# Parallel Unstructured Volume Rendering in ParaView

Kenneth Moreland[a], Lisa Avila[b], and Lee Ann Fisk[a]

[a]Sandia National Laboratories, 1515 Eubank SE, Albuquerque, NM, USA
[b]Kitware Inc., 28 Corporate Dr., Suite 204, Clifton Park, NY, USA

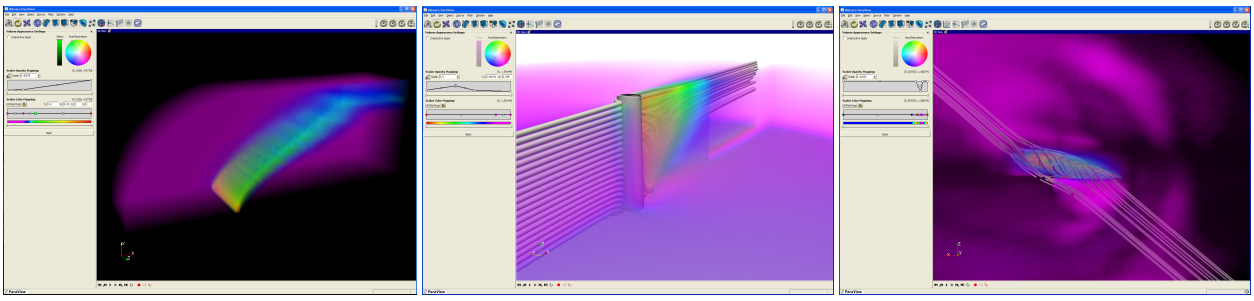

**Figure 1.** The blunt fin, oxygen post, and delta wing volume rendered with ParaView on a distributed memory cluster.

## ABSTRACT

ParaView is a popular open-source general-purpose scientific visualization application. One of the many visualization tools available within ParaView is the volume rendering of unstructured meshes. Volume rendering is a technique that renders a mesh as a translucent solid, thereby allowing the user to see every point in three-dimensional space simultaneously. Because volume rendering is computationally intensive, ParaView now employs a unique parallel rendering algorithm to speed the processes. The parallel rendering algorithm is very flexible. It works equally well for both volumes and surfaces, and can properly render the intersection of a volume and opaque polygonal surfaces. The parallel rendering algorithm can also render images for tiled displays. In this paper, we explore the implementation of parallel unstructured volume rendering in ParaView.

**Keywords:** ParaView, volume rendering, volume visualization, parallel rendering

## 1. INTRODUCTION

The technique of volume rendering has been in use for over two decades and has been applied to scientific visualization for nearly that long. Volume rendering is a popular visualization technique because it provides information about the entire data set and can lead to insightful views, such as those demonstrated in Figure 1.

To capitalize on its virtues, the ParaView application (http://www.paraview.org) incorporates unstructured grid volume rendering into its suite of scientific visualization tools. Volume rendering is a well studied problem with many known efficient algorithms. Nevertheless, volume rendering remains a compute intensive problem. For this reason, we choose to implement a parallel unstructured grid volume rendering algorithm.

In this paper we report on ParaView's implementation of parallel unstructured grid volume rendering. Following a brief review of previous work, we describe ParaView's serial implementation of volume rendering. We then explicate how we implement volume rendering on a distributed memory parallel computer. We conclude by analyzing the performance of ParaView's unstructured-grid volume rendering on a large visualization cluster.

Further author information (Send correspondence to Kenneth Moreland):
Kenneth Moreland: E-mail: kmorel@sandia.gov, Telephone: 1 505 844 8919
Lisa Avila: E-mail: lisa.avila@kitware.com, Telephone: 1 518 371 3971
Lee Ann Fisk: E-mail: lafisk@sandia.gov, Telephone: 1 505 844 2059

## 2. PREVIOUS WORK

In this section we give a brief overview of volume rendering algorithms. See Moreland[1] for a more comprehensive review. In particular, we focus on algorithms pertaining to volume rendering unstructured grids.

By far the most common unstructured grid volume rendering algorithm is projected tetrahedra.[2] This algorithm projects tetrahedra on the viewing plane, determines a set of triangles that represent the tetrahedra in the viewing plane, and then sends these triangles to graphics hardware for rasterization. The original Shirley and Tuchman[2] implementation uses a rough approximation of the "absorption plus emission" optical model.[3] Stein, Becker, and Max[4] improve the approximation by using texture hardware to provide the nonlinear interpolation of opacity in rasterized triangles. Röttger, Kraus, and Ertl[5] later introduce a pre-integration table to allow texture hardware to apply arbitrarily complicated 1D transfer functions to volume rendering.

As with any projection method, the projected tetrahedra algorithm requires cells to be projected in an order respective of their visibility (unless the optical model used is order independent). Simply sorting cells based on their centroids' distances from the viewer usually provides a reasonable approximation for the visibility ordering so long as the cells are not too elongated. Algorithms exist to provide a correct visibility ordering, albeit at the expense of extra computation or memory overhead.[4,6–8]

Several ray-cast algorithms are also known. Garrity[9] casts rays through an unstructured grid by tracing intersections through neighboring cells. Bunyk, Kaufman, and Silva[10] improve on the algorithm by performing the remaining cell intersections in bulk by projecting external faces. Programmable graphics hardware can also be used to perform ray casting.[11]

Several algorithms accelerate ray casting by employing sweep planes.[12–14] Farias, Mitchell, and Silva[15] combine the approaches of sweeping, projection, and A-buffers[16] to implement the ZSWEEP algorithm. Modern graphics hardware also supports a ZSWEEP-like rendering algorithm called HAVS.[17]

There are also a number of methods to perform volume rendering in parallel. Some divide screen real estate and assign each region to a process.[18–20] This is analogous to the sort-first class of parallel rendering algorithms for polygons.[21] This approach requires transferring geometry as the viewport changes, duplicating the geometry, or using a shared memory machine. Other parallel volume rendering algorithms statically partition geometry and combine image data ,[22–24] as is done with sort-last parallel rendering algorithms. This approach requires combining ray segments

## 3. SERIAL VOLUME RENDERING

ParaView's parallel volume rendering builds on the serial volume rendering algorithms it also uses. This section describes the implementation of serial unstructured volume rendering in ParaView. The following section expands the discussion to describe how they are applied to render grids distributed in parallel.

