanti, and M. Brown (eds.). Visualization in Scientific Computing. *Computer Graphics*, 21(6), 1987.

[22] Mark C. Miller, James F. Reus, Robb P. Matzke, William J. Arrighi, Larry A. Schoof, Ray T. Hitt, and Peter K. Espen. Enabling interoperation of high performance, scientific computing applications: Modeling scientific data with the sets & fields (saf) modeling system. In *ICCS '01: Proceedings of the International Conference on Computational Science-Part II*, pages 158–170, London, UK, 2001. Springer-Verlag.

[23] P. Miller, P.-T. Bremer, W. Cabot, A. Cook, D. Laney, A. Mascarenhas, and V. Pascucci. Application of morse theory to analysis of rayleigh-taylor topology. In *Proocedings of the 10th International Workshop on the Physics of Compressible Turbulent Mixing*, 2006.

[24] B. Moon, H. Jagadish, C. Faloutsos, and J. Saltz. Analysis of the clustering properties of hilbert spacefilling curve. *IEEE Transactions on knowledge and data engeneering*, 13(1):124–141, 2001.

[25] V. Natarajan, Y. Wang, P.-T. Bremer, V. Pascucci, and B. Hamann. Segmenting molecular surfaces. *Comput. Aided Geom. Des.*, 23(6):495–509, 2006.

[26] 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.

[27] 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.

[28] Brian Paul, Sean Ahern, E. Wes Bethel, Eric Brugger, Rich Cook, Jamison Daniel, Ken Lewis, Jens Owen, and Dale Southard. Chromium Renderserver: Scalable and Open Remote Rendering Infrastructure. *IEEE Transactions on Visualization and Computer Graphics*, 14(3), May/June 2008. LBNL-63693.

[29] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hop per. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[30] Lisa Roberts. *Silo User's Guide*. Lawrence Livermore National Laboratory, 2000. UCRL-MA-118751-REV-1.

[31] Larry Schoof and Vincent Yarberry. *EXODUS II: A Finite Element Data Model*. Sandia National Laboratory, 1994. Tech Rep. SAND92-2137.

[32] William J. Schroeder, Kenneth M. Martin, and William E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *VIS '96: Proceedings of the conference on Visualization '96*, pages 93–ff. IEEE Computer Society Press, 1996.

[33] John Shalf and E. Wes Bethel. How the Grid Will Affect the Architecture of Future Visualization Systems. *IEEE Computer Graphics and Applications*, 23(2):6–9, May/June 2003.

[34] Kurt Stockinger, E. Wes Bethel, Scott Campbell, Eli Dart, , and Kesheng Wu. Detecting Distributed Scans Using High-Performance Query-Driven Visualization. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, October 2006.

[35] Kurt Stockinger, John Shalf, Kesheng Wu, and E. Wes Bethel. Query-Driven Visualization of Large Data Sets. In *Proceedings of IEEE Visualization 2005*, pages 167–174. IEEE Computer Society Press, October 2005. LBNL-57511.

[36] Craig Upson, Thomas Faulhaber Jr., David Kamins, David H. Laidlaw, David Schlegel, Jeffrey Vroom, Robert Gurwitz, and Andries van Dam. The application visualization system: A computational environment for scientific visualization. *Computer Graphics and Applications*, 9(4):30–42, July 1989.

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

[38] Kesheng Wu, Ekow Otoo, and Arie Shoshani. On the performance of bitmap indices for high cardinality attributes. In *International Conference on Very Large Databases*, pages 24–35, 2004.

[39] Kesheng Wu, Ekow Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31:1–38, 2006.

