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

A Scalable, Hybrid Scheme for Volume Rendering Massive Data Sets†

Hank Childs[1,2] Mark Duchaineau[1] and Kwan-Liu Ma[2]

[1]Lawrence Livermore National Laboratory
[2]University of California at Davis

Abstract

We introduce a parallel, distributed memory algorithm for volume rendering massive data sets. The algorithm's scalability has been demonstrated up to 400 processors, rendering one hundred million unstructured elements in under one second. The heart of the algorithm is a hybrid approach that parallelizes over both the elements of the input data and over the pixels of the output image. At each stage of the algorithm, there are strong limits on how much work each processor performs, ensuring good parallel efficiency. The algorithm is sample-based. We present two techniques for calculating the sample points: a 3D rasterization technique and a kernel-based technique, which trade off between speed and generality. Finally, the algorithm is very flexible. It can be deployed in general purpose visualization tools and can also support diverse mesh types, ranging from structured grids to curvilinear and unstructured meshes to point clouds.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Distributed/Network Graphics; I.3.3 [Computer Graphics]: Picture/Image Generation; and I.4.1 [Computer Graphics]: Sampling

1. Introduction

The power and capacity of parallel supercomputers are growing at a fast pace, enhancing scientists' ability to simulate and study complex physical phenomena at increasingly greater accuracy. This added accuracy often is the key to advances in their studies. Their 3D simulations can generate large volume data sets, which are best visualized using direct volume rendering. Volume rendering is particularly effective in displaying complex 3D structures and features at varying scales. Real-time volume rendering has become feasible with commodity graphics hardware, but it requires the data to fit in video memory. Because the size of the data sets generated by these simulations are currently in the terascale regime, and will soon reach petascale, we must seek a distributed-memory, parallel rendering solution.

We have developed a new, massively parallel, distributed-memory, volume rendering algorithm. It is designed to run on the same parallel supercomputers used to run terascale/petascale simulations. It has demonstrated excellent scalability and been used on some of the largest simulations ever performed. The algorithm can be applied to many diverse types of meshes, ranging from structured grids to curvilinear and unstructured meshes to point clouds. It is implemented inside of VisIt [CBB*], a richly featured visualization and data analysis application focusing on large data sets. Throughout the development of this algorithm, a major goal was to develop a technique that would seamlessly integrate into a general-purpose visualization tool. This required us to avoid schemes that place special requirements on I/O, communication, available hardware, etc.

In a distributed memory environment, adaptive techniques for dividing work among the processors are not possible due to data ownership issues. In this environment, data must be partitioned among the processors and each processor is responsible for calculating its portion of the picture. Of course, load balancing issues can occur, especially with irregular data or when camera placement focuses on a small subregion of the volume. This requires us to develop data partitioning strategies that ensure even workloads on each processor. The quality of these strategies dictate the degree of scalability we can achieve at high processor counts.

† This is UC report UCRL-CONF-218622

We employ a multi-phase approach for distributing work. This technique is a hybrid of previous techniques and allows us to incorporate their best aspects with respect to balanced processing. During each phase of our algorithm, our data organization naturally enforces hard limits on how much work each processor can do. This creates excellent load balancing which in turn creates excellent scalability.

In this paper, we present the details of our algorithm, a scaling study, and some applications. Our algorithm is sample-based, which requires us to implement custom sampling techniques. We present a 3D rasterization technique and a kernel-based technique, which trade off between speed and quality, respectively. The scaling study shows linear scalability of our algorithm up to four hundred processors. In addition, we present the performance of this algorithm for diverse mesh types: a one billion-particle point cloud simulation, a 27 billion-element rectilinear mesh simulation, and a one hundred-forty million element unstructured mesh simulation.

2. Related Work

Among the parallel rendering algorithms specifically designed for visualizing irregular-grid volume data, the most relevant example for this work is the sort-last cell-projection algorithm introduced by Ma and Crockett [MC97]. This distributed-memory algorithm achieves high performance by completely overlapping rendering and compositing, and by keeping sorting costs to a minimum. However, its high scalability partially relies on manual setting of communication parameters. The other class of algorithms, those that require a view-dependent sorting step, are not feasible when rendering terascale and petascale data sets.

Wang, Gao and Shen [WGS] use a multi-resolution wavelet tree to allow parallel volume rendering of large data in an error-guided fashion. In contrast, our goal is to render the data at its highest possible resolution.

