Eurographics Symposium on Parallel Graphics and Visualization (2006)
Alan Heirich, Bruno Raffin, and Luis Paulo dos Santos (Editors)

# Remote Large Data Visualization in the ParaView Framework

Andy Cedilnik,[1] Berk Geveci,[1] Kenneth Moreland,[2] James Ahrens,[3] and Jean Favre[4]

[1]Kitware Inc., Clifton Park, New York
[2]Sandia National Laboratories, Albuquerque, New Mexico
[3]Los Alamos National Laboratory, Los Alamos, New Mexico
[4]Swiss National Supercomputing Centre, Switzerland

**Abstract**

*Scientists are using remote parallel computing resources to run scientific simulations to model a range of scientific problems. Visualization tools are used to understand the massive datasets that result from these simulations. A number of problems need to be overcome in order to create a visualization tool that effectively visualizes these datasets under this scenario. Problems include how to effectively process and display massive datasets and how to effectively communicate data and control information between the geographically distributed computing and visualization resources. We believe a solution that incorporates a data parallel data server, a data parallel rendering server and client controller is key. Using this data server, render server, client model as a basis, this paper describes in detail a set of integrated solutions to remote/distributed visualization problems including presenting an efficient M to N parallel algorithm for transferring geometry data, an effective server interface abstraction and parallel rendering techniques for a range of rendering modalities including tiled display walls and CAVEs.*

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Parallel Processing

## 1. Introduction

Simulation packages generate massive amounts of data that are usually impossible or unreasonable to move. The size of these datasets are in gigabyte to terabyte ranges. Exacerbating the problem, the data are commonly at a remote location, often in a separate room or even at a geographically remote site. Remote visualization of these large datasets on tiled displays or CAVEs is a challenging problem.

The use of even consumer grade graphics hardware can significantly improve rendering speed over software techniques. However, even the most sophisticated graphics hardware is limited by the amount of memory it can hold and the bandwidth of the processor. Furthermore, in most cases the scientific data are too large to be processed and displayed by a single computer. That is why, to perform fast interactive rendering of large datasets, parallel rendering is a common practice. The parallelization is also necessary when large multi-displays are used to display the data.

The need for client/server separation is necessary since it is difficult to move the data. The additional separation of server into multiple components, such as a data server and render server can also be beneficial. This design will make a logical separation between the nodes that perform the data processing/visualization and the nodes that perform the rendering. Furthermore, not all nodes require graphics hardware. Since data processing algorithms typically produce visualization results that are significantly smaller than the original data size, the number of rendering nodes with graphics hardware is usually smaller than the number of data processing nodes. This difference in the number of nodes presents an interesting problem: there needs to be a mapping from the large number of data server nodes to the small number of render server nodes.

Since the data are located remotly, some form of network data transfer has to be implemented. In general there are two major options. The first option is to send the geometry together with the rendering parameters to the rendering engine. The second option is to send the final images to be displayed on the client. Both options have their advantages and disadvanatages. The biggest drawback of the sending the geometry is that the geometry may be too large for it to be reason-

able to be sent over the network. In addition, the sending of geometry requires the client to have rendering capabilities. However, sending images can also be problematic because of the latency of interaction and the size of images.

To overcome these problems, the ParaView Framework proposes a set of methodologies, which when used in combination, will allow interactive viewing of data on remote locations [AJL*00, ABM*01, LHA01, Hen05]. This interactive viewing is possible without special hardware, and without sacrificing the fidelity of the experience. These methodologies have been tested in practice through extensive use by developers and scientists on several supercomputers all over the world. Furthermore it provides a set of client tools to act as an end-user application.

## 2. Related Work

A large body of research in visualizing large datasets using parallelism focuses specifically on parallel visualization algorithms. Much of the previous algorithmic work in the field of parallel visualization has focused on the area of parallel rendering. Parallel rendering approaches include photo-realistic rendering (i.e. ray tracing, radiosity and particle tracing) [RCJ98], polygon rendering [Cro97] and volume rendering [MPHK94, Yag96, Wit98]. Additional work has focused on parallel iso-surfacing [HH92, PSL*98] and geometry optimization [HH93]. These efforts are complementary to our efforts since our goal is the creation of a fully functional visualization framework. Previous algorithmic work can be integrated into ParaView as modules, further augmenting the framework's functionality for users.