### 3.1. VTK Implementation

Although volume rendering of unstructured grids is a well-studied field, there are many known algorithms, and no single one of them stands out as the best solution in all cases. One technique may work well for large data but another will significantly out-perform it for smaller data sizes. The best technique to use on a computer with many CPUs but no dedicated GPU may be quite different than the one you would choose to use on a system with dual high-end GPUs and a slow CPU. For this reason, we implement several different unstructured-grid volume rendering algorithms for ParaView, and we have created a framework that allows researchers to easily include additional techniques in the future.

The base classes to support serial volume rendering in ParaView are implemented within the Visualization Toolkit (VTK),[25] which is one of the foundation libraries on which ParaView is built. In VTK, one C++ object is used to represent a data item in the scene including its position and orientation, another C++ object is employed to store the appearance parameters of that object, and a "mapper" object is tasked to render a representation of the data to the screen. Figure 2 shows the hierarchy of unstructured-grid volume mappers. The hierarchy is designed to allow abstraction of the algorithm used, the type of data rendered (structured or unstructured), or even the type of rendering performed (surface or volume).

All 3D mappers in VTK are derived from `vtkAbstractMapper3D`, with functionality common to all structured and unstructured volume rendering methods provided by `vtkAbstractVolumeMapper`. The `vtkUnstructuredGridVolumeMapper` class holds the pointer to the unstructured data, ensuring a match between the data and the rendering algorithm.


**Figure 2.** Inheritance diagram of VTK unstructured-grid mapper objects.

Four concrete rendering solutions based on well-known algorithms for unstructured grids are implemented. The simplest technique is the projected tetrahedra method from Shirley and Tuchman[2] that approximates the volume rendering of tetrahedral cells by using graphics hardware to render translucent triangles. Our current implementation approximates a visibility ordering by depth sorting cell centroids. The structure of our projected tetrahedra mapper is such that we can easily change the visibility ordering algorithm. More accurate results can be obtained with the ZSWEEP mapper based on the algorithm of Farias, Mitchell, and Silva[15] that sweeps a plane parallel to the view plane through the data in order to sort and render the cells. Finally, the ray caster is a flexible mapper that can support multiple techniques and automatically distributes rays amongst processors on a multi-processor (or multi-core) shared memory system. Currently available in VTK is the technique developed by Bunyk, Kaufman, and Silva[10] that transforms the data into screen space in order to accelerate the ray traversal process. This is the most memory intensive solution and therefore is not appropriate for large unstructured grids. ParaView also implements the HAVS algorithm.[17] HAVS is both faster and more accurate than the projected tetrahedra algorithm, but it only works on modern graphics cards with certain OpenGL extensions.

### 3.2. Intermixed Geometry

All of the techniques described in the previous section support opaque geometry intermixed with the volume. VTK employs a multi-pass rendering strategy whereby each mapper is required to render its opaque 3D elements in the first pass, followed by the translucent 3D elements in the second pass. Since the volume rendering process is converted to a geometric rendering process in the projected tetrahedra and HAVS techniques, OpenGL directly provides the intermixed geometry support (although there is some error, which is usually insignificant, where the surface intersects volume cells). ZSWEEP and ray casting are software rendering techniques, and they utilize the geometric results stored in the hardware depth buffer to support intermixed geometry. This is done by essentially clipping the unstructured grid with the depth values stored in the depth buffer, rendering only the portion of the data that lies closest to the viewer.

### 3.3. Optical Model

Every rendering system needs an optical model to determine the colors of objects. ParaView uses the popular "absorption plus emission" model defined by Max[3] for volume rendering. Given a viewing ray entering a volume at $a$ and exiting at $b$, the intensity of light reaching the eye (per wavelength) can be expressed as

$$I(a,b) = \int_a^b L(s)\tau(s)e^{-\int_a^s \tau(t)dt} + I_0 e^{-\int_a^b \tau(t)dt} \tag{1}$$

where $L$ is the luminance of the cloud along the ray, $\tau$ is the attenuation coefficient, and $I_0$ is the intensity of the background light. The derivation of this formula can be found in numerous works.[1,3,26] For simplicity, let us represent Equation 1 by substituting the functions $\alpha(a,b) = 1 - \exp(\int_a^b \tau(t)dt)$ and $c(a,b) = \int_a^b L(s)\tau(s)e^{-\int_a^s \tau(t)dt}$

$$I(a,b) = c(a,b) + I_0(1 - \alpha(a,b)) \tag{2}$$

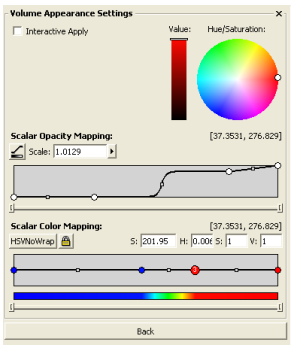


**Figure 3.** The Volume Appearance editor in ParaView.

Equation 2 is problematic to solve for all but very simple forms of the $L$ and $\tau$ functions. Thus, it is customary to break the integral into pieces. The integrals in the range $[a,b]$ can be broken into integrals in the range $[a,m]$ and $[m,b]$ as follows.[1]

$$I(a,b) = c(a,m) + (1 - \alpha(a,m))I(m,b) \tag{3}$$

An observant reader may notice that Equations 2 and 3 are really just instances of the Porter and Duff **over** operator.[27] This is convenient as the **over** operator is easy to compute and is readily available on graphics hardware.* As we break our integral into $n$ pieces, the computation becomes

$$I = [c_1, \alpha_1] \textbf{ over } [c_2, \alpha_2] \textbf{ over } \ldots \textbf{ over } [c_n, \alpha_n] \textbf{ over } [I_0, 0] \tag{4}$$

The volume rendering algorithms we use naturally break the integral by the discrete cells of the volumes on which they work. Our projected tetrahedra algorithm integrates a viewing ray through a cell using the same method as Stein, Becker, and Max.[4] Our HAVS algorithm integrates a viewing ray with a static partial pre-integration table.[1] Our software based algorithms (ray casting and ZSWEEP) use a pre-integrated table.[5] The computations for filling the table are accelerated with a static partial pre-integration table[1] and incremental integration.[11] We do not use pre-integrated tables with the projected tetrahedra and HAVS methods because the delay for rebuilding the table while editing the transfer function is irritating for users.