There is a growing interest in parallel GPU-based volume rendering [LM, SMW*, CMF]. Nevertheless, a PC cluster that is capable of rendering irregular-grid data sets at the scale we are faced would be prohibitively expensive to build. Our algorithm is designed to harness the power of parallel supercomputers. Another option is to utilize geographically distributed computing resources, which, if appropriately coordinated, could become very attractive. Gao et al. demonstrated such an approach with parallel rendering of large volume data distributed over a wide area network [GHJA].

3. Algorithm Overview

The algorithm itself consists of two high-level phases. In the first phase, samples are generated from the input data set. In the second phase, those samples are classified and composited to form the final picture.

Our goal in the sampling phase, for a given view frustum, is to determine the value of billions of samples along millions of rays, where there is one ray for each pixel of the image and thousands of samples along each ray. In abstract terms, a sampling algorithm takes a data set as input, as well as parameters describing the volume rendering requirements: the view frustum (i.e. camera location, view direction, view angle, and near and far clipping planes), the screen size (number of pixels in width, W, and height, H), and the number of samples, S, to take along each ray.

Given these inputs, the goal of a sampling algorithm is to create an output with the following properties:

1. A logically structured grid, G, of dimensions WxHxS
2. A field F defined on G, which has sampled the values of the input data set
3. Each F[w, h, s] corresponds to the value of s^{th} sample along the ray corresponding to pixel (w, h) of the output image for that view frustum

Given a black box that performs a sampling algorithm, it is trivial to implement an algorithm for the second, compositing phase. For each pixel (w, h) and each sample for that pixel, we can classify the field value based on some user defined transfer function. Then the resulting colors and opacities can be composited using the "over" function (front-to-back compositing) to create the color for that pixel. Doing this for all pixels yields the final, volume rendered picture.

We also note that a sample-based approach makes for easy integration with renderings of standard geometric primitives. An additional image input to the compositing module can specify background colors and prematurely terminate rays when they reach the depth of the first encountered geometric primitive for that pixel. This image is generated on a previous render pass.

By using samples as our fundamental unit, we have made sacrifices. We lose the ability to resolve complex interactions between elements, like the type addressed in [CMSW04]. But, because the samples are calculated on a per view frustum-basis, this issue is greatly mitigated. It is our belief that, with a sufficient number of samples per ray and by taking care in assigning the individual sample values, a sample-based method can provide good results. This approach of choosing a representative value for a sample is reminiscent of the clustering approach presented in [HE03], although at a different resolution.

4. Hybrid Sampling Algorithm

In the following subsections, we give an overview of the data management of the hybrid sampling scheme, without consideration of how to infer the values of the sample points. These subsections address how to organize the data in an efficient way for parallelization and how to store the field F without exceeding primary memory. The purpose of our

technique is to address the load balancing issues in a distributed memory environment. In a shared memory environment, such as the one described in [DPH*03], many of these load balancing issues do not occur because the parallelization across image space can be done adaptively.

The principal contribution of our data organization scheme is obtaining good load balance for meshes with great variation in the spatial density of their elements. For these meshes, in the extreme, the number of elements within a region of space can differ by orders of magnitude. Schemes that parallelize over the output image's pixels suffer extreme load imbalance with these meshes, because some pixels will cover many more elements within their projection than others. Similarly, schemes that parallelize over the input data set's elements suffer extreme load imbalance, because some elements will take up disproportionately large portions of the view frustum.

Our multi-stage algorithm, presented in the following subsections, is well balanced in the amount of elements *and* the portion of the view frustum processed. Our hybrid scheme utilizes the best portions of schemes that parallelize over the output's image and schemes that parallelize over the output's elements, while avoiding their pitfalls. It does this by processing the data in stages. In the first stage, it parallelizes across the elements of the input data set, but defers processing of the large elements. In a subsequent stage, it parallelizes across the output image and only then processes the large elements. Again, this organization of the data processing places hard limits on the work performed at each stage, ensuring good parallel efficiency.

The processing of meshes with great variation in the spatial density of their elements is a very important special case. The camera transformation often creates meshes of this form. An example is when the camera is placed in the middle of the data set, which often happens during fly-throughs in movies. In this case, even if the input mesh has uniform-sized elements, elements that are near the camera will occupy orders of magnitude larger portions of the view frustum than those farther from the camera.

Our sampling algorithm contains a total of three stages, characterized as Small-Element Sampling, Communication, and Large-Element Sampling. Also, the algorithm is a dual partition scheme; one partition is of the input data, the other partition is of the rays. In the Small-Element Sampling stage, the first partition is used. The Communication stage re-distributes between the two partitions. And the Large-Element Sampling stage uses the second partition.