There are a number of related efforts that provide solutions to the large data remote visualization and analysis problems. These include the VisIt and EnSight Gold large data visualization packages. Both have a similar data-server/client architecture to support remote visualization. They both currently use the Chromium package for parallel rendering and rendering to different display modalities [HHN*02]. Chromium provides an abstraction of graphics API commands for applications such as VisIt and Ensight Gold, and supports both sort-first and sort-last parallel rendering methods. ParaView incorporates parallel rendering methods [MWP01] directly as part of the framework making the communication between the visualization and rendering components more efficient and effective. Other related parallel data-flow based visualization tools include Data Explorer (DX) [ea92, AT95], AVS Express [ea89, KH93, DMO95] and SCIRun [PDJ97, MHJ98, JP99]. These tools provide a standard client/server architecture but do not address solving the large scale data parallelism and the remote visualization problems together as we do in this paper. Each of these tools uses a centralized executive for control and execution. However, controlling a large number of processes from a single centralized executive is difficult. In contrast to these tools, ParaView avoids the use of a centralized executive and therefore provides a more scalable solution.

## 3. Approach

The ParaView Framework solves the large data visualization problem using several approaches. These approaches include parallel processing, client/server separation, and render server/data server separation. These approaches give the ParaView Framework the flexibility to run in several modes. The simplest mode is the stand-alone application, in which all the processing is performed on the local system, or on a local cluster with the master node being the first node.

When data reside on a different system, or the local system is not capable of performing full rendering, remote visualization is necessary. To achieve this, the ParaView Framework supports a client/server mode. In this mode, the rendering and data processing are performed on the server and only geometry or imagery is sent to the client.

Since the ParaView Framework takes full advantage of the graphics hardware, it is beneficial for the rendering nodes to have special hardware. In the general case, however, the requirements for data processing resources is much higher than for rendering resources. This is why it is often more cost effective to have a smaller secondary rendering clusters than it is to add rendering hardware to the larger supercomputer. For this scenario, the ParaView Framework supports a separation of the data server and render server. In this mode, the user can take full advantage of the supercomputer and a smaller computer system with graphics hardware. This requires taking geometry generated by $M$ data server nodes and redistributing it so that it can be rendered on $N$ render server nodes, where $M > N$. In order to address this problem, Paraview uses an $M$ to $N$ redistribution algorithm.

### 3.1. Rendering Modes

As a parallel, general-purpose visualization tool, it is vital that ParaView supports multiple rendering modes. Different launch modes require the use of specialized render modes as do different display environments. The following list samples several of the rendering modes that ParaView must support.

**Local** A single process renders a locally available data set. Although the most mundane rendering mode, this is also probably the most frequently used.

**Image Delivery** A remote render server with a large amount of graphics processing power delivers images to a remotely connected desktop. Using image delivery allows users to employ a shared, graphics resource without leaving the comfort of their office. The challenge of implementing image delivery is to render and ship images fast enough to maintain interactive rendering rates.

**Tiled Display** A render server locally displays images upon

one or more images arranged in a tiled display. Such displays are frequently built in conference rooms or other theater-type environments. Because theater displays seldom have user interfaces, the GUI is provided as a remote client.

**Cave** An immersive display consisting of a room with a virtual environment projected upon two or more walls. The perspective projection of the image for each wall is designed to provide a consistent landscape for the user.

In addition to supporting all of these rendering modes, the ParaView Framework must provide a means of adapting to new rendering modes. There is no way to predict every possible rendering use case, so the ParaView Framework must be adaptable to new rendering requirements.

## 4. Technical Implementation

As described in the Section 3, the ParaView Framework consists of a data server, render server, and client. The following sub-sections describe the functionality of individual components. The majority of the code is, or is based on, the Visualization Toolkit (VTK). VTK provides ParaView with all the capabilities for reading, writing, processing, and visualizing data. Beyond that, it has capabilities for performing parallel processing of the data. VTK also provides a robust software engineering platform with its strict object oriented design and reference counting, as well as garbage collection schemes. This section describes the details of our remote visualization solutions: client/server streams for platform independent data transfer and remote invocation, a Server Manager for additional server abstraction, a M to N parallel geometry communication algorithm, rendering modules for rendering abstraction, and various clients for user interaction.