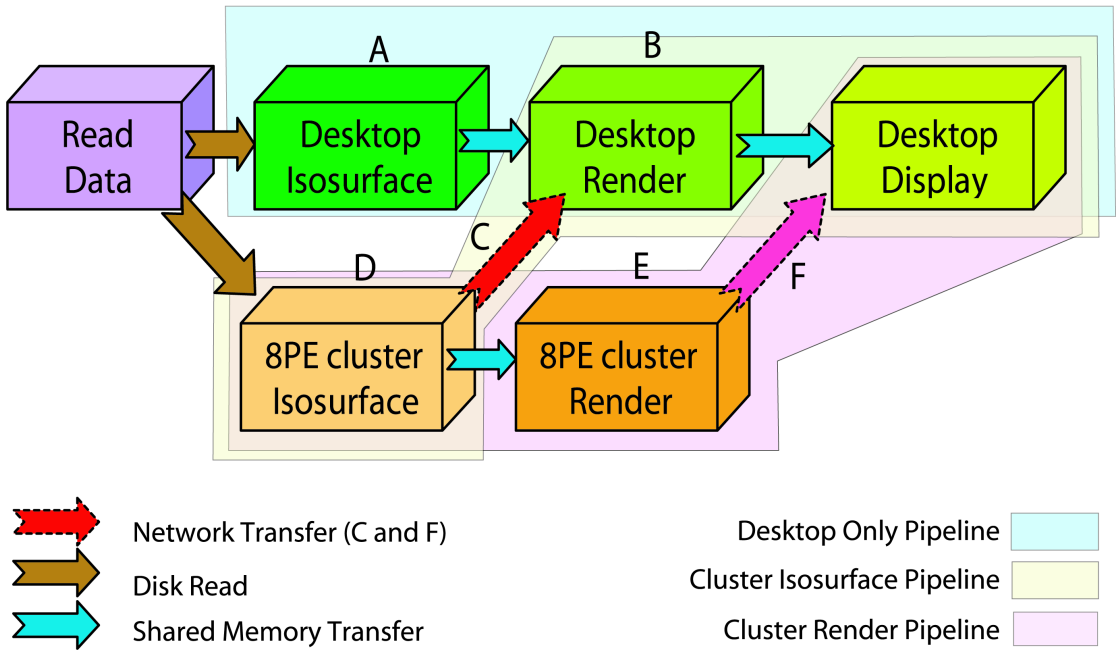


FIGURE 7.1: This image shows three different partitionings of a simple visualization pipeline consisting of four stages: data I/O, computing an isosurface, rendering isosurface triangles, and image display. In one partitioning, all operations happen on a single desktop machine (A-B, blue background). In another, data is loaded onto an eight-processor cluster for isosurface processing, and the resulting isosurface triangles are sent to the desktop for rendering (D-C-B, yellow background). In the third, data is loaded onto the cluster for isosurface processing and rendering, and the resulting image is sent to the desktop for display (D-E-F, magenta background).

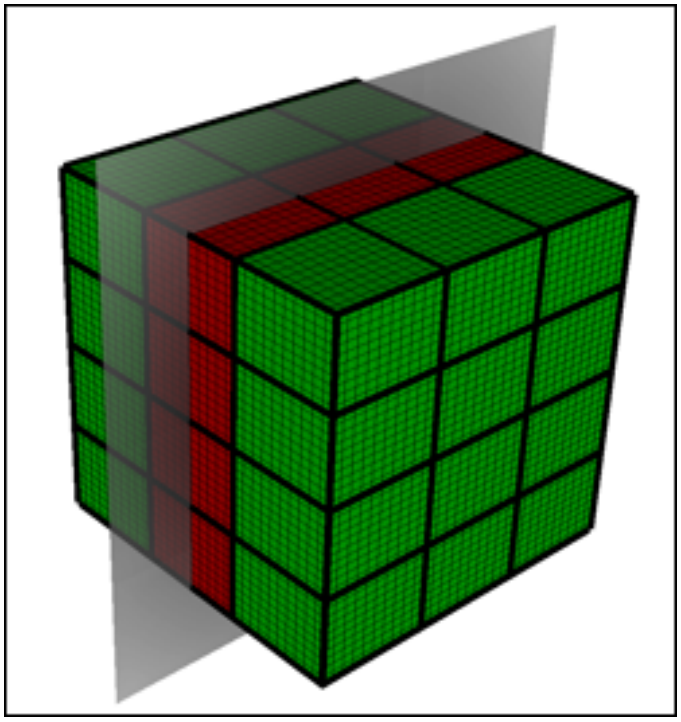


FIGURE 7.2: Shown is a 36 domain data set. The domains have thick black lines and are colored red or green. Mesh lines for the elements are also shown. To create the data set sliced by the transparent grey plane, only the red domains need to be processed. The green domains can be eliminated before ever being read in.