Section 4.1 discusses the two partitions, 4.2 discusses the three phases of the algorithms, and 4.3 discusses the load balancing.

4.1. Partitions

The partition of the input data is done by the greater VisIt system and is guaranteed to assign approximately equal numbers of elements to each processor. Although we might be able to modestly reduce communicate costs by re-partitioning the data, we do not employ this technique. First, given the massive size of the data sets we are operating on, it is not viable to re-partition all of the data for each rendering. The only viable re-partitioning scheme would be to re-partition one time and use that for all subsequent renderings. However, even if we did perform a one-time re-partition, it is difficult to find one that will truly help with parallel efficiency. It is often not possible to create partitionings that are balanced in both number of elements and spatial footprint. Further, this will not mitigate the issue when the camera is in the middle of the data set.

The second partition is of the samples. We start by dividing the image's pixels among the processors. The division of the samples then follows naturally. Our goal with this partition is to re-assign the samples so that each processor can composite its pixels with no further communication.

4.2. Sampling

We define "small" elements as elements that cover a small number of samples, for example less than one hundred. Similarly, "large" elements are elements that cover more than one hundred samples. We describe how large elements are identified later in this paper.

In the first, Small-Element Sampling stage, we sample each small element and defer sampling of large elements. This stage has two outputs. One output is the large elements that are not sampled. The other is the grid G and the partially populated field F. Because we focus on "small" elements and we know that each processor is working on an approximately equal sized subset of the elements, we know that no processor will see a large number of samples. This reasoning will be further formalized in subsection 4.3. Our implementation of the structure that stores F is able to scale its size based on how many samples have been encountered. Thus, this counteracts the problem where F can exceed the size of primary memory, because we only have to store the samples we encounter and we will encounter only a small number of samples.

The second, Communication stage, re-distributes the output of the first stage to honor the second partitioning. The second partition is created dynamically at this point in the algorithm. By waiting until the first stage has completed, we can assign the pixels so that the maximum number of samples and elements remain on their processor of origin, minimizing communication. This technique of dynamically assigning pixels to processors to minimize total communication was also used in the hybrid scheme of [SFLS00],

although their application was to surface rendering, not volume rendering.

The communication stage consists of "all-to-all" communications between the processors. Sample points are communicated among the processors, using the partition to determine their destination. When sending a large element, we first examine its bounding box, and determine which portions of the image space partition the bounding box overlaps. We then send this element to the set of processors that need to sample it in the next stage. At the end of this stage each processor contains all of the data (as either samples or elements) necessary to create its portion of the image with no further communication.

Our scheme does not take advantage of the optimization that allows contiguous samples to be pre-composited, allowing a few bytes to take their place. This optimization is not well suited to the sampling scheme in Section 6. But the need for this optimization is greatly mitigated by the high network bandwidth on modern supercomputers. The sampling stage dominates the algorithm, making this shortcoming a non-issue.

The third, Large Element Sampling stage samples the remaining elements (all of which are "large") and adds them to the field F. The large elements are sampled selectively. Samples outside a processor's portion of the image space partition are not examined. If an element spans two portions of the partition, then the element will be sampled entirely by the corresponding two processors, but no part of it will be sampled twice.

At the end of this stage, all elements have been sampled and the field F is fully populated. This is the output of our sampling algorithm, which is now well suited for compositing. Even though F is distributed across the processors, each ray has all of its samples on the same processor. So compositing can take place with the only necessary communication being the collection of the pixel colors to the master processor at the end of the algorithm. The entire pipeline can be seen in Figure 1.

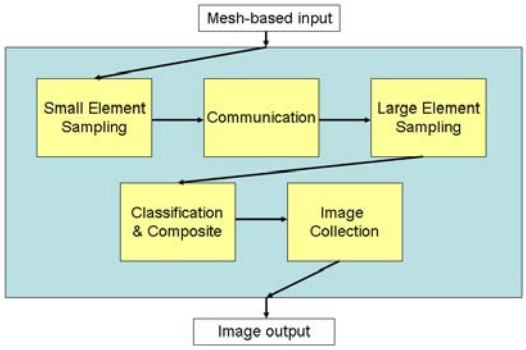

Figure 1: *The volume rendering portion of the pipeline.*

4.3. Load Balancing

Our goal is to maximize parallel efficiency. In this subsection, we motivate how our three-stage approach, outlined in the last subsection, helps to accomplish this goal.