### 4.1. Client/Server Streaming

There are inherent problems with interprocess communication. These problems are compounded when the processes are running on heterogenous systems. The first problem is that various systems interpret data from memory in different ways. The second problem with interprocess communication is the lack of remote invocation of the procedures.

To overcome these problems, ParaView uses a Client/Server Streaming (CSS) module. This module abstracts the data type, byte ordering, and type size. It also provides a C++ API to package data into byte streams that can be then transported over the network to a remote system. Finally, it provides a platform independent way to perform remote invocations of specific ParaView server functionalities.

To marshal data, CSS uses the C++ IO streaming interface. Every CSS marshalling call consists of the command, arguments, and the *End* tag. The command is used by the CSS Interpreter to evaluate its content. The streams can also be used to marshal the data without being parsed by the interpreter. In those cases the command is ignored. The *End* tag is there to identify the end of the stream and to act as a separator between multiple commands/data blocks.

When unmarshaling the CSS, several methods are used to examine the stream. First of all, there is a method to check how many messages are on the stream. After the number of messages is known, messages can be examined for their content.

To abstract the remote invocation of methods, the ParaView Framework uses a CSS Interpreter. This interpreter uses the CSS to encode the remote method call and invoke it on the remote side. The advantage of this approach is that there is no difference between invoking methods on the local host or on a remote host. The server code (all the filters and other auxiliary classes) are automatically wrapped into the CSS wrapping. This wrapping is a thin layer of C++ code that, given a CSS, knows how to execute the set of methods described in the stream.

### 4.2. Server Manager

Although the CSS module makes it possible to remotely invoke methods and exchange data in a platform independent way, it has limitations that the Server Manager addresses. These limitations are:

**Complexity of use:** To invoke a method, the programmer has to create a stream, assign the method name and arguments, terminate the stream, select the right server and node, and finally send the stream. Compared to a simple C++ method invocation, this is tedious.

**Lack of state:** Once a method is invoked and the stream freed, the arguments passed to the server are lost. To query the state, the client program has to send another stream to the server and parse the returned stream. This is not simply a convenience issue, as communicating with the server to query the state may cause too much network traffic and slow down the client.

**Lack of Data gathering:** A stream can be invoked on either the root node or all the nodes of a server. The CSS module can receive the result returned from the first node but cannot gather the results across all server nodes. For example, to compute the global bounds of a dataset requires a reduction operation that uses the values returned from all nodes. The CSS modules cannot do this.

#### 4.2.1. Higher Level Interface

The main objective of the Server Manager module is to provide a simple and narrow interface accessible by clients and scripting interfaces. Another important objective is to hide the low-level CSS interface from client code so that it is not impacted by interface changes (such as VTK API changes). These are accomplished by using the proxy design pattern

[GHJV95]. Each server-side, distributed C++ object is represented by a C++ proxy object on the client side, which is responsible of managing the life-cycle of server side objects. Proxies are managed by the proxy manager, which is responsible for loading XML configuration files as well as creating, managing, and destroying proxies. The configuration files contain proxy definitions organized in groups.

Once the configuration file is loaded, the client can ask the Server Manager for a new proxy by passing the name of the group and proxy, for example "sources" and "DEMReader". The proxy manager then creates a proxy object, configures it based on the contents of the XML definition, and returns a pointer.

#### 4.2.2. State Management

In order to have full access to the server state, the client needs to know the values of server objects' properties. These properties are usually set by a *set* method, obtained by a *get* method, and stored in property objects contained by proxies. When a proxy is created, the proxy manager creates appropriately initialized property objects.

