# From Cluster to Wall with VTK

Kenneth Moreland[*] David Thompson[†]
Sandia National Laboratories

## Abstract

This paper describes a new set of parallel rendering components for VTK, the Visualization Toolkit. The parallel rendering units allow for the rendering of vast quantities of geometry with a focus on cluster computers. Furthermore, the geometry may be displayed on tiled displays at full or reduced resolution. We demonstrate an interactive VTK application processing an isosurface consisting of nearly half a billion triangles and displaying on a power wall with a total resolution of 63 million pixels. We also demonstrate an interactive VTK application displaying the same geometry on a desktop connected to the cluster via a TCP/IP socket over 100BASE-T Ethernet.

**CR Categories:** I.3.8 [Computing Methodologies]: Computer Graphics—Applications

**Keywords:** parallel rendering, desktop delivery, tile display, PC cluster, Chromium, VTK

## 1 Introduction

"That's great. How can I use it?" This is the question our visualization research team is faced with whenever we present a promising new tool to our analysts. Until recently, our best solution was to wrap the tool in its own user interface. This proliferation of visualization tools wastes human resources on many fronts. It requires our researchers to design, build, and maintain user interfaces for their tools. Since our visualization researchers are often neither motivated nor experienced at designing user interfaces, the tool interfaces are often ill-conceived, rarely consistent, and never tightly coupled. Analysts are forced to learn a wide variety of interfaces for the tools they wish to use, if they are aware of the tools' existence at all. Often this results in tools being unused.

To alleviate these problems, we have chosen to leverage the **Visualization Toolkit** (VTK) [Schroeder et al. 2002]. VTK is an open-source API containing a comprehensive suite of visualization tools. More importantly, VTK incorporates an extensible, component-based architecture. Our new approach to delivering visualization tools is to wrap these tools into VTK components and store these components in a common component library. This should allow each tool to work together with current and future tools under the same, consistent user interface. All three DOE Advanced Simulation and Computing (ASC) national labs, Sandia,

[*]e-mail: kmorel@sandia.gov
[†]e-mail: dcthomp@sandia.gov

IEEE Symposium on Parallel and Large-Data Visualization and Graphics 2003,
October 20-21, 2003, Seattle, Washington, USA,
pages 25-31
Copyright 2003 IEEE

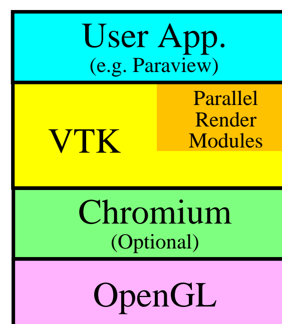


Figure 1: A high level view of our proposed visualization system.

Los Alamos, and Lawrence Livermore, are using VTK to some extent in their visualization R&D programs.

Our visualization needs put a large strain on VTK. ASC routinely creates simulations on the order of 50 million cells [Heermann 1999], and it is predicted that by 2004 applications will routinely create 250 million cell simulations requiring up to a petabyte of storage [Smith and van Rosendale 1998]. The processing of such data is well beyond the capabilities of a standard workstation. Our most cost-effective means for handling this data in a timely manner is the use of commodity cluster computers running distributed memory algorithms. The creation of such high fidelity models also necessitates the use of high fidelity displays for visualization. To realize these high resolution displays we build **power walls,** projected displays driven by tiled arrays of commodity projectors. We require our VTK applications to run interactively on clustered computers while either driving a power wall or shipping images back to a remote computer for desktop delivery.

Figure 1 shows the layout of the high-level functional components of our system. Rather than build our user level application directly on top of a graphical API such as OpenGL, we plan to build our applications on top of the VTK framework. The advantage is twofold: We can leverage the enormous visualization code base available with VTK, and we can build an application that can be flexible enough to accept emerging visualization technologies. We can also potentially layer VTK on top of Chromium to leverage Chromium's powerful parallel rendering capabilities. Because Chromium's interface mimics that of OpenGL, we can optionally add or remove that layer with little impact on the system. This paper describes our work adding modules to VTK that can interchangeably perform various cluster-driven parallel rendering algorithms.