Before we analyze the costs of each stage, consider the cost of doing sampling, whether it is in the first or third stage. When sampling E elements, there is an overhead with the processing of each individual element. And there is also a cost for each of the S_i sample points updated when processing element E_i. If $S=\sum S_i$, then the overhead for processing the elements is O(S + E).

How much work takes place in the first stage? Because we do not process large elements, we can easily bound this number. We do not process any elements that contain more than a constant number of samples, α. So the work in the first stage is $O(S + E) = O(\alpha*E + E) = O(E)$. Of course, asymptotic analysis can be misleading. In this case, however, it gives us an actual result. There are strict limits on the amount of sampling work for each processor. In addition, by modifying α, we can control the amount of allowed discrepancy in work between the processors. Note that choices of α that are *too* small, however, cause a large number of elements to be communicated, which affects performance in the next stage.

How much work takes place in the second stage? First, consider the cost of sending a large element. When communicating a large element, the many samples it covers are *not* being communicated. If our constant, α, is large enough, then the cost to communicate large elements will always be less than the cost to communicate its samples. Following this logic, the most communication the algorithm can undergo is when communicating only samples. Although this number can be quite large, the bound on the amount of data communicated is a nice property in the context of extremely massive data sets, like the 27 billion-element simulation we discuss in the performance section.

The only work done in the third stage is the sampling of large elements. In this stage, each processor is only responsible for a limited number of samples – those that are contained within its portion of the view frustum. This bounds how much sampling work can be performed. In addition, each element in this stage is "large" and covers many samples. So the number of elements (E) will be much less than the number of samples (S): $E < S/\alpha$. So, in this stage, the amount of work per processor is also bounded: $O(E+S) = O(S/\alpha + S) = O(S)$.

Summarizing, for all three stages, the amount of work per stage is bounded. In addition, most processors approach these bounds, leading to good parallel efficiency. Of course, degenerate cases exist where some processors will be at their theoretical bounds while other processors spin idly. In practice, however, the amount of work per processor is relatively even.

5. 3D Rasterization

The 3D Rasterization algorithm described in this section is an example of a sampling algorithm from section 3. It is similar in spirit to the 2D Rasterization algorithms employed by graphics hardware. The key difference, of course, is that the rasterization occurs in three dimensions (over a volume), rather than two (over an image). This technique is highly related to the method described in [YRL*96].

The rasterization algorithm simply operates on each element, one at a time, updating F as it goes. For each element E, the resampler works as follows:

1. Transform E's coordinates from world space to screen space
2. Calculate E's bounding box in screen space
3. If we are deferring large elements and the bounding box of E covers too many samples, add E to the output and continue to the next element
4. Determine the set of integer depths, { D_i }, that overlap between E and and the output grid G
5. For each D_i, slice E by D_i, resulting in polygons†
6. For each slice, employ a standard rasterization technique on the resulting polygons, updating the portion of the field F corresponding to D_i.

For clarity, consider the following example for a single element:

The first step is to transform the element to screen space (defined over Width, Height, and Depth axes). Assume the result is a tetrahedron, T, with points (8.5, 6.5, 4.2), (12.5, 6.5, 4.2), (10.5, 11.1, 4.2) and (10.5, 7.5, 6.8). The second step would be to calculate its bounding box: [8.5-12.5, 6.5-11.1, 4.2-6.8]. In the third step, we determine the bounding box that the element can contain no more than 40 ($4 \times 5 \times 2$) samples, so this element is not "large". Then we proceed to step 4 and determine the integer depths that overlap between T and G: 5 and 6 (see figure 2).

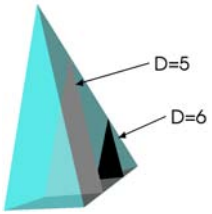

Figure 2: *Tetrahedron T, rendered transparently in cyan. Slice D=5 is gray, D=6 is black.*

We would then slice T by D=5 (see figure 3). This slice results in a triangle. We then employ a standard rasterization technique to the triangle. We would start by rasterizing along Heights 7, 8, and 9. For Height=7, we find samples at Width=10 and Width=11. Height=8 also yields samples at 10 and 11. Height=9 yields no samples. The entire slice at D=6 also yields no samples. In all, tetrahedron T yields 4 samples. These samples are then placed into F.