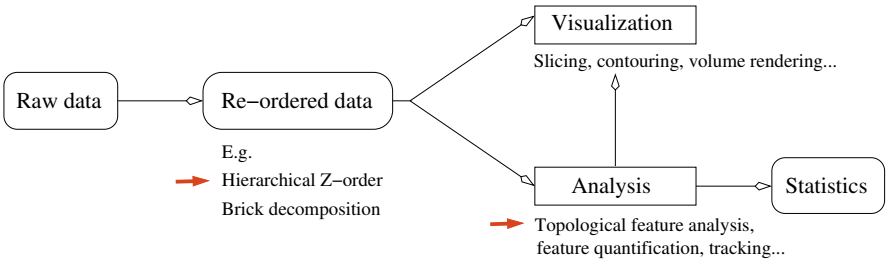


FIGURE 7.3: Hierarchical z-ordered data layout and topological analysis components highlighted in the context of a typical visualization and analysis pipeline.

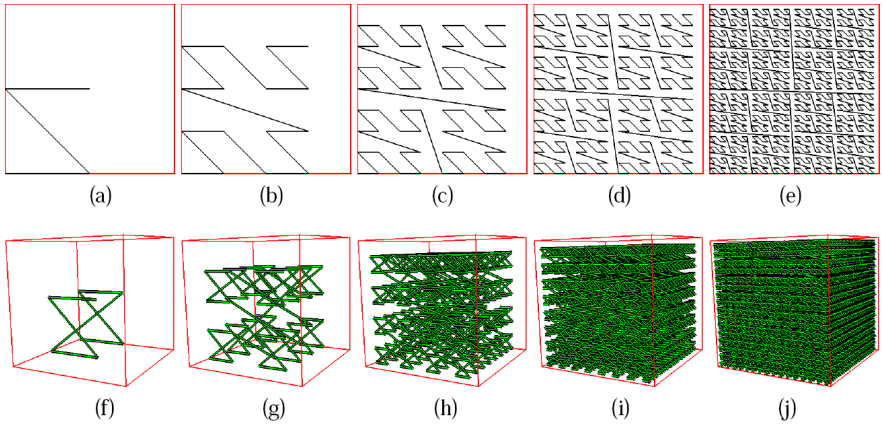


FIGURE 7.4: (a-e) The first five levels of resolution of the 2D Lebesgue's space filling curve. (f-j) The first five levels of resolution of the 3D Lebesgue's space filling curve.

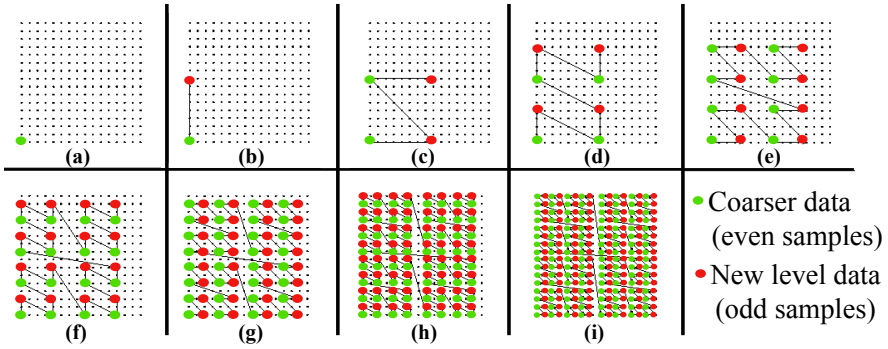


FIGURE 7.5: The nine levels of resolution of the binary tree hierarchy defined by the 2D space filling curve applied on 16×16 rectilinear grid. The coarsest level of resolution (a) is a single point. The number of points that belong to the curve at any level of resolution (b) to (i) is double the number of points of the previous level.

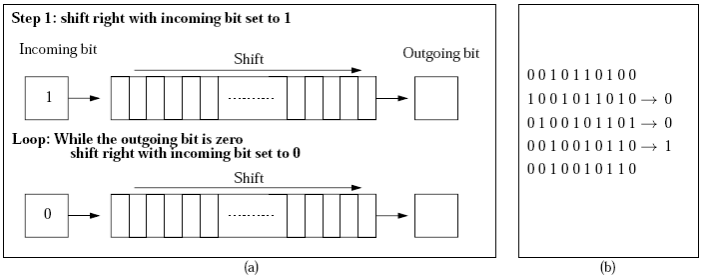


FIGURE 7.6: (a) Diagram of the algorithm for index remapping from Z-order to the hierarchical out-of-core binary tree order. (b) Example of the sequence of shift operations necessary to remap an index. The top element is the original index and the bottom is the remapped, output index.