## 2 Previous Work

Thanks to a recent collaboration between Kitware and the ASC/VIEWS program, VTK now supports parallel programming and can be run on cluster computers [Ahrens et al. 2000]. The parallel VTK libraries support several parallel modes. The mode we concern ourselves with in this paper is data parallelism in which the input data is split amongst processes, the pieces are filtered and rendered independently, and then composited to one image in the end. This method allows us to make effective use of the parallel and

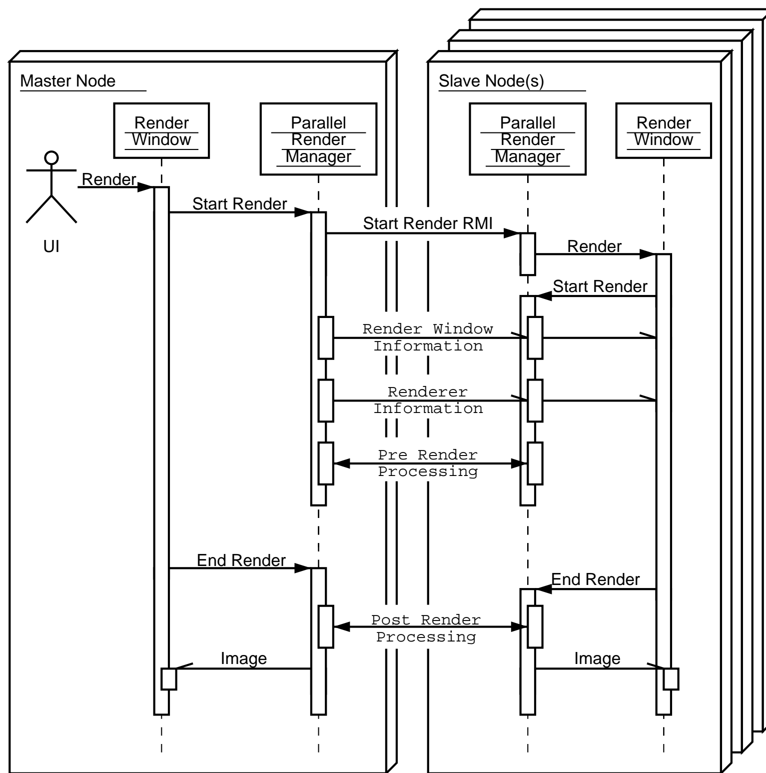


Figure 2: Interactions of the parallel render manager during a render.

distributed nature of our clusters, but can break down if the model pieces are too big to be processed by the available resources[1].

While parallel visualization and rendering are now supported by VTK, display to a power wall and desktop delivery currently are not. Instead, all current parallel rendering codes in VTK assume that the user interface is available at the "root" process in a parallel job. One possible approach for driving power walls is to use Chromium [Humphreys et al. 2002]. Chromium is capable of replacing the OpenGL library loaded by an application at runtime, intercepting the stream of OpenGL commands, and performing several alterations to it, including driving a power wall. Unfortunately, an OpenGL stream is inherently serial and must be issued from a single process, which is a huge bottleneck when dealing with large amounts of data. Chromium also supports a parallel rendering mode, but an application must be "aware" of Chromium to take advantage of this parallel rendering. We describe our efforts to make VTK applications Chromium aware.

Another power wall display solution available to us is the Image Composite Engine for Tiles (ICE-T). The ICE-T API is an implementation of the work presented by Moreland, Wylie, and Pavlakos for efficiently performing sort-last parallel rendering onto tile displays [Moreland et al. 2001]. The ICE-T methodology for rendering fits well with VTK's existing parallel rendering paradigm. We therefore implemented a new parallel rendering engine for VTK using the ICE-T API.

[1]Ahrens et al. [Ahrens et al. 2001] describe using VTK in a streaming mode to perform out-of-core processing of data.

# 3 Parallel Rendering Interface

Parallel rendering in VTK is supported via the `vtkCompositeManager` class. This class works by listening for render events on any renderable VTK window. After each render, but before the images are presented to the user, `vtkCompositeManager` reads back the frame buffers, performs a sort-last compositing, and writes the image back to the frame buffer.