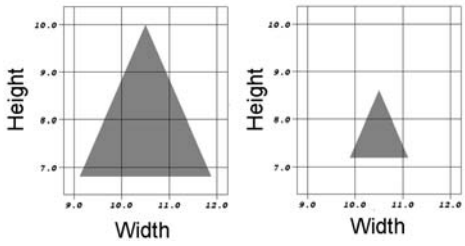

Figure 3: *Slice at D=5 (left) and D=6 (right)*

The 3D rasterization algorithm has many positive traits. First, for each element, it is possible to immediately determine which samples overlap. It is not necessary to trace rays to discover the list of cells along that ray. Second, unlike a standard projection scheme, the ordering of the resulting samples is preserved by the field F and the grid G. Third, this scheme is fairly general. It can accommodate structured grids, and curvilinear and unstructured meshes (but not point clouds). For unstructured meshes, it can accommodate the complete finite element zoo and virtually any additional element type. It can operate on multiple fields, allowing for multi-variate volume renderings. Fields can be either element-centered (i.e. piecewise constant) or node-centered. All of these options are incorporated into our implementation.

There are also negative aspects to the rasterization scheme. First, the technique for sampling is prone to missing data. If an element does not overlap with a sample point then that element is not reflected in the resampling. This is a common case for extremely small elements and more discussion can be found in [MWSC03]. We overcome this problem by having small elements locate the nearest sample point and attempt to affect its value. This can lead to two or more elements trying to affect the same sample. In this case, we use an arbitrator that chooses the "best" sample, where "best" is chosen to be the one with the highest opacity. The second problem with this scheme is less serious. The method of examining an element's bounding box to estimate the number of samples it contains can lead to overestimates. We do not attempt to correct this problem, because it does not lead to appreciable performance degradation.

6. Kernel-Based Sampling

Our Kernel-Based Sampling algorithm is designed to have each data point influence a region around it. This is a natural operation with element-centered variables. The data point is

† Curvilinear meshes can technically give curved polygons. We approximate these with linear polygons, similar to techniques used for isocontouring.

placed at the middle of an element and its region is guided by its bounding box. For nodal variables, each node's neighboring elements must be examined to determine the correct size for its sampling region. In our current implementation, we operate only on element-centered variables, and we re-center nodal variables.

The procedure to process an element E with variable value V is:

1. Transform E's coordinates from world space to screen space
2. Calculate E's bounding box in screen space
3. If we are deferring large elements and the bounding box of E covers too many samples, add E to the output and continue to the next element
4. Calculate a maximum radius of influence for E, $R_{max}(E)$
5. For every sample within $R_{max}(E)$ of E's center, update the field F with V and a weight ω

The size of the kernel, $R_{max}(E)$, is based on the size of the element, allowing larger elements to have greater impact than smaller elements. $R_{max}(E)$ is assigned to be fifty percent bigger than the distance from the center of the element's bounding box to one corner of the bounding box. However, for small elements, $R_{max}(E)$ can be so small that it will not affect any samples. To counteract this, we never let $R_{max}(e)$ drop below the global constant, R_{min}. R_{min} is chosen so that even the smallest element will affect at least one sample.

The weights, ω, vary based on the distance to the element center. Samples closest to the element center should strongly reflect the element's value. Moving away from the element center should allow for more blending with the neighboring element values. So we set the weight to be inversely proportional to the distance to the center of the element.

The formula for weight, ω, is:

$\omega = \frac{1}{(D_p+\varepsilon)}$ - ω_0, where

D_p = proportional distance to center of element

and ε and ω_0 are shape factors

For each sample, $D_p = \frac{D_s}{R_{max}}$, where D_s is the distance from the sample to the center of the element. The ε term eliminates division by zero, and determines the peak contribution an element can make, which occurs when the element center and a sample are coincident. When the distance from a sample to the element center is R_{max}, D_p is one. Since we would like the weight to be zero at the maximum radius, we introduce our second shape factor, ω_0, and set it to $\frac{1}{(1+\varepsilon)}$.

Once each element has updated the samples, we assign the final sample value to be the weighted average. Formally, if elements E_i update a specific sample with their values V_i and weights W_i, then the final value of the sample will be $\frac{\sum W_i * V_i}{\sum W_i}$. Also, note that this calculation does not require storing all encountered samples. Instead, running totals can be kept for $\sum W_i * V_i$ and $\sum W_i$, minimizing storage overhead.