### 3.4. Volume Appearance Editor

The Volume Appearance editor in ParaView allows the user to control the transfer functions that map the scalar value to color and opacity. As can be seen in the image shown in Figure 3, the color and opacity mappings are independent although the interface provides a mechanism for locking the two functions together if so desired. In order to provide better control over the mappings with fewer control points, a modified piecewise Hermite function is employed. The user is able to control a sharpness coefficient per function segment interactively within the range of $[0.0, 1.0]$ where a sharpness of 0.0 yields a linear mapping and a sharpness of 1.0 leads to a piecewise constant solution. The user is also able to interactively adjust the location of the midpoint of the interpolated function. In Figure 3, the control nodes in the mapping functions are shown as circles, while the midpoint location is represented as a square. Both HSV and RGB are supported as the basis for color interpolation.

### 3.5. Interactive Rendering

Although ParaView uses very efficient volume rendering algorithms, volume rendering is, by its nature, computationally intensive. Volume rendering of large unstructured data is not yet an interactive process, however ParaView is an interactive scientific visualization application. To provide interactive rendering rates, ParaView renders a lower quality or approximate representation of the unstructured data while the user is actively manipulating the scene, with a higher quality technique employed to render the final image. HAVS has an internal level of detail parameter that automatically reduces detail

*To implement the **over** operator with OpenGL, we must change the blending function to (GL_ONE, GL_ONE_MINUS_SRC_-ALPHA), which is *not* the default.

for interactive rendering. On systems where HAVS is not available, large data may require a more approximate solution whereby only the outer faces of the unstructured grid are rendered as a translucent surface. Although not used as interactive techniques in ParaView, the ZSWEEP and ray casting solutions implemented in VTK do offer the opportunity to trade off accuracy for speed by generating a smaller image. This resulting image is rendered into the scene as a translucent texture-mapped polygon, providing fast bilinear interpolation of the reduced resolution image.

## 4. PARALLEL VOLUME RENDERING

In the previous section we discuss the implementation of volume rendering in ParaView for a serial application. In this section we extend that discussion to cover unstructured volume rendering on a distributed-memory parallel computer.

### 4.1. Approach

Our design of parallel volume rendering has several competing requirements.

- The implementation must work well on distributed parallel machines.
- There must be a minimal amount of geometry duplication.
- The solution must scale well.
- The code complexity must be minimized.
- The implementation must render translucent volumes and opaque surfaces, possibly intersecting, in the same scene.

These requirements, with the exception of the last one, are also requirements for parallel surface rendering within ParaView. ParaView's solution for surface rendering is a sort-last image-compositing library called IceT.[28–30] If we could apply our existing surface parallel rendering code to volumes, all of our requirements would be satisfied.

Of course, the problem is that the z-buffer operation we use to composite images of opaque surfaces does not apply to volumes. That does not mean that image compositing cannot be applied to rendered volumes. Ma[22] shows how a simple image compositing algorithm like binary swap can be used to composite rendered volumes, and our algorithm, described in the following two sections, is based largely on this system.

### 4.2. Color Blending

Consider a viewing vector $\vec{v}$ that intersects $n$ cells in the order $C_1, C_2, \ldots C_n$. Let us assume we have a function $\text{color}(C, \vec{r})$ that computes an RGBA color/opacity value for the intersection of a cell and a viewing ray. (All the volume rendering algorithms discussed in Section 3.1 inherently compute this function.) As we show in Section 3.3, the correct color for the viewing ray can be expressed in terms of the Porter and Duff[27] **over** operator.

$$\text{color}(C_1, \vec{v}) \textbf{ over } \text{color}(C_2, \vec{v}) \textbf{ over} \ldots \textbf{ over } \text{color}(C_n, \vec{v}) \tag{5}$$

When using the **over** operator, the order of operations is important. It is often said that one must composite in either front to back or back to front order. That statement is not strictly true. Although it is not commutative, the **over** operator is associative.

$$a \textbf{ over } (b \textbf{ over } c) = (a \textbf{ over } b) \textbf{ over } c$$

The benefit is that we can group our **over** operations in any way we want so long as we do not commute the values. Hence, we can group our cells in arbitrarily sized blocks, composite the blocks individually, and then composite those results together.

$$\begin{aligned} &\big(\text{color}(C_1, \vec{v}) \textbf{ over} \ldots \textbf{ over } \text{color}(C_a, \vec{v})\big) \textbf{ over} \\ &\big(\text{color}(C_{a+1}, \vec{v}) \textbf{ over} \ldots \textbf{ over } \text{color}(C_b, \vec{v})\big) \textbf{ over} \\ &\big(\text{color}(C_{b+1}, \vec{v}) \textbf{ over} \ldots \textbf{ over } \text{color}(C_c, \vec{v})\big) \textbf{ over} \ldots \textbf{ over} \\ &\big(\text{color}(C_{z+1}, \vec{v}) \textbf{ over} \ldots \textbf{ over } \text{color}(C_n, \vec{v})\big) \end{aligned} \tag{6}$$

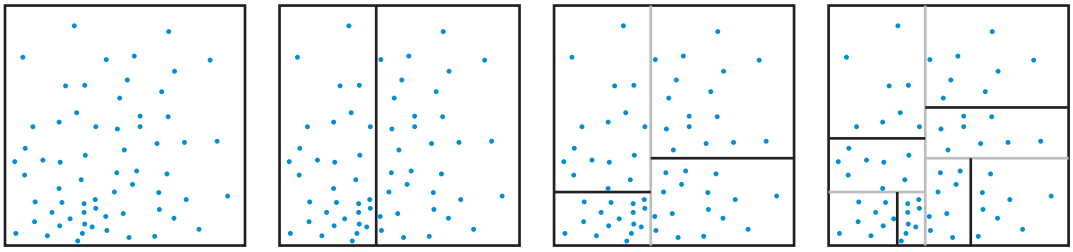

**Figure 4.** An example of building a kd-tree in two dimensions.

Consider what happens if all the cells in each group of Equation 6 are accessible to a single process. In this case, each process can locally combine the colors from all of its cells. This simplifies our rendering by allowing us to apply a serial volume rendering algorithm locally on each data partition, and then use an image compositing algorithm similar to the ones used for opaque surfaces.