Although `vtkCompositeManager` provides much of the functionality we need for our parallel rendering tools, it is unnecessarily constrained to a single mode of parallel rendering. Our initial approaches to creating parallel rendering modules involved subclassing `vtkCompositeManager`. Unfortunately, `vtkCompositeManager` expects its subclasses to only handle image data that has already been read from frame buffers. Attempting to work around this limitation resulted in obfuscated code.

Our response was to create a new class, `vtkParallelRenderManager`. It works in a manner similar to `vtkCompositeManager` in that they both listen to render events and perform appropriate actions around them. The difference is that `vtkParallelRenderManager` is an abstract class designed to give subclasses as much or as little control as they need.

Figure 2 outlines how the `vtkParallelRenderManager` behaves during a render event. After the "master" or "root" parallel render manager receives the start render event, it broadcasts a remote method invocation to all other parallel render managers to also start a render. It then broadcasts rendering information to all other parallel render managers. The parallel render manager also calls protected virtual methods to allow subclasses to perform their own data synchronization. Finally, the parallel render manager class calls another protected virtual function to allow a subclass to perform any other necessary processing before the actual render oc-

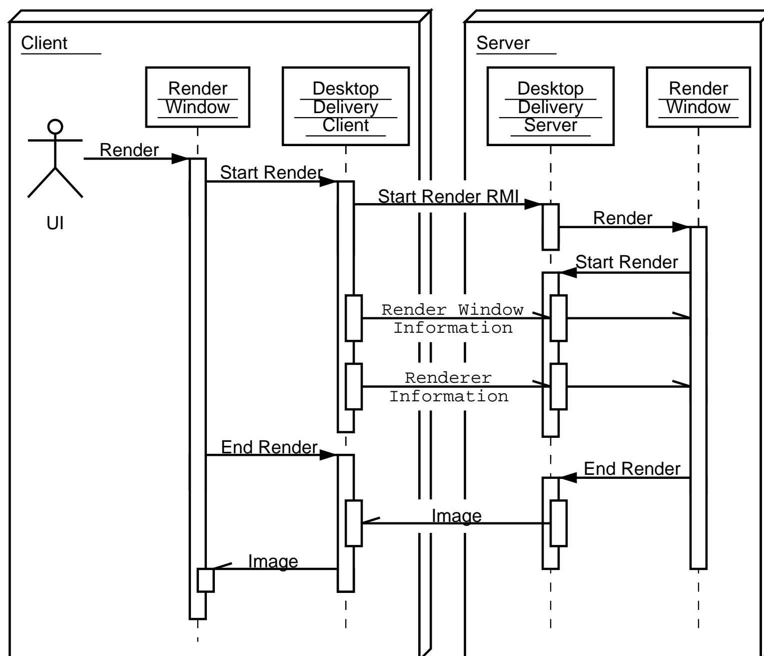


Figure 3: Interactions during rendering with desktop delivery.

curs.

The `vtkParallelRenderManager` then relinquishes control back to the render window, which proceeds to perform the actual image synthesis. Afterwords, each render window sends an end render event to its respective parallel render manager. The `vtkParallelRenderManager` calls yet another protected virtual function to allow subclasses to perform any post-processing, such as image composition.

In addition to handling render events and their propagation and synchronization as described above, `vtkParallelRenderManager` also has the following features.

**Object Creation** Provides factory methods for creating renderer and render window objects. Some parallel rendering schemes require fine control over the rendering process. As such, it may become necessary to override the behavior of renderers and render windows. By providing object creation, the parallel render manager can help ensure that the rendering objects match the parallel rendering scheme.

**Boundaries** Will compute the physical boundary, in 3-space, of the composite object rendered by all processes.

**Image Reduction** Can reduce the size of the image being rendered and then restore the image to full size for viewing. Reducing the image size can drastically reduce the time required for image composition strategies. The class can also automatically set the image reduction based on the user's desired rendering rate.

**Image Caching** If the post render processing requires the image to be read from the graphics card's frame buffer, the image will be cached for potential `vtkParallelRenderManager` users.

All of the parallel rendering classes described in this paper inherit from `vtkParallelRenderManager`. Doing this affords us a significant amount of code reuse. It also provides an abstraction that allows us to easily swap the parallel rendering method in our applications.