#### 4.2.3. Data Gathering

Another important responsibility of the Server Manager is to collect meta-data from the server and present it to the client as serial information. For example, the Server Manager can compute the global bounds of a dataset or the global range of a data array from a distributed dataset. It does this by creating parallel algorithm objects that send reduced information to the master process. This information is then communicated to the client and presented through the Server Manager interface using Information objects and information property objects. Information properties have the same interface as other properties except they do not allow setting of values.

### 4.3. M to N Parallel Geometry Communication

Often it is not practical or even possible to move the simulation data to the render server and process it there. One solution to this problem is to process the data on the server/cluster where it was generated and larger resources are available. Once processed, the resulting geometry is transfered to a render server/cluster.

The algorithm presented here redistributes geometry from $M$ to $N$ processes and can be shown to run in two $\log_2 P$ stages, where $P$ is the number of processors [BA].This algorithm has two parts: the first redistributes geometric elements to a balanced arrangement and the second redistributes the geometric elements from the balanced arrangement to the goal distribution. In more detail:

1. Make a transfer schedule to redistribute the initial element distribution, A, to a balanced distribution.
2. Make a transfer schedule to redistribute from the goal distribution, B, to the balanced distribution.
3. Reverse schedule 2 by running exchanges in the opposite order and replacing sends with receives.
4. Apply schedule 1.
5. Apply the reversed schedule.

The $4^{\text{th}}$ and $5^{\text{th}}$ steps here each have $\log_2 P$ stages. These 2 steps are much the same so only the balancing part of redistribution is described. For simplicity in describing and analyzing the redistribution algorithm, let $P$, the number of processors, be a power of 2. Also assume that the total number of geometric elements is a multiple of $P$. The elements are balanced in a series of $\log_2 P$ stages. During stage $i$, where $i$ iterates from 0 to $(\log_2 P - 1)$, the processors are split equally into sequential groups of length $2i$. Groups are paired into a left and a right group. Each processor in the left group is paired with a processor in the right group that is in the same position within the group. In parallel, the processors with more elements send half the difference in number of elements to the processors with fewer elements. This results in an equal number of elements on each processor in the pair. At the beginning of the stage all processors within a group have the same number of elements and the processors are paired and balanced with processors in the second group, each with the same number of elements. The result is that all processors in both groups have an identical number of elements at the end of the stage. For the subsequent stage, both the right and left groups are merged into one group where all processors have an identical number of geometric elements. Thus, at the completion of the final stage, all processors contain the same number of geometric elements and the geometric load is balanced.

### 4.4. Render Modules

A design criterion unique to a parallel visualization framework like ParaView is the necessity of efficient parallel rendering. ParaView's rendering must adapt to a variety of rendering platforms run with any of the client/server modes specified in Section 3. Furthermore, for each client/server mode ParaView must implement each of the rendering modes specified in Section 3.1, where applicable. Future developments in visualization hardware or software may necessitate further rendering algorithms.

Properly rendering data in ParaView requires the coordination of many VTK objects. To shield the majority of code from the complexities that arise from rendering, ParaView employs the facade design pattern [GHJV95]. It encapsulates the task of rendering into a Render Module object. The Render Module sets up all the necessary VTK objects for rendering and provides a unified interface for all rendering tasks.

Using the facade pattern provides another important feature beyond merely simplifying code. Because the interface is abstracted from the implementation, ParaView can easily use a different implementation by substituting another Render Module object with the same interface. Figure 1 shows

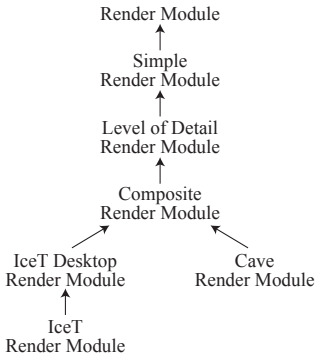


**Figure 1:** *Render Modules available in ParaView.*

the hierarchy of some of the Render Module objects available, and a description of each follows.

The Simple Render Module is a class that handles the most basic rendering functions. It maintains a collection of objects that make it possible to perform rendering: rendering context, user interaction, camera control, and lighting. The Simple Render Module also holds a collection of Display objects. A Display object manages the rendering of a single viewable object in a scene. Displays are described in detail in Section 4.4.1.