There are several complications to this approach. First, on a distributed memory machine we need to make sure that each process holds a contiguous set of intersected cells for each viewing ray. Second, it is best to have the same view ordering of processes for all viewing rays to avoid making our image composition more complicated and less efficient. Third, to avoid moving data in between rendering frames on a distributed memory machine, we need to make sure that we divide the data in such a way that a proper view ordering exists for any viewpoint. Our solution involves finding a partitioning of the data which has a proper view ordering for all viewing rays from all viewpoints.

### 4.3. Data Partitioning

There are several partitioning strategies we can use to ensure a proper visibility ordering amongst all processes. Ma[22] uses a kd-tree to partition data. A kd-tree is a simple, recursive method for partitioning a $k$-dimensional space (where in our case $k$ is 3). The kd-tree divides a space as follows. We first pick an axis to divide the space by, and then choose a plane perpendicular to this axis. This plane divides the region into two smaller regions. We then recursively divide each subregion in the same way until we have the desired amount of regions. Figure 4 demonstrates partitioning a collection of points with a kd-tree.

In many ways, a kd-tree is well suited to our purposes. We can recursively view order regions by computing the location of the camera with respect to each dividing plane. Although the view ordering will change with respect to the viewpoint, a view ordering is guaranteed to exist and is easily found. We can also get a balanced distribution of cells by judiciously picking the location of planes with which we divide the data and thereby adjust the size of the regions to match the density of the volume.

Although kd-trees are used in other systems to divide regular data,[22] they are not used in previous systems to partition unstructured grids. The reason is that in order for the visibility ordering of the kd-tree regions to apply to the cell partitions, every cell must be completely contained within its corresponding region. The dividing planes of the kd-tree must not intersect any cells. Arranging the kd-tree in this way for a regular grid is trivial, but is simply not possible for a general unstructured grid.

Our unique solution is to simply change the grid to conform to the kd-tree we choose. We create a kd-tree that partitions the cells evenly. Inevitably, many cells will straddle regions. These cells are copied to all regions they straddle and then clipped by the bounds of each region. The result is a new unstructured grid where every region completely contains all cells within it. Thus, the view ordering of the kd-tree regions is also valid for the partitions of cells.

Ideally, we would like to skip the clipping step and force the clipping to happen during rendering. However, some important rendering algorithms do not support clipping well. For example, using a hardware clipping plane with the projected tetrahedra algorithm will not appropriately clip tetrahedra in the view direction because they are really drawn as flat triangles with special shading. This approach creates noticeable artifacts along the cutting planes which are particularly noticeable in vary transparent and fairly homogeneous regions. Hence, we instead use geometric clipping.

A pleasant side effect of our redistribution is that it ensures a reasonable load balancing of our rendering. During rendering, the data is guaranteed to be split evenly amongst processes cell-wise. This balancing is helpful as ParaView

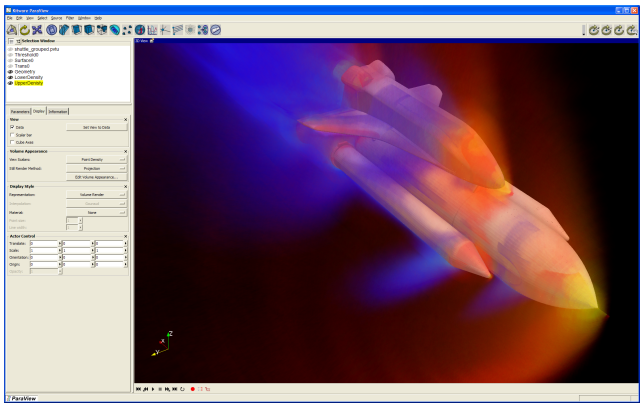

**Figure 5.** A NASA PLOT3D simulation of the space shuttle launch vehicle. The volume rendering represents air density. The shuttle surface is drawn for context.

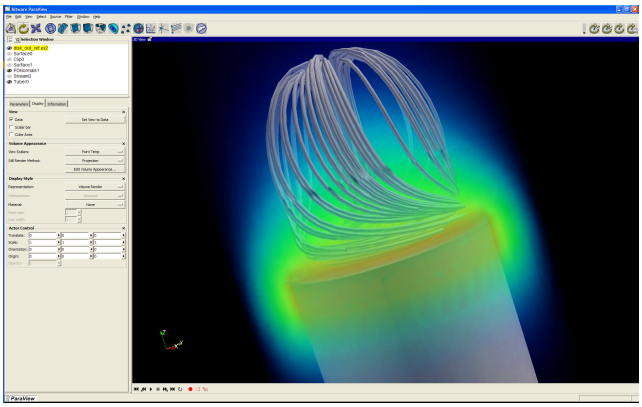

**Figure 6.** A simulation of air movement over a spinning heated disk. The volume rendering represents the temperature of the air while the streamlines display its movement as it spins, rises, and then falls.