Our $\sum W_i$ term also plays an important role in establishing the boundary of the data set. Each data point is sampled onto the region around it, regardless of the samples actual membership in the data set. Samples outside the boundary will have low $\sum W_i$ values. Reducing the opacity of these samples effectively creates the boundary in an anti-aliased way. Note that this is similar to the strategy applied in [PH89], although we do this correctly in the post-transformed space, where [PH89] did this in the pre-transformed space.

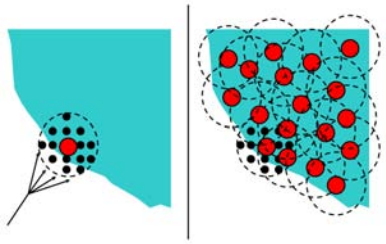

Figure 4: *On the left, an element's (red circle) sampling region (dotted circle) includes samples outside the data set boundary (marked with the four-way arrow). These samples will be updated with the element's value, but they will not affect the final picture because their $\sum W_i$ will be low. On the right, we see how each of the samples inside the data set are affected by many elements, giving higher $\sum W_i$, while those outside the data set are affected by fewer elements.*

The Kernel-Based Sampling algorithm also has many positive traits. First, like the 3D Rasterization algorithm, elements can be sampled independently and efficiently. The ordering of the resulting samples are preserved by using the field F and the grid G. Second, this scheme is also very general. In addition to handling all mesh-based data sets, this scheme is ideal for rendering point clouds. Third, unlike the 3D Rasterization algorithm, this technique consistently produces high-quality pictures and more accurately represents the entire input, including small elements.

There are also negative aspects to this scheme. First, element shapes are completely disregarded in the current implementation. Samples are updated based entirely on the element's R_{max}. In the common case for large scale data, however, an element is projected to only a few samples and this is inconsequential. Second, this algorithm processes the same sample multiple times, one for each of the nearby elements. This contrasts with the 3D rasterization algorithm, which only operates on a given sample one time. This results, of course, in higher running time.

7. Scalability

We now present the scalability of the algorithm. We performed the study on a 100 million element unstructured mesh and a 1024x1024 image with 1000 samples in depth. For each processor count, we measured the elapsed rendering time with multiple options. One option was the sampling

algorithm to use: 3D rasterization or Kernel-based. The other option was whether the camera was located inside or outside the data set. Table 1 shows the results.

Although we feel the algorithm has excellent scalability in this regime of processor count, we foresee some limitations as processor counts get larger. The time to create the final output image, including collecting portions from other processors and transferring the image from the server to the client for display, takes one tenth of a second. When more processors are used, this constant will begin to affect scalability. We do believe, however, that this algorithm will still be effective with larger processor counts, but as a weakly scalable algorithm. We believe the frame rate can never fall below some constant (minimally 0.1 seconds), but, given more and more processors, we feel that proportionally more data can be rendered in the same amount of time.

The timings were run on gauss, a 512-processor cluster of 2.4GHz Opterons connected by an InfiniBand network. Gauss is currently #409 on the Top500 listing of SuperComputers. For the timings, the algorithm was embedded in VisIt, rather than run as a stand-alone application. We disabled the mode that incorporates geometric primitives (for bounding boxes and other annotations), however, because delays that mode introduces are unrelated to our algorithm.

Procs	3D Rasterization/ outside	3D Rasterization/ inside	Kernel-Based/ outside	Kernel-Based/ inside
25	12.0s	21.9s	12.1s	63.7s
50	5.8s	12.1s	5.8s	30.5s
100	3.0s	7.0s	3.1s	15.3s
200	1.6s	3.6s	1.3s	7.6s
400	0.9s	2.1s	0.7s	4.1s

Table 1: *Our algorithm was strongly scalable in all of the configurations we tested.*

8. Results

8.1. Shock Propagation in Nano-Porous Metal

Classical Molecular Dynamics (MD) simulations are a common means to study material properties at the fundamental level of individual atoms, including such phenomena as plasticity, ductile failure, brittle failure, material response to ion or micro-meteorite impacts, laser ablation and more. The simulation we use for test purposes involved 1,013,455,626 (over a billion) atoms resulting from the study of shock wave propagation through a metallic foam. In this case, the shock was introduced in a high density region to the left of Figure 5, and has traveled partway through the porous solid, which was set up to gradually decrease in density as the shock travels to the right of the material sample. A typical variable of interest is the local energy potential of each atom, which can reveal dislocation and slip-plane structures that arise in response to the shock passage. The transfer function was chosen to reveal these structures (seen toward the left of the image as cross-hatch like patterns), as well as the unshocked, lower-density filament structure to the right (with the red outlines).