We have also implemented a class called `vtkCompositeRenderManager`. This class is identical in features to the `vtkCompositeManager` class distributed with VTK. However, having it also inherit from `vtkParallelRenderManager` allows it to play with our applications along with our other parallel rendering classes. The code for `vtkCompositeRenderManager` is quite small since `vtkParallelRenderManager` does most of the work. It took us very little time to code and debug `vtkCompositeRenderManager`. Our goal is to integrate these changes into VTK's composite manager.

## 4 Desktop Delivery

Since we in no way expect each analyst to have a parallel cluster available in his/her office, we find it important to provide the ability of remote desktop delivery. That is, we wish a user to be able to interactively control a visualization job rendering on a cluster and display back to a typical desktop machine connected via a local area network.

To this end, we have built a pair of collaborating objects: `vtkDesktopDeliveryClient` and `vtkDesktopDeliveryServer`. The client object is responsible for accepting user input and displaying rendered images. The server object is responsible for the visualization processing and rendering. The two objects are connected together with a pair of `vtkMultiProcessController` objects. `vtkMultiProcessController`, which is part of the VTK distribution, is an abstract interface to parallel process control and communication. The desktop delivery objects expect the process controller to control exactly two processes with client and server in opposite processes. The controller used is typically implemented with a socket, but does not have to be.

The `vtkDesktopDeliveryClient` object attaches itself as an observer to a render window. As shown in Figure 3, when a render event occurs, the desktop delivery client object sends a render re-

quest to the server along with the rendering parameters, waits for an image to come back, and pastes the image to the window. As one would expect, the `vtkDesktopDeliveryServer` object responds to render requests from the client by invoking the render window and shipping the resulting image back.

The reader may notice a striking similarity between the operational features of the desktop delivery client/server and the parallel render manager described in Section 3. In particular, the interactions shown in Figures 2 and 3 are nearly identical. Because of this, both `vtkDesktopDeliveryClient` and `vtkDesktopDeliveryServer` inherit from `vtkParallelRenderManager`. While this inheritance strains the *is-a* requirement dictated by good object-oriented program design, we feel the amount of code re-use we achieved was too important to neglect. Furthermore, by inheriting from `vtkParallelRenderManager` we were able to take advantage of its boundary calculation and image reduction capabilities, both of which are vital properties of our desktop delivery.

Because the desktop delivery server was designed to enable delivery from clusters, it can optionally work with another `vtkParallelRenderManager` object. The `vtkDesktopDeliveryServer` performs image delivery after the other `vtkParallelRenderManager` generates the image. The desktop delivery server object uses the other parallel render manager to compute the complete object bounds, get timing statistics, and otherwise control the parallel rendering process. The desktop delivery server will also take advantage of the other parallel render manager's image caching to avoid multiple reads and writes from the graphics card frame buffer.

## 5 Chromium

We also need to drive power walls in our parallel VTK applications. One means of doing this is with the **Chromium** system [Humphreys et al. 2002]. Chromium intercepts a stream of OpenGL commands and filters them. Chromium's filters are pluggable components called **stream processing units** (SPUs). One such unit distributed with Chromium is the `tilesort` SPU. The `tilesort` SPU determines the area of the screen that groups of primitives occupy and ships them to cluster nodes responsible for displaying that portion of the screen.

Because Chromium has the ability to put itself in place of an OpenGL shared object library, it can provide a tile display and/or a number of other filtering operations to almost any OpenGL application without modification or even recompilation. While flexibility was a major design consideration for Chromium, so was efficiency. As such, many SPUs like `tilesort` can be applied with little or no degradation of frame rates. However, as mentioned before, using Chromium in serial mode is not feasible for large data models, so we must make our application Chromium aware so that we can use it in parallel.

To make our VTK applications Chromium aware, we built the `vtkCrRenderManager` object. Because the parallel rendering is really being handled by Chromium, `vtkCrRenderManager` does little but allow its superclass, `vtkParallelRenderManager`, to propagate render events and parameters. We also implemented `vtkCrOpenGLRenderWindow` and `vtkCrOpenGLRenderer` objects, which the Chromium render manager will build for the application. They work very much like their OpenGL counterparts except that they can interface directly with the Chromium API. Currently, they suppress the creation of unused windows, provide bounding box hints for `tilesort`, and provide some interaction with a few other select SPUs. In time, the features of these objects may grow to strengthen the coupling between VTK applications and Chromium.