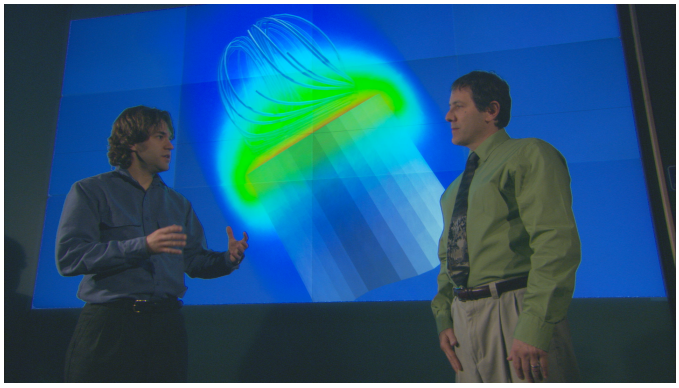

**Figure 7.** A high-resolution tile display driven by ParaView while performing parallel unstructured volume rendering.

does not ensure even distribution of unstructured data. For example, unstructured data readers rely on the layout of data in files for partitioning, and this layout may not be well balanced. Furthermore, some common operations such as threshold may have an unbalanced output even if the input is well balanced. However, the benefits of the load balancing are limited to the data size. Other factors that may affect rendering performance such as screen projection size or the number of external faces is not considered.

A major advantage of this approach is that it is applicable to both 2D polygons and 3D polyhedra as well as both opaque and translucent items. Consequently, we can apply our partitioning approach to mixes of volumes and polygons. We can draw volumes with surfaces of relevant objects for context as demonstrated in Figure 5. We can also combine volume rendering with other visualization techniques, such as streamlines as demonstrated in Figure 6 and elsewhere in this paper, to create even more information rich images.

Another major advantage of our approach is that we can apply our previously existing sort-last tile display techniques to volume rendering.[29,30] Figure 7 demonstrates using a high-resolution tile display in conjunction with volume rendering.

### 4.4. Kd-Tree Building

Our data partitioning relies directly on creating a kd-tree to create a partitioning of a large amount of data in a short amount of time. The process for building the kd-tree is a basic recursive algorithm.

1. Find all the centroids for all the cells. We use the centroids to quickly and deterministically place each cell in a region.

2. Choose a coordinate direction to split the region. This is usually the coordinate for which the region is the longest. We choose this axis because it prevents elongated regions and because the dividing plane has a minimal area so will probably intersect less cells.

3. Find the index of the "median" cell such that there is an equal number of cells to either side along the direction chosen in 2.

4. Partition the cells based on the median computed in step 3.

5. Apply steps 2 through 4 recursively on the subregions until we have the desired number of regions.

This process is called "recursive coordinate bisection" in the finite element world[31] and is analyzed by Berger and Bokhari.[32]

The computationally intensive part of creating this decomposition is the parallel median find. Our algorithm needs to not only find the median, but also rearrange the coordinates so that those less than the median are on one side, and coordinates greater than or equal to the median are on the other side. This problem is known as "selection," and it should be possible to do this with less computational complexity than an array sort ($O(n \log n)$).

We begin with the selection algorithm from Floyd and Rivest.[33] The appeal of this algorithm is that the number of comparisons required to rearrange the array is sub-linear on average with respect to the size of the array. Their serial algorithm recursively identifies and processes subintervals of array elements that are likely candidates for containing the median element. The elements of the candidate subinterval are rearranged into elements less than a contained proposed median and elements greater than the proposed median.

Parallelizing the algorithm is somewhat straightforward. Preceding the exchanges of array elements are tests of element values. In the parallel implementation, these array values may be in another process. Since the tests and exchanges run in consecutive array index order for a time before a test indicates they should stop, we prefetch a range of values each time we need to do an off processor test. This small enhancement has a noticeable effect on run times.

Another concern is that an artifact of the Floyd Rivest algorithm is that it exhibits its worst case behavior when long sequences of the same value occur in the array. In our problem domain, it is common to have these long sequences of a repeated value. It occurs in volume models when coordinates of cells are aligned along an axis, and stored together in the array of coordinates. Running time on models exhibiting this worst case behavior could take hours rather than the expected seconds. Our current implementation has a modified algorithm to detect duplicate entries and skip over these sequences, an essential change. Adding the check and correction does not increase the complexity of the algorithm, and returns the running time of these worst case arrays to normal.

## 5. RESULTS

The design of Parallel Volume Rendering in ParaView focuses on handling large quantities of data that often originate from high fidelity physical simulations on large supercomputers. ParaView is currently deployed at Sandia National Laboratories where it is used to visualize simulations from some of the fastest supercomputers in the world such as Red Storm.[34] One such example is a simulation of a fire test facility using the Sierra/Fuego/Syrinx/Calore physics code,[35] shown in Figure 8.

We demonstrate our system using six data sets displayed throughout the figures in the paper: heated disk, blunt fin, oxygen post, delta wing, Shuttle, and Fire. In addition, we test our system on a synthetic data set, which is simply a $100^3$ rectilinear grid that has been tetrahedralized and then had its points perturbed to ensure that cutting planes split the cells. The size of these data sets ranges over four orders of magnitude. In practice, we are most interested in rendering the large data sets in parallel. In fact, there is little practical reason to render the smaller data sets in parallel. However, we also want to ensure that our system performance does not degrade with smaller data sets, and we are interested in the behavior of our system for the full spectrum of data set sizes.

The timings in this section come from Sandia National Laboratories' Europa cluster. Each node in this cluster is a Dell Precision 530 workstation with dual 2.0 GHz Pentium-4 Xeon processors, 1 GB RDRAM, a GeForce FX 5900 Ultra graphics card, and a Myrinet 2000 interconnect. Although each node has two processors per node, only one is used in the experiments. In timings that involve rendering, the projected tetrahedra method is used to generate $800 \times 800$ images. We use the projected tetrahedra method because it is our fastest rendering method available on our cluster (and therefore