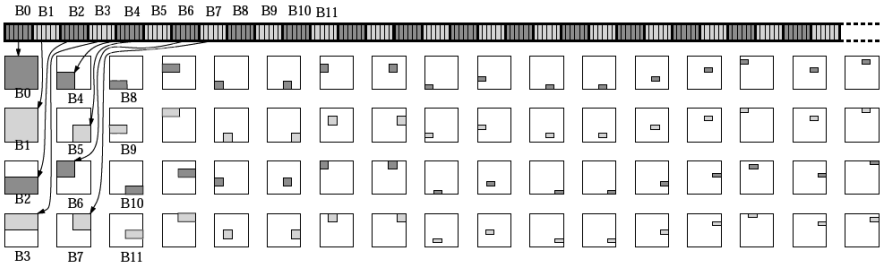
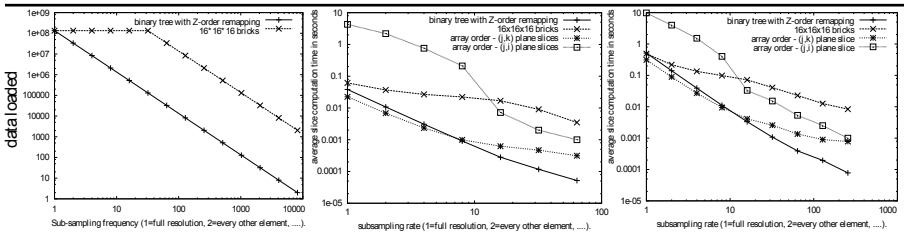


FIGURE 7.7: Data layout obtained for a 2D matrix reorganized using the index I' (1D array at the top). The 2D image of each block in the decomposition of the 1D array is shown below. Each gray region (odd blocks dark gray, even blocks light gray) shows where the block of data is distributed in the 2D array. In particular the first block is the set of coarsest levels of the data distributed uniformly on the 2D array.The next block is the next level of resolution still covering the entire matrix. The next two levels are finer data covering each half of the array. The subsequent blocks represent finer resolution data distributed with increasing locality in the 2D array.

static analysis	$1k \times 1k \times 1k$ Grid 500Mhz Intel PIII, 128MB RAM	$2k \times 2k \times 2k$ Grid 300Mhz MIPS R12000, 600MB RAM



(a) (b) (c)

FIGURE 7.8: Comparison of static analysis and real performance in different conditions. For an 8GB dataset, (a) compares the amount of data loaded from disk (vertical axis) per slice while varying the level of subsampling and using two different access patterns/storage layouts: Z-order remapping and brick decomposition. The values on the vertical axis are reported using a logarithmic scale to highlight the performance difference – orders of magnitude – at any level of resolution. (b-c) Two comparisons of slice computations times (log scale) of four different data layout schemes with slices parallel to the (j, k) plane (orthogonal to the x axis). The horizontal axis is the level of subsampling of the slicing scheme, where values on the left are finer resolution. Note how the practical performance of the Z-order versus brick layout is predicted very well by the static analysis. The only layout that can compete with the hierarchical Z-order the row-major array layout optimized for (j, k) slices (orthogonal to the x axis). Of course, the row-major layout performs poorly for (j, i) slices (orthogonal to the z axis), while the hierarchical Z-order layout would maintain the same performance for access at any orientation.