Using Chromium and its `tilesort` SPU, one can achieve very impressive frame rates on large power wall displays. However, this approach has scaling issues. Generally, the number of rendering nodes is fixed to the number of tiles being displayed. Adding more rendering nodes requires splitting the tiles into smaller pieces. This approach means the smaller pieces must be recombined to form a full image for each tile. This must be done either with additional image combining hardware [Stoll et al. 2001], which we don't have, or with software by reading back the frame buffer, largely at the expense of frame rates. Furthermore, the sort-first approach to parallel rendering performed by `tilesort` has inherent scalability and load balancing issues. Some work has been done to correct these issues [Samanta et al. 1999], but has not yet been implemented in Chromium.

Of course, Chromium is not bound to the sort-first approach to parallel rendering that `tilesort` implements. It can just as easily support a sort-last approach, and the `binaryswap` SPU does just that. Like most sort-last parallel renderers, `binaryswap` scales well with the size of geometry being rendered and the number of processors doing the rendering, but at the expense of frame rate speeds. However, the sort-last technique is very sensitive to the resolution of the output display. Generating images that fit on a typical desktop is feasible, but generating images on high resolution tiled displays is not. We are currently considering techniques that may use `tilesort` and `binaryswap` together to better distribute the work for large amounts of data on tiled displays.

## 6 ICE-T

Another power wall display solution available to us is the **Image Composite Engine for Tiles** (ICE-T). The ICE-T API allows applications to easily perform sort-last parallel rendering. It uses several image space reduction and compression techniques to make the composition of even high resolution tiled displays feasible. The details of ICE-T's algorithms can be found in [Moreland et al. 2001].

Our initial goal was to embed ICE-T into a Chromium SPU. However, we quickly found that this was an impractical target. The Chromium SPUs, because they operate on streams of OpenGL commands, use a *push* model. That is, the application pushes OpenGL commands to the first SPU, which pushes them to the next SPU, and so on. A SPU generally processes one command and then moves on. In contrast, ICE-T uses a *pull* model. ICE-T can perform compositions for images that can be much larger than what the available graphics card can render in one pass. It therefore may have to pull several images of the same geometry rendered with different projections. For this to be performed in a Chromium SPU, the entire OpenGL stream would have to be cached each frame. This would be very inefficient for large amounts of geometry, which ICE-T was specifically designed to handle. We briefly considered Chromium hints and extensions that might make the application's data available. Ultimately, we decided that trying to shoehorn ICE-T into Chromium like this would require applications to be so tailored to the ICE-T SPU that they might as well use the ICE-T API directly.

We were able to instead embed ICE-T into the VTK framework. We found this much more practical because, like ICE-T, VTK uses a *pull* model. A render request is given to the window at the bottom of the VTK pipeline. The event is propagated up the pipeline as each component pulls fresh geometry or images from the component above it. ICE-T fits well in this framework.

To start, we built an object called `vtkIceTRenderManager`. As always, this class inherits from `vtkParallelRenderManager`. As such, it attaches itself to a render window and listens for render events. `vtkIceTRenderManager` works in conjunction with the ICE-T API to compose images.

However, ICE-T also must in some cases transform the projection matrix and perform several renderings. This behavior is

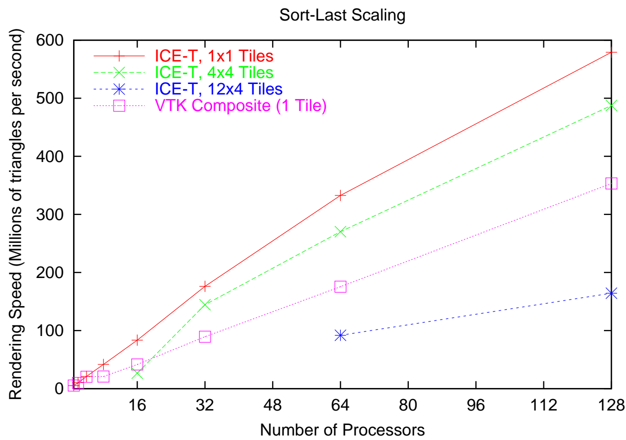


Figure 4: Scaling of sort last parallel rendering. ICE-T is used to render to various tile layouts. The performance of the compositer distributed with VTK is also measured. Each application processor renders about 7.4 million triangles (i.e. larger jobs have more triangles).