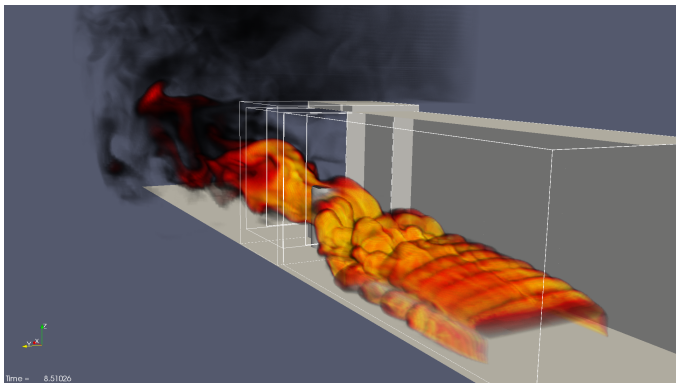

**Figure 8.** A simulation of Sandia National Laboratories' fire test facility. The simulation was performed on Red Storm with the Sierra/Fuego/Syrinx/Calore physics code. The visualization is performed with ParaView. Polygonal representation of the environment gives context to the volume rendering of the fire.

should give the best representation of the parallel rendering overheads), because it has the most predictable rendering speeds (which is almost linear with respect to data size for all but the smallest data sets), and because it is the most commonly used method.

### 5.1. Data Partitioning Overhead

ParaView's parallel volume rendering borrows much from well studied serial volume rendering and parallel image compositing algorithms. As such, we will not specifically analyze the performance of these parts of the algorithm here. The main unique feature, and potential drawback, of our method is the splitting of cells by kd-tree region boundaries. For our approach to be practical, the growth in tetrahedra as we divide regions must be manageable.

Table 1 demonstrates how data sets grow as they are divided by kd-tree region boundaries. Empty entries in the table occur where the data size is too large to handle with the given number of processes. As expected, the number of tetrahedra grows with every partition. Figure 9 shows how many tetrahedra a kd-tree level adds with respect to the data size. On average, the data size grows by about 15%, a factor represented by a dashed line in the figure. This factor is manageable because adding a single level to the kd-tree doubles the number of partitions.

There is, however, some variance in the data sets' rate of growth with actual values ranging between 5% and 30%. Given that we are starting with unstructured grids, having variance in the cuts is not surprising. We can, however, make two observations about the variance. One observation is that the first cuts usually have a smaller growth than the later ones. Another observation is that the larger data sets have a smaller growth, which suggests that the growth may really be slightly sub-linear with respect to the original data size.

Another metric of our system is the time it takes to partition and clip the data. This partitioning does not need to be done every time we render, but it is performed whenever a change is made to the data. Figure 10 shows, for each data set, the time it takes to partition. A general trend is for the partitioning time to reduce with larger jobs. However, there

**Table 1.** Size of Data Sets (in Tetrahedra)

| | Number of Kd-Tree Partitions | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 32 | 64 |
| Heated Disk | 44,832 | 54,028 | 69,227 | 83,844 | 104,570 | 136,519 | 179,274 |
| Blunt Fin | 187,395 | 206,460 | 241,676 | 289,503 | 346,859 | 406,874 | 504,961 |
| Oxygen Post | 513,375 | 559,921 | 606,467 | 725,017 | 846,378 | 994,142 | 1,170,188 |
| Delta Wing | 1,005,675 | 1,152,528 | 1,255,349 | 1,442,323 | 1,620,487 | 1,856,141 | 2,143,737 |
| Synthetic | 5,000,000 | 5,250,564 | 5,509,581 | 5,771,703 | 6,298,454 | 6,860,217 | 7,454,184 |
| Shuttle | 5,315,388 | - | 5,844,248 | 6,452,930 | 7,296,891 | 8,258,916 | 9,587,031 |
| Fire | 14,094,684 | - | - | 15,447,150 | 16,079,250 | 17,143,521 | 18,532,405 |

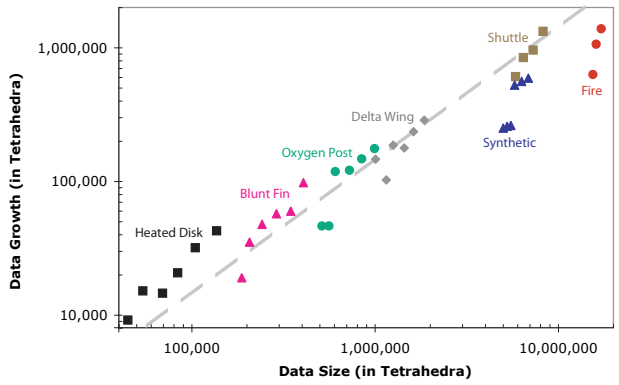


**Figure 9.** Plot of data size growth. The x-axis is the data size in tetrahedra. The y-axis is the number of tetrahedra the data set grows when adding one level to the kd-tree. Both axes have a logarithmic scale.

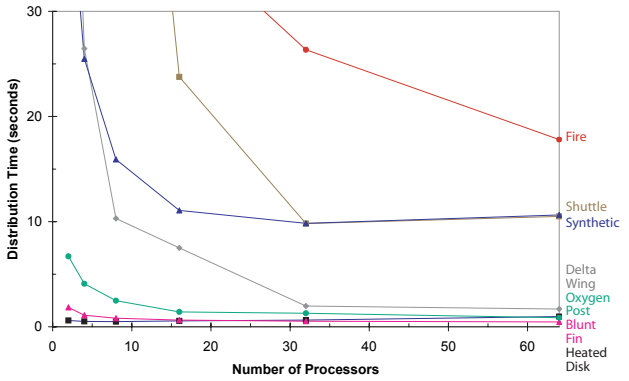


**Figure 10.** Plot of the time to distribute and clip the rendered tetrahedra of each experimental data set for various numbers of processors.