The Level of Detail Render Module adds the ability to sacrifice rendering quality for speed. The interactivity and responsiveness of ParaView directly impacts its usability. At times of user interaction (for example, rotates, pans, and zooms of the view), this Render Module can render a low level of detail of the data. When the user is not interacting with the view, ParaView renders the full details of the data. Lower levels of detail are formed by performing quadric clustering [Lin00] on the polygon meshes. Users frequently complain if interactivity is not maintained whereas changing levels of detail are seldom problematic. For this reason, all render modules inherit from the Level of Detail Render Module.

The Composite Render Module is an abstract class that handles the added complexities associated with parallel rendering with clients and servers. These duties include moving data between partitions and adding level of detail controls for parallel rendering overhead. The Composite Render Module, being an abstract class, does not define the parallel rendering algorithm used. That is implemented by the subclasses.

When running ParaView in data-server/render-server/client mode, data must first be moved from the data server to the render server. Since the number of processes in the data server can be different than the number of processes in the render server, the composite render module requires the M to N algorithm discussed in Section 4.3. When the data size is small, the Composite Render Module also has the ability to collect the data and send it all to the client or replicate it on all processes of the render server. This allows ParaView to remove the overhead of parallel rendering for small data sets.

As the name implies, the Composite Render Module is designed to do sort-last parallel rendering [MCEF94]. This approach to parallel rendering is used because, unlike sort-first or sort-middle, it scales very well with the number of polygons and number of processes [WPLM01]. The sort-last approach combined with levels of detail and data replication performs equally well or better than a sort-first approach in nearly all cases, including tile displays.

The overhead of sort-last parallel rendering as well as image delivery are proportional to the size of the image being generated. Thus, the Composite Render Module provides an additional level of detail: image reduction. During interaction, smaller images are rendered and then inflated to fill the display. Although this process results in images with reduced fidelity, it can increase the rendering rates dramatically.

The IceT Render Modules (IceT Desktop Render Module and IceT Render Module) use the IceT library to implement parallel rendering. Details of how the IceT library interfaces with VTK can be found in [MT03].

In addition to parallel rendering, the IceT Desktop Render Module must also deliver images to the client. Image delivery can be a bottleneck and may benefit greatly from a compression algorithm. However, the compression algorithm must be chosen carefully. The time to encode and decode the image must be recovered by savings in transfer time. Thus, a heavy compression algorithm such as JPEG has the potential of actually increasing the image delivery time. To deliver images, the IceT Desktop Render Module uses the Sequential Unified Image Run Transfer (SQUIRT) algorithm. As a run length encoding algorithm, SQUIRT can encode and decode images in a very small amount of time. SQUIRT further improves its speed by performing operations on RGB triplets with single 32-bit integer operations. Run lengths are packed in the fourth byte, normally reserved for alpha, which helps reduce both the time and memory required for encoding.

For most images, run length encoding alone provides very poor compression. Simple polygon shading usually results in run lengths of size one. The run lengths can be dramatically improved, however, if the color fidelity is reduced. Thus, SQUIRT masks out several bits in the colors, those that will have a minimal impact on color perception, before comparing them. Although the bits are masked out for run length computation, they are still included in the encoded image for two reasons. First, it is usually faster to leave the extra bits in the data so that operations can continue to perform operations on 32-bit integers. Second, even though the extra bits provide no extra information in principle, they tend to reduce banding artifacts in practice.

The IceT Render Module behaves much like its super-

class, except that images are displayed locally on the render server to theater environments. The IceT library also supports image compositing to tiled displays, and these algorithms are enabled in the IceT Render Module. The tile-display compositing algorithms are described in [MWP01].

The Cave Render Module is used to drive CAVE environments, which have multiple displays that are rendered with different perspectives. Although the Cave Render Module currently does not actually do any image compositing, it shares enough functionality to inherit from the Composite Render Module. The Cave Render Module always replicates its data amongst all processes in the render server.

#### 4.4.1. Displays

A *display* object holds information about something drawn as visible geometry. Because of the large variety of displayable objects drawn by ParaView, it is convienent to encapsulate their state and description in a hierarchy of *display* objects. Render Modules hold collections of *display* objects that define what is drawn inside them.