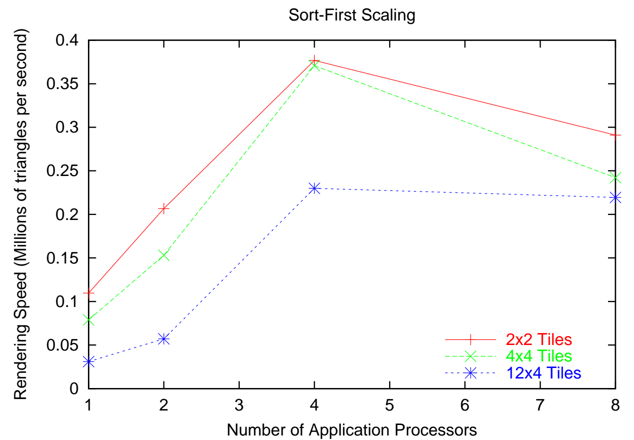


Figure 5: Scaling of parallel rendering using Chromium's `tilesort` SPU. Each client renders about 7.4 million triangles (i.e. larger jobs have more triangles). Measurements were taken using TCP/IP connections over a 100BASE-T network.

beyond that of a `vtkParallelRenderManager` unit, so we designed an object called `vtkIceTRenderer` to handle this. `vtkIceTRenderer` responds to ICE-T render requests and returns the appropriate images. The ICE-T render manager object will create ICE-T renderer objects for the user level application.

The ICE-T render manager also builds on the concept of image reduction. Unlike the image reduction of the parent `vtkParallelRenderManager` class, the ICE-T render manager does not reduce the size of the renderable viewport. Recall that in a tiled display, the overall display size is larger than any one image. ICE-T uses an extension of the *floating viewport* described in Moreland, Wylie, and Pavlakos [Moreland et al. 2001] to render images spanning multiple tiles and therefore reduce the number of times the geometry must be rendered.

# 7 Steering Station

Another important issue with rendering to tiled displays is providing an interface with which users can interact. The typical solution for VTK applications such as ParaView using a composite manager is to place the user interface at node 0 where the image is displayed [Law et al. 2001]. This is problematic with a tile display for several reasons. The most significant problem for us is the fact that, for security reasons, our display wall and driving cluster are in different rooms, separated by a bolted steel door. Instead, our user interface resides on a **steering station,** a separate PC located in view of the display and connected to the cluster via a standard Ethernet connection. Changes made in the user interface running on the steering station are then propagated to the images on the display wall.

It so happens that this exact functionality is already implemented by the desktop delivery objects described in Section 4 except that images are displayed on the server rather than shipped back the client. Therefore, the desktop delivery server object has a flag to select the display. If displaying to the client, the behavior is as described in Section 4. If displaying to the server, the render windows on the server side are never resized and images are not transferred back to the client. The client simply renders a very coarse representation of the geometry, which is a bounding box by default.

# 8 Conclusion

With the introduction of desktop delivery, Chromium, and ICE-T components into VTK, we have shown that VTK is a viable framework for cluster-based interactive applications that require remote display or display to high-resolution power walls. By using both the image-centric level of detail provided by the reduction factors in conjunction with the geometry-centric level of detail directly provided by VTK, we can achieve highly interactive rendering for almost any image transfer, compositing, or rendering speeds.

Our ability to abstractly swap the parallel rendering method is of tremendous importance. Each method of parallel rendering has its own strengths and weaknesses, and it may be imperative to pick the method that best fits the current usage for the system. For example, when using our sort last methods on our cluster (comprised of 128 Dell Precision 530 workstations, each with dual 2.0 GHz Pentium-4 Xeon CPUs, 1 GB RDRAM, a GeForce 3 graphics card, and interconnected with Myrinet 2000), we find that even with very little geometry we can achieve frame rates of only about 10 Hz for our tile displays due to the overhead of image compositing. However, as can be seen in Figure 4, with sufficiently large input geometry we can achieve very impressive rendering performance even on tiled displays.