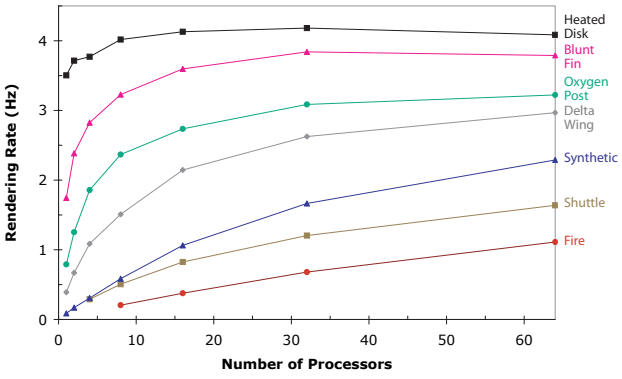


**Figure 11.** Plot of the rendering rates of each experimental data set for various numbers of processors.

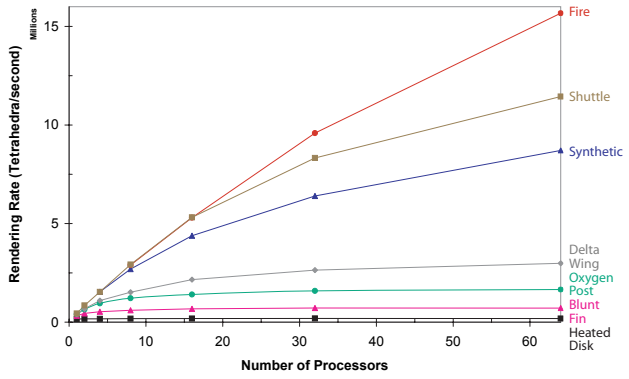


**Figure 12.** Plot of the rendering rates of each experimental data set for various numbers of processors.

also appears to be a lower limit on the time to distribute the data that increases with the data size. This is probably due to the operations that do not change much with the number of processors such as building the kd-tree or transferring data amongst processors.

## 5.2. Rendering Rates

We also analyze the total rendering rate of the system. Figure 11 plots the rendering rate of the system on each data set across a range of processors. In general, the rendering speed increases with the number of processors, although there are diminishing returns when adding more processors than necessary for a given data set. As with any parallel algorithm, we expect diminishing returns due to the parts of the algorithm that do not become faster with extra processors such as image readback and compositing. Although many of the rendering rates are too low for user interaction, ParaView's level of detail rendering,[36] described in Section 3.5, makes it perfectly adequate.

Figure 12 also plots the same rendering rates except that the rates are given in tetrahedra per second rather than frames per second. The rendering rate is given with respect to the original data set size, not the inflated size after distribution and clipping. This second plot shows that the system has a bigger speedup for the larger data sets even if the overall frame rate is lower. The general trends in the plots imply that the rendering rate of the largest data sets would increase were we able to use more processors.

## 6. DISCUSSION

This paper presents parallel unstructured volume rendering in ParaView. The key idea of our algorithm is to divide our data by a kd-tree in such a way that we can apply a traditional sort-last compositing algorithm.

There are a few disadvantages to our approach. First, the redistribution requires ParaView to hold at least two copies of the data at some time. With the current state of hardware, rendering rates drop impractically low before this becomes a problem. Second, splitting cells makes the data set size grow with the number of processes. We show in Section 5.1 that the data growth is manageable and the solution is still scalable. Third, changes in viewable geometry require lengthy redistribution.

Of course, our system also has advantages that we feel outweigh the disadvantages. The approach allows us to apply our existing volume rendering and compositing codes to parallel unstructured volume rendering. Not only does this increase our code reuse and simplify our code, but it also gives us the freedom to apply the largest variety of algorithms to the problem. For example, an approach that allows for the combination of viewing ray segments through interlocking non-convex partitions[23,24] would be difficult to combine with GPU accelerated rendering.

Our parallel rendering approach also has the advantage of working well on a distributed-memory parallel computer. Once cells are distributed amongst processes, no further transfer of cells is required. The only communication necessary for rendering is of image data for compositing. This is not true for many other parallel rendering approaches.[18–20]

Our parallel rendering algorithm maintains the advantage of directly rendering unstructured grids. Another approach to the problem could be to sample the volume in a regular grid and use an algorithm like Ma.[22] Although a simple idea, this solution is deceptively difficult to implement. Large unstructured meshes must have a large number of samples to yield useful renderings. This would necessarily have to happen in parallel, and on a distributed memory would require distribution of the unstructured grid or samples. Overall, the extra time to distribute and extra memory to store samples would be at least as large as the overhead of our approach. Furthermore, there is always the issue of the appropriate level of sampling. Many real-world unstructured grids have varying cell sizes, which is a big reason for using an unstructured grid in the first place. Sampling unstructured grids usually results in oversampling in some areas and undersampling in others. Leven and Corso[37] solve this problem by creating a hierarchy of grids, but the system requires significant preprocessing time and generates data much larger than the original input. Neither of these side effects are acceptable by our users. In contrast, our solution simply renders the unstructured grid natively.

## 7. ACKNOWLEDGMENTS

We would like to thank those who have contributed their data. The blunt fin, oxygen post, delta wing, and shuttle data sets are courtesy of the NASA Advanced Supercomputing (NAS) Division.† The fire data set is courtesy of Sheldon Tieszen and the Sierra/Fuego/Syrinx/Calore group.

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

1. K. Moreland, *Fast High Accuracy Volume Rendering*. PhD thesis, University of New Mexico, July 2004. http://www.cs.unm.edu/~kmorel/.
2. P. Shirley and A. Tuchman, "A polygonal approximation to direct scalar volume rendering," in *Computer Graphics (Proceedings San Diego Workshop on Volume Visualization)*, **24**, pp. 63–70, December 1990.
3. N. Max, "Optical models for direct volume rendering," *IEEE Transactions on Visualization and Computer Graphics* **1**, pp. 99–108, June 1995.
4. C. Stein, B. Becker, and N. Max, "Sorting and hardware assisted rendering for volume visualization," in *Proceedings of the 1994 Symposium on Volume Visualization*, pp. 83–89, October 1994.
5. S. Röttger, M. Kraus, and T. Ertl, "Hardware-accelerated volume and isosurface rendering based on cell-projection," in *Proceedings of IEEE Visualization 2000*, pp. 109–116, October 2000.
6. J. Comba, J. T. Klosowski, N. Max, J. S. B. Mitchell, C. T. Silva, and P. L. Williams, "Fast polyhedral cell sorting for interactive rendering of unstructured grids," *Computer Graphics Forum (Eurographics '99)* **18**, pp. 369–376, April/June 1999.