Simple geometries such as *Axes* or *Scalar Bars* inherit right from the base class. *3D Widgets* have their own hierarchy. A *3D Widget* is comprised of a simple geometry, such as a point, line, plane, or sphere and some user interaction. It is the *display's* responsibility to allow interaction to occur on the client and send the parameters to the render server (if they are in different processes).

A *consumer display* is attached to the end of a VTK pipeline and draws a representation of its data. If the data server, render server, or client reside on different processes, the *consumer display* must move data accordingly. A *data object display* renders the geometric representation of a VTK data set. These *displays* have the potential of rendering large amounts of data and must therefore be careful about where data are moved and how it is rendered. To make sure this is done properly, the *data object display* has several specialized subclasses for use with the different Render Modules (discussed in Section 4.4).

### 4.5. Clients

The ParaView Framework supports arbitrary clients. The most prevalent client is the desktop client build on top of Tcl/Tk that is distributed with the ParaView Framework. Furthermore, there is a WebVis client, which exposes the ParaView Framework functionality through the Web. It was developed by Kitware and Army Research Laboratory to facilitate enterprise level visualization. Finally there are Python and Tcl scripting clients available that allow users to write simple Python or Tcl scripts to drive ParaView server. Using these scripts, ParaView Framework can be driven in a batch mode in scenarios when the user interface is not feasible. In addition to these clients, there are several applications available that use the ParaView Framework internally.

## 5. Results and Discussion

ParaView has been in development since 2001, and since that time it has seen its functionality grow broadly. The solutions presented in this text come directly from the challenges that Sandia National Laboratories and Los Alamos National Laboratory have met, creating a highly scalable and interactive framework for visualization. As a measure of ParaView's success, over 30 laboratories and universities around the world use it. Furthermore, it is downloaded over 45 times a month by other users.

Apart from simple usage numbers, the success of a parallel framework such as ParaView's can be measured in its ability to scale to large problems. ParaView is continuously challenged with ever growing data problems and processing supercomputers. To date, the biggest visualization problems are being performed at Sandia National Laboratories for the Red Storm project.

Red Storm is a new supercomputer built at Sandia National Laboratories designed to run physics simulations with very high fidelity [JS05]. Red Storm routinely creates some of the largest sets of scientific data in the world. To handle the visualization requirements of such large physical simulations, Sandia has also built some of the biggest visualization clusters, such as the Red RoSE cluster shown in Figure 3. As Red Storm came on line, Sandia was obligated to show the US Department of Energy visualization capability for the data Red Storm generates. Using ParaView, this milestone was successfully completed [Whi05]. Figure 2 shows large Red Storm data visualized with ParaView.

Another success story is the example of ParaView usage for CFD simulation of a Pelton turbine, with the modeling of turbulent three-dimensional confined flows and free surface flows. The complexity of the simulation arises from the large range of scales in this flow which range typically from distributor inlet to droplet diameters. The flow requires an unsteady rotor-stator modeling to properly predict flow features.

The solver used is the parallel flow solver ANSYS. Initially it was used with a single processor application (CFX-5 Post), which was clearly over-loaded with mesh sizes beyond 20-30 millions cells. ParaView, was able to perform post-processing in a fully parallel fashion. The parallel solution starts from reading the data on multiple data server nodes, respecting the natural sub-division of the model, as shown in Figure 4. Load-balancing and volume assignment is decided at run-time, based on the number of processors available. A full transient animation handles tera-bytes of data and allows a data analysis that was previously unobtainable.

ParaView works well on data sets ranging from kilobytes to terabytes because it is designed from the ground up to be a scalable parallel application. ParaView's client/server design makes it possible to minimize the movement of large data

sets while maximizing its accessibility. ParaView's architecture makes it convenient to use by a variety of users with various display environments. Finally, the ParaView Framework can be utilized by multiple clients, each targeting a different visualization need.