In contrast, the Chromium `tilesort` SPU has been shown to be able to render with frame rates in excess of 40 Hz [Humphreys et al. 2002]. However, our current method of rendering has the number of Chromium "server" nodes fixed to the number of tiles and we see from Figure 5 that we get negative returns attempting to use more than about four application processes. Thus, we can only use our current Chromium configuration with modest sized data. It is conceivable to build other Chromium configurations that allow for better rendering performance of large data on tile displays, but race conditions existing in the Chromium network layer at the time of this writing have prohibited us from experimenting in this arena. These race conditions have also prohibited us from showing the scaling behavior past eight application processors in Figure 5.

# 9 Future Work

We have just begun the process of creating a viable production-quality tool to run on our clusters and display on our power walls. There is much work still to be done.

Our desktop delivery objects allow for any type of interface to be built on the client side. However, we currently have not built a GUI capable of much more than simple navigation controls. Ultimately, we would like a fully-featured visualization tool such as **ParaView** [Law et al. 2001] to be presented at the desktop. We currently have a contract with Kitware (www.kitware.com) to incorporate the technologies described in this paper into ParaView. The latest version of ParaView is available free from www.paraview.org.

The two methods of rendering to tile walls discussed have diametrical performance properties. Chromium's `tilesort` excels at providing fast frame rates for small amounts of geometry while ICE-T excels at rendering large amounts of geometry. We would like our application to be able to pick the most appropriate rendering method. Unfortunately, Chromium and ICE-T have to be launched in different ways. An application cannot easily switch between the two. Our proposed solution is to instead modify ICE-T to take advantage of data replication. By replicating the geometric data we can reduce the amount of image data that needs to be processed and can thereby greatly increase the speed of image composition.

While the concept of a steering station allows an arbitrarily complex user interface, it can be less than ideal. Often, we find the user walking to the wall for a closer look and back to the steering station to drive the application. We would like to incorporate a hand-held device that could move with the user and at least provide navigation controls.

## 10 Acknowledgments

This work was performed at Sandia National Laboratories. Sandia is a multi-program laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

We would also like to give a special thanks to Brian Wylie for his support, inspiration, and the encouragement to write this paper in the first place.

## References

AHRENS, J., LAW, C., SCHROEDER, W., MARTIN, K., AND PAPKA, M. 2000. A parallel approach for efficiently visualizing extremely large, time-varying datasets. Tech. Rep. LAUR-00-1620, Los Alamos National Laboratory.

AHRENS, J., BRISLAWN, K., MARTIN, K., GEVECI, B., LAW, C. C., AND PAPKA, M. E. 2001. Large-scale data visualization using paralle data streaming. *IEEE Computer Graphics and Applications 21*, 4 (July/August), 34–41.

HEERMANN, P. D. 1999. Production visualization for the ASCI one TeraFLOPS machine. In *Proceedings of Visualization '99*, 459–462.

HUMPHREYS, G., HOUSTON, M., NG, R., FRANK, R., AHERN, S., KIRCHNER, P. D., AND KLOSOWSKI, J. T. 2002. Chromium: A stream-processing framework for interactive rendering on clusters. In *Proceedings of ACM SIGGRAPH 2002*, vol. 21 of *acm Transactions on Graphics*, 693–702.

LAW, C. C., HENDERSON, A., AND AHRENS, J. 2001. An application architecture for large data visualization: A case study. In *Proceedings of the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, 125–128.

MORELAND, K., WYLIE, B., AND PAVLAKOS, C. 2001. 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*, 85–92.

SAMANTA, R., ZHENG, J., FUNKHOUSER, T., LI, K., AND SINGH, J. P. 1999. Load balancing for multi-projector rendering systems. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 107–116.

SCHROEDER, W., MARTIN, K., AND LORENSEN, B. 2002. *The Visualization Toolkit*, 3nd ed. Kitware, Inc. ISBN 1-930934-07-6.

SMITH, P. H., AND VAN ROSENDALE, J. 1998. Data and visualization corridors, report on the 1998 DVC workshop series. Tech. Rep. CACR-164, Center for Advanced Computing Research, September.