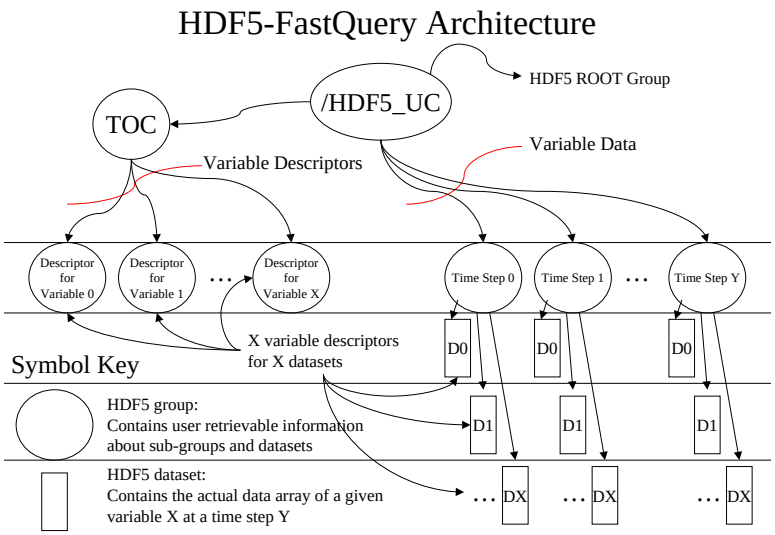
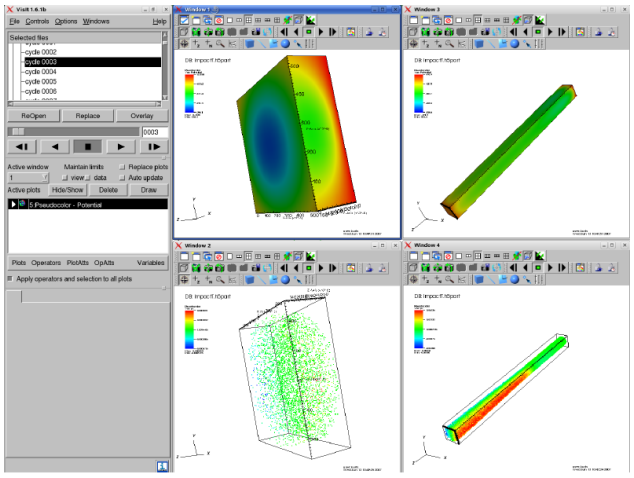
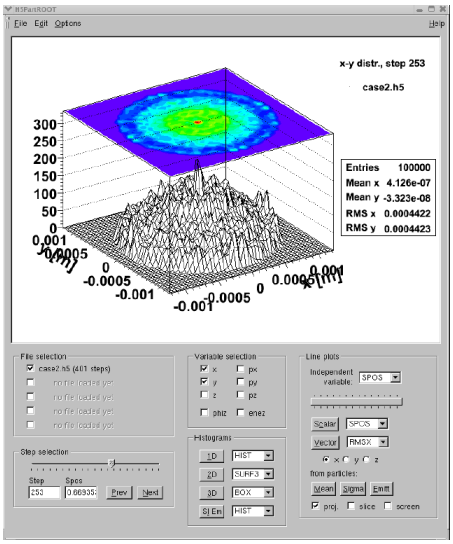


FIGURE 7.9: Architectural layout of HDF5-FastQuery.



(a)

FIGURE 7.10: H5part readers are included with visualization and data analysis tools in use by the accelerator modeling community. This image shows H5part data loaded into VisIt for comparative analysis of multiple variables at different simulation timesteps.



(a)

FIGURE 7.11: Root, a freely available, open source system for data management and analysis, is in widespread use by the high energy physics community. An H5part reader is included with Root. This image shows an example of visual data analysis of an H5part dataset produced by a particle accelerator modeling code.

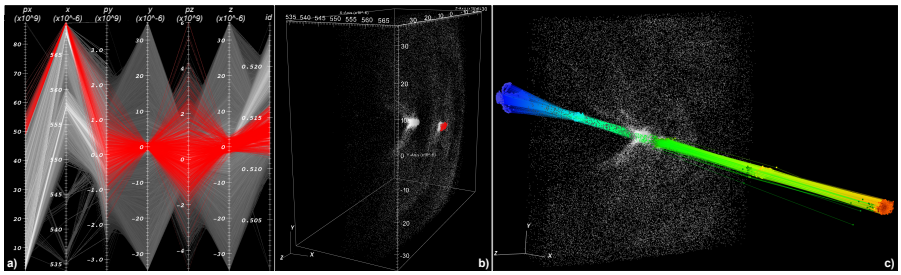
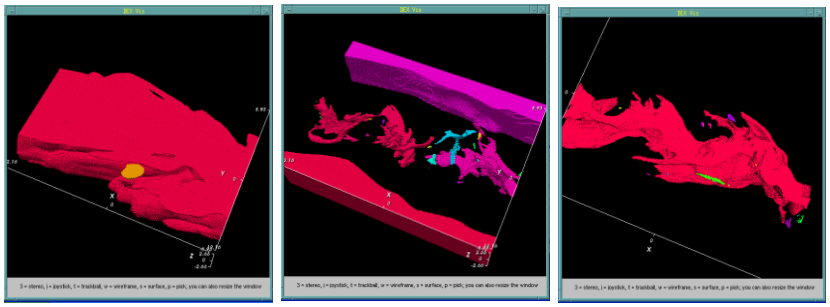
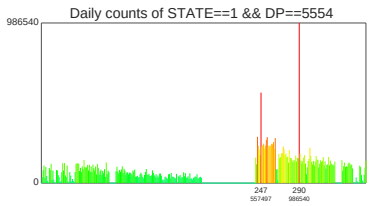