## 6. Acknowledgments

This work was done in part at Sandia National Laboratories. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

## References

[ABM*01] AHRENS J., BRISLAWN K., MARTIN K., GEVECI B., LAW C. C., PAPKA M.: Large scale data visualization using parallel data streaming. *IEEE Computer Graphics and Applications 21*, 4 (2001), 34–41.

[AJL*00] AHRENS J., JAMES, LAW C., SCHROEDER W., MARTIN K., PAPKA M.: *A Parallel Approach for Efficiently Visualizing Extremely Large Time-Varying Datasets, Technical Report #LAUR-00-1620,*. Tech. rep., 2000.

[AT95] ABRAM G., TREINISH L.: An extended dataflow architecture for data analysis and visualization. IEEE Computer Society Press, pp. 263–270.

[BA] BRISLAWN K., AHRENS J.: *MxN Load Redistribution for the Efficient Use of Parallel Computation and Rendering Resources.* Tech. rep., Los Alamos National Laboratory, LA-UR-03-5481.

[Cro97] CROCKETT T.: An introduction to parallel rendering. In *Parallel Computing* (1997), p. 23(7):819£843.

[DMO95] DUNCAN B. S., MACKE T. J., OLSON A. J.: Biomolecular visualization using avs. *Journal of Molecular Graphics 13*, 5 (1995), 271–282.

[ea89] ET. AL. C. U.: The application visualization system: A computational environment for scientific visualization. In *IEEE Computer Graphics and Applications* (1989).

[ea92] ET. AL. B. L.: An architecture for a scientific visualization system. In *Proceedings of Visualization 1992* (1992), IEEE Computer Society Press, pp. 107–114.

[GHJV95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J.: *Design Patterns*. Addison Wesley, 1995. ISBN 0-201-63361-2.

[Hen05] HENDERSON A.: *ParaView Guide, A Parallel Visualization Application*. Kitware Inc., 2005.

[HH92] HANSEN C., HINKER P.: Massively parallel isosurface extraction. In *Proceedings of Visualization 1992* (1992), IEEE Computer Society Press, pp. 77–83.

[HH93] HINKER P., HANSEN C.: Geometric optimization. In *Proceedings of Visualization 1993* (1993), IEEE Computer Society Press.

[HHN*02] HUMPHREYS G., HOUSTON M., NG R., FRANK R., AHERN S., KIRCHNER P., KLOSOWSKI J.: Chromium: A stream processing framework for interactive rendering on clusters. In *Proceedings of the 29th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH-02)* (New York, July 21–25 2002), Spencer S., (Ed.), vol. 21, 3 of *ACM Transactions on Graphics*, ACM Press, pp. 693–702.

[JP99] JOHNSON C., PARKER S.: The scirun parallel scientific computing problem solving environment. In *Ninth SIAM Conference on Parallel Processing for Scientific Computing* (1999).

[JS05] JEFFERSON K. L., STURTEVANT J. E.: *Red Storm Usage Model: Version 1.12*. Tech. Rep. SAND2005-6926, Sandia National Laboratories, Albuquerque, NM, 2005.

[KH93] KROGH M., HANSEN C.: Visualization on massively parallel computers using cm/avs. In *In AVS Users Conference* (1993).

[LHA01] LAW C., HENDERSON A., AHRENS J.: An application architecture for large data visualization: A case study. In *Procedings of Visualization 2000, PVG Symposium* (2001), ACM Press.

[Lin00] LINDSTROM P.: Out-of-core simplification of large polygonal models. In *SIGGRAPH 2000 Conference Proceedings* (July 2000), pp. 259–262.

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4 (July 1994), 23–32.

[MHJ98] MILLER M., HANSEN C., JOHNSON C.: Simulation steering with scirun in a distributed environment. In *Lecture Notes in Computer Science* (1998), Springer-Verlag.

[MPHK94] MA K., PAINTER J., HANSEN C., KROGH M.: Parallel volume rendering using binary-swap compositing. In *IEEE Computer Graphics* (1994), pp. 59–67.

[MT03] MORELAND K., THOMPSON D.: From cluster to wall with VTK. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (October 2003), pp. 25–31.

[MWP01] MORELAND K., WYLIE B., PAVLAKOS C.: Sort-last parallel rendering for viewing extremely large data sets on tile displays. In *Proceedings of the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics* (October 2001), pp. 85–92.