STOLL, G., ELDRIDGE, M., PATTERSON, D., WEBB, A., BERMAN, S., LEVY, R., CAYWOOD, C., TAVERIA, M., HUNT, S., AND HANRAHAN, P. 2001. Lighting-2: A high-performance display subsystem for PC clusters. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics, 141–148.

Figure 6: The VIEWS Corridor display. The display contains about 63 million pixels. The isosurface being displayed contains about 470 million triangles. Our VTK application can render this isosurface on our display in about 15 seconds. With coarser levels of both geometric and image detail, we can sustain a minimum frame rate of 5–10 Hz.

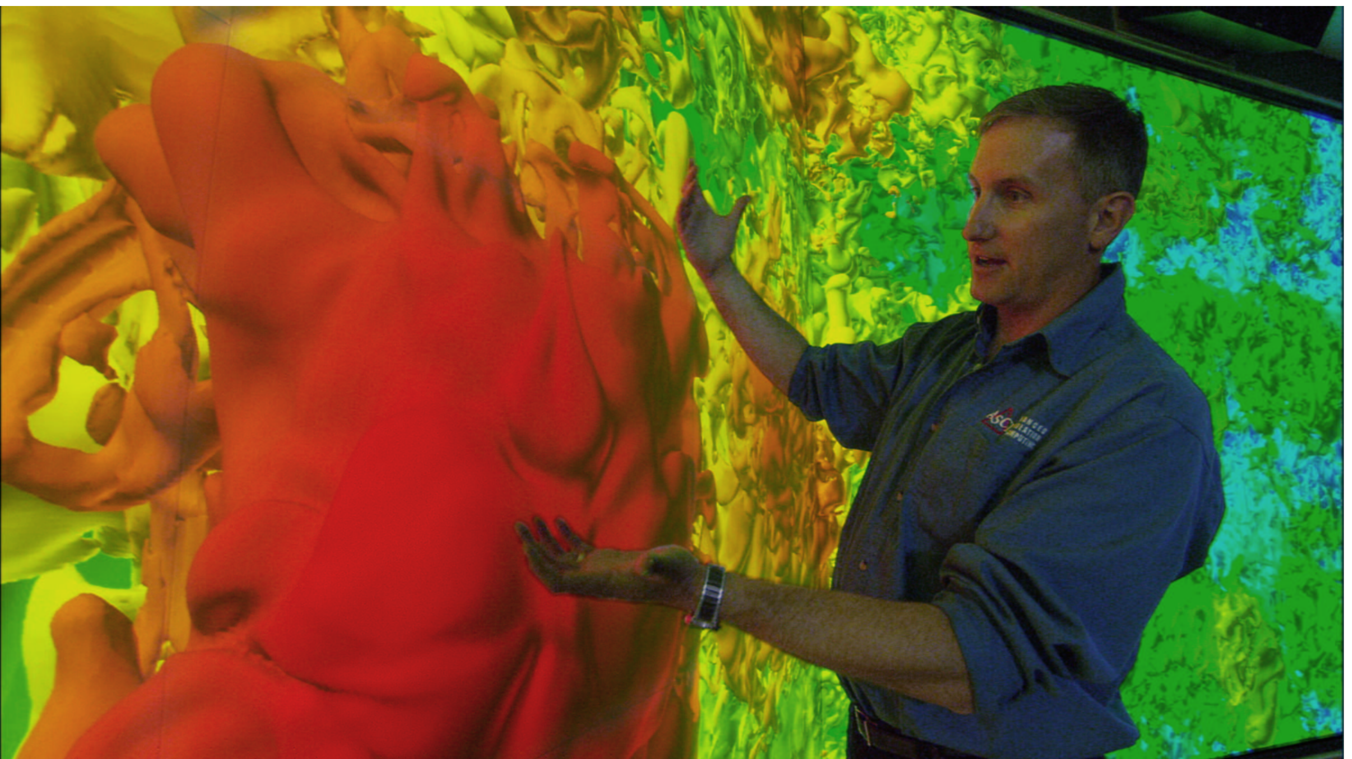

Figure 7: Our manager, Philip Heermann, inspecting a plume of gas. Philip can inspect the details of the plume without losing the context of the rest of the model.