†It would be wonderful if NAS would post these on their web site again.

7. S. Krishnan, C. T. Silva, and B. Wei, "A hardware-assisted visibility-ordering algorithm with applications to volume rendering," in *Proceedings of Data Visualization (Eurographics/IEEE Symposium on Visualization)*, pp. 233–242, 2001.
8. P. L. Williams, "Visibility ordering meshed polyhedra," *ACM Transactions on Graphics* **11**, pp. 103–126, April 1992.
9. M. P. Garrity, "Raytracing irregular volume data," in *Computer Graphics (Proceedings of the San Diego Workshop on Volume Visualization)*, **24**, pp. 35–40, November 1990.
10. P. Bunyk, A. Kaufman, and C. Silva, "Simple, fast, and robust ray casting of irregular grids," in *Proceedings of the 1997 Scientific Visualization Conference (Dagstuhl '97)*, pp. 30–36, 1997.
11. M. Weiler, M. Kraus, M. Merz, and T. Ertl, "Hardware-based ray casting for tetrahedral meshes," in *Procceedings of IEEE Visualization 2003*, pp. 333–340, October 2003.
12. C. Giertsen, "Volume visualization of sparse irregular meshes," *IEEE Computer Graphics and Applications* **12**, pp. 40–48, March 1992.
13. C. T. Silva and J. S. B. Mitchell, "The lazy sweep ray casting algorithm for rendering irregular grids," *IEEE Transactions on Visualization and Computer Graphics* **3**, pp. 142–157, April-June 1997.
14. R. Yagel, D. M. Reed, A. Law, P.-W. Shih, and N. Shareef, "Hardware assisted volume rendering of unstructured grids by incremental slicing," in *Proceedings of the 1996 Symposium on Volume Visualization*, pp. 55–62, 1996.
15. R. Farias, J. S. B. Mitchell, and C. T. Silva, "ZSWEEP: An efficient and exact projection algorithm for unstructured volume rendering," in *Proceedings of the ACM/IEEE Volume Visualization and Graphics Symposium*, pp. 91–99, 2000.
16. L. Carpenter, "The A-buffer, an antialiased hidden surface method," in *Computer Graphics (Proceedings of ACM SIGGRAPH 84)*, **18**, pp. 103–108, July 1984.
17. S. P. Callahan, J. L. D. Comba, P. Shirley, and C. T. Silva, "Interactive rendering of large unstructured grids using dynamic level-of-detail," in *Proceedings of IEEE Visualization 2005*, pp. 199–206, October 2005.
18. R. Farias and C. T. Silva, "Parallelizing the ZSWEEP algorithm for distributed-shared memory architectures," in *Proceedings of the International Volume Graphics Workshop, 2001*, 2001.
19. M. E. Palmer and S. Taylor, "Rotation invariant partitioning for concurrent scientific visualization," in *Procedings of Parallel Computational Fluid Dynamics '94*, Elsevier Science Publishers B.V., 1994.
20. S. Uselton, "Volume rendering for computational fluid dynamics: Initial results," Tech. Rep. RNR-91-026, Nasa Ames Research Center, 1991.
21. S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A sorting classification of parallel rendering," *IEEE Computer Graphics and Applications* , pp. 23–32, July 1994.
22. K.-L. Ma, "Parallel volume rendering using binary-swap image composition," *IEEE Computer Graphics and Applications* **14**, pp. 59–68, July 1994.
23. K.-L. Ma and T. W. Crockett, "A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data," in *Proceedings of the 1997 Symposium on Parallel Rendering*, pp. 95–104, October 1997.
24. C. T. Silva, *Parallel Volume Rendering of Irregular Grids*. PhD thesis, State University of New York at Stony Brook, November 1996.
25. W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit*, Kitware, Inc., 3rd ed., 2002.
26. P. Sabella, "A rendering algorithm for visualizing 3D scalar fields," in *Computer Graphics (ACM SIGGRAPH 88)*, **22**, pp. 51–58, August 1988.
27. T. Porter and T. Duff, "Compositing digital images," in *Computer Graphics (ACM SIGGRAPH 84)*, **18**, pp. 253–259, July 1984.
28. A. Cedilnik, B. Geveci, K. Moreland, J. Ahrens, and J. Favre, "Remote large data visualization in the ParaView framework," in *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, May 2006.
29. K. Moreland and D. Thompson, "From cluster to wall with VTK," in *Proceedings of IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pp. 25–31, October 2003.
30. K. Moreland, B. Wylie, and C. Pavlakos, "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*, pp. 85–92, October 2001.
31. K. Devine, E. Boman, R. Heapby, B. Hendrickson, and C. Vaughan, "Zoltan data management service for parallel dynamic applications," *Computing in Science and Engineering* **4**, pp. 90–97, March/April 2002.
32. M. J. Berger and S. H. Bokhari, "A partitioning strategy for nonuniform problems on multiprocessors," *IEEE Transactions on Computing* **36**(5), pp. 570–580, 1987.
33. R. W. Floyd and R. L. Rivest, "Expected time bounds for selection," *Communications of the ACM* **18**, pp. 165–172, March 1975.
34. K. L. Jefferson and J. E. Sturtevant, "Red storm usage model: Version 1.12," Tech. Rep. SAND2005-6926, Sandia National Laboratories, Albuquerque, NM, 2005.
35. C. D. Moen, G. H. Evans, S. P. Domino, and S. P. Burns, "A multi-mechanics approach to computational heat transfer," in *ASME International Mechanical Engineering Congress and Exposition*, (New Orleans, LA), November 2002.
36. A. H. Squillacote, *The ParaView Guide*, Kitware, Inc., ParaView 2.4 ed., 2006. ISBN 1-930934-17-3.
37. J. Leven and J. Corso, "Interactive visualization of unstructured grids using hierarchical 3D textures," in *Proceedings of the 2002 IEEE Symposium on Volume Visualization and Graphics*, pp. 37–44, October 2002.