[PDJ97] PARKER S. G., D.M.WEINSTEIN, JOHNSON C. R.: The scirun computational steering software system. In *Modern Software Tools in Scientific Computing*

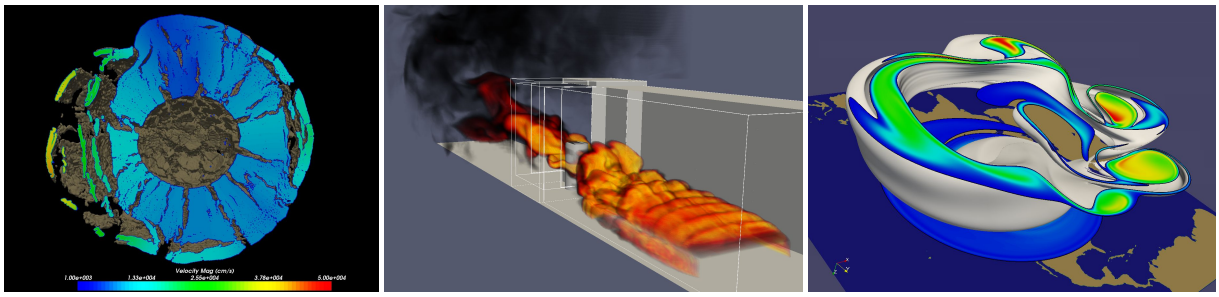

**Figure 2:** *Examples of physical simulations run on Red Storm and visualized with ParaView. The left image is of an asteroid detonation simulation comprising over 1 billion structured cells. The middle image is a fire test facility simulation with over 8 million unstructured cells. The right image is a 1 billion structured cell climate simulation of the polar vortex.*

(1997), Arge E., Bruaset A. M.,, Langtangen H. P., (Eds.), Birkhauser Press, pp. 1–40.

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.: Interactive ray tracing for isosurface rendering. In *Proceedings of Visualization 1998* (1998), IEEE Computer Society Press.

[RCJ98] REINHARD E., CHALMERS A., JANSEN F.: Overview of parallel photo-realistic graphics. In *Proceedings of Eurographics 98* (1998).

[Whi05] WHITE D.: *Red Storm Capability Visualization Level II ASC Milestone #1313 Final Report*. Tech. Rep. SAND2005-5989P, Sandia National Laboratories, Albuquerque, NM, 2005.

[Wit98] WITTENBRINK C.: Survey of parallel volume rendering algorithms. In *Proceedings of Parallel and Distributed Processing Techniques and Applications* (1998), pp. 1329–1336.

[WPLM01] WYLIE B., PAVLAKOS C., LEWIS V., MORELAND K.: Scalable rendering on PC clusters. *IEEE Computer Graphics and Applications 21*, 4 (July/August 2001), 62–70.

[Yag96] YAGEL R.: Towards real time volume rendering. In *Proceedings of GRAPHICON'96, volume 1* (1996), pp. 230–241.

**Figure 3:** *Visualization nodes of Sandia National Laboratories' Red RoSE cluster. The cluster has 264 visualization nodes.*

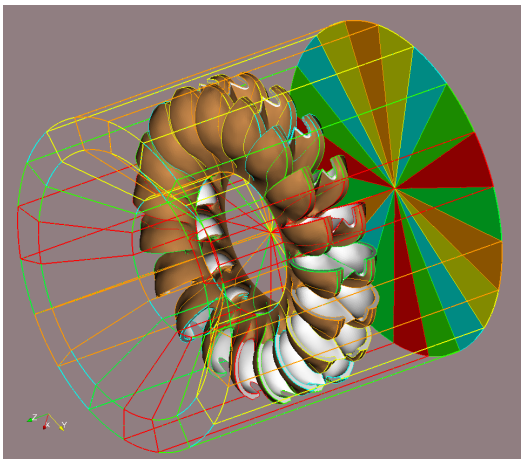

**Figure 4:** *3D mesh sub-regions assigned cyclically (see color coding) to different data server nodes for load-balancing.*