Given there is no mesh structure for MD simulations, the kernel-based resampling is the only choice for applying volume rendering. For solid mechanics problems like this, there is a fairly tight distribution of inter-atomic spacings (distance to nearest neighbor). Typically, it is appropriate to set the resample kernel radius to be a factor of 1.5-3 times the average inter-atomic spacing, depending on whether views of fine void structures or smoother averaging is desired for the application analysis.

We visualized this data set using 256 processors. For smaller images (400x400), each rendering takes about 8 seconds. For large images (1024x1024), it takes 30 seconds.

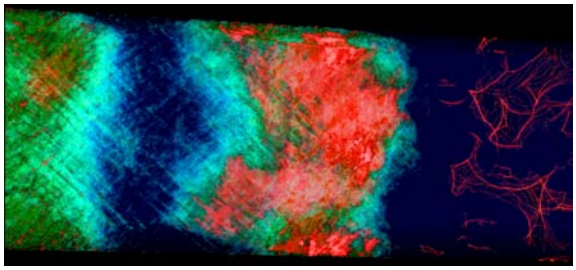

Figure 5: *A volume rendering of over a billion atoms, rendered using the kernel-based resampling method. The simulation was produced by Farid Abraham of LLNL on 8000 processors of the ASC Purple supercomputer. A shock is traveling from left to right through the metallic foam, which was designed to gradually change from high to low density during the shock front propagation. The cross-hatch patterns on the left show the emergence of plasticity due to dislocations, while the undisturbed filament structure is highlighted on the right.*

8.2. Rayleigh-Taylor Instability

We next applied our algorithm to a 27 billion element rectilinear grid simulation of a Rayleigh-Taylor Instability, where heavy and light fluids mix. This calculation was done on the BG/L Supercomputer by the MIRANDA code. We used 256 processors[‡] and each 1Kx1K image took 3.8 seconds to render (see figure 6). This data set highlights one of the problems with our 3D rasterization scheme. When using one thousand samples per ray, the 3D rasterization scheme cannot represent all of the data, since there are so many more elements than sample points.

[‡] Machine unavailability forced us to use this smaller number of processors.

For rectilinear grids, our 3D rasterization scheme has been optimized to efficiently find elements that overlap with samples. The purpose for transforming individual elements from world space to screen space is to quickly identify the samples an elements overlaps with. For rectilinear grids, this technique is not necessary, as the location of a sample point can be directly and efficiently calculated. As such, we revert to a simplified sampling scheme for rectilinear grids that does not transform individual elements to screen space. With this optimization, voxels that do not overlap with samples are ignored. In effect, this makes it a scheme that is indifferent to data set size. For any size rectilinear grid, the exact same amount of work is performed. When rendering a 1 billion element rectilinear grid, with the same number of processors, the rendering rate improved by a factor of 2.5. We attribute this unexpected speedup to paging through memory.

When we switched to the kernel-based sampling scheme, performance dropped considerably because every element had to be traversed. In this mode, it took 45 seconds to generate a picture. Because of the nature of the data set, (smooth features that span many elements), differences between the two pictures were not significant.

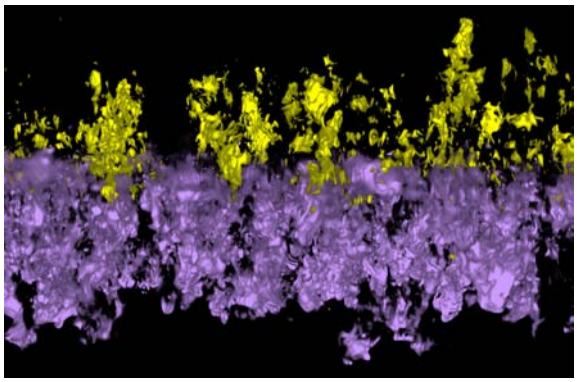

Figure 6: *The mixing layer between heavy and light fluids. We are rendering based on vertical velocity. Yellow volumes are moving up, purple volumes are moving down.*

8.3. Blast Calculation

Our next calculation simulated an explosive blast wave as it passed over a section of a wall consisting of reinforced concrete. It was calculated using the ALE3D simulation code with Arbitrary Lagrange-Eulerian (ALE) methods on a 3D unstructured hexahedral mesh. For this study, we subdivided each hexahedron into five tetrahedrons, for a total of 138 million, to produce an unstructured mesh with an element count that would stress our algorithm. Because some surfaces were so thin, even the kernel-based sampler had problems properly reflecting the data. The averaging of values within a sample's region led to smearing. We increased the samples per ray to 2000, which obviously led to a degradation in performance. We used only 128 processors and were able to render a 1Kx1K frame every 35 seconds (see figure 7). Considering we were operating on a larger data set and had twice as many samples, this is consistent with the scaling study we performed earlier.