FIGURE 7.12: a) Parallel coordinates of timestep $t = 12$ of the 3D dataset. Context view (gray) shows particles selected with $px > 2 * 10^9$. The focus view (red) consists of particles selected with $px > 4.856 * 10^{10}$ && $x > 5.649 * 10^{-4}$, which indicates particles forming a compact beam in the first wake period following the laser pulse. b) Pseudocolor plot of the context and focus particles. c) Traces of the beam. We selected particles at timestep $t = 12$, then traced the particles back in time to timestep $t = 9$ when most of the selected particles entered the simulation window. We also trace the particles forward in time to timestep $t = 14$. In this image, we use color to indicate px. In addition to the traces and the position of the particles, we also show the context particles at timestep $t = 12$ in gray to illustrate where the original selection was performed. We can see that the selected particles are constantly accelerated over time (increase in px) since their colors range from blue (relatively low levels of px) to red (relatively high levels of px) as they move along x over time.

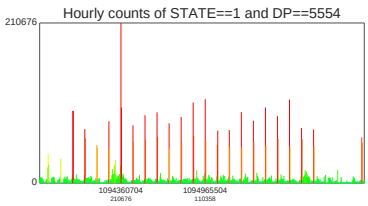


(a) $CH_4 > 0.3$ (b) $temp < 3$ (c) $CH_4 > 0.3$ AND $temp < 4$

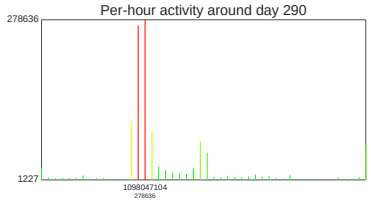
FIGURE 7.13: A visualization of flames in a high-fidelity simulation of methane-air jet. The images show the cells in a 3D block-structured dataset that were returned by three different queries.



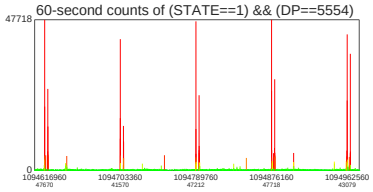
(a) Histogram of suspicious activity levels over a one-year period at one-day temporal resolution.



(b) Suspicious activity levels over a four-week period at one-hour temporal resolution.

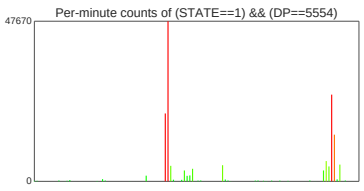


(c) Suspicious activity levels over a one-day period at one-hour temporal resolution.

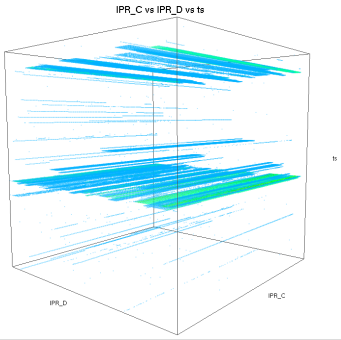


(d) Suspicious activity levels over a five-day period at one-minute temporal resolution. The attack is occurring every day at 21:15 local time.

FIGURE 7.14: Forensic network traffic analysis is conducted by examining histograms of suspicious traffic activity at varying temporal resolution. These examples go from coarse, per-day resolution over a one-year time window down to per-minute resolution over a five-day window, and show a regular pattern of systematic network attacks that occur with temporal regularity.



(a) This histogram shows suspicious activity over a two-hour period at one-minute temporal resolution. The spikes in this histogram correspond to the "sheets" in the adjacent image.



(b) A 3D histogram: the vertical axis is time, the other two axes are the C and D octets of the destination host address. The "sheets" indicate that the remote host(s) are performing a scan of all IP addresses within a given IP address space, indicating the attack is a scan.

FIGURE 7.15: Two- and three-dimensional histograms are the building blocks for visual data exploration of network traffic analysis. Here we see evidence of an organized scan: one or more remote hosts are probing sequential IP addresses within a block of addresses hoping to find a vulnerability.