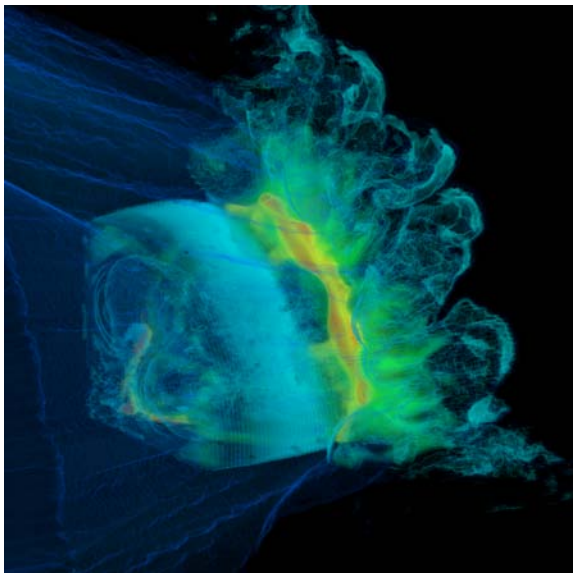

Figure 7: *An explosive driven blast wave passing over a round section of reinforced concrete (1/4 symmetry)*

9. Future Work

There are many performance improvements that can be made to our algorithm. We did not partially composite samples before sending them, because a sample's value with the kernel-based technique can not be determined until all of the surrounding elements have made their contribution. We believe that, despite this issue, we can proceed with this optimization on a restricted basis by determining situations where samples have received their full contribution and allowing those samples to be partially composited. Another option would be to apply compression algorithms on the samples themselves before communication. In addition, we could further reduce communication by changing the way we partition the image volume. We currently partition over the pixels, which leads to long shafts in depth. An alternative would be to create cubes by dividing the volume in depth as well. This would increase the volume to surface area ratio, minimizing the number of times a large element is mapped to multiple processors. We have experimented with implementations like this, but decided not to present this scheme, because the extra steps to composite the samples would further complicate the overall algorithm. Finally, we have considered accelerating the sampling phases by using 3D graphics

hardware. Of course, this approach would only be viable on clusters where graphics hardware is available.

There are also opportunities for improvements in picture quality. We currently have no lighting model, and we would benefit from incorporating pre-integration techniques.

10. Conclusions

We have presented an algorithm that allows massive data sets to be volume rendered at interactive speeds, given adequate computing resources. The algorithm is sample-based, and the overhead for processing the samples is high. As a result, this algorithm would be a poor choice for volume rendering small data sets with low compute power, because a disproportionate amount of time is spent calculating the values of the samples. But, again, the algorithm is ideal for massive data sets and we have demonstrated good performance on some of the largest data sets ever simulated.

We caution the reader against characterizing this algorithm as a brute-force algorithm applied on a large machine to achieve a result. When scaling up to large numbers of processors, the difficulty comes in maintaining good parallel efficiency. We are able to do this elegantly with our hybrid approach. By splitting processing into small and large element sampling phases, the work performed in each phase cannot exceed known bounds. These bounds guarantee that no processor is tasked with a disproportionate amount of work to perform while other processors spin idly. This hybrid algorithm is the principal contribution of this paper. In addition, we have introduced some sampling schemes that are well suited for this general approach and allow for either quick or accurate sampling.

11. Acknowledgments

VisIt has been developed by B-Division of Lawrence Livermore National Laboratory and the Advanced Simulation and Computing Program (ASC). This work was performed under the auspices of the U.S Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. Lawrence Livermore National Laboratory, P.O. Box 808, L-159, Livermore, Ca, 94551 In addition, this work has been sponsored in part by the U.S. National Science Foundation under contracts CCF 0301194 (PECASE), CCF 0222991, OCI 0325934 (ITR), IIS 0552334 and the U.S. Department of Energy under Memorandum Agreement No. DE-FC02-01ER41202 (SciDAC) and DE-FG02-05ER54817 (SciDAC).

References

[CBB*] CHILDS H., BRUGGER E., BONNELL K., MEREDITH J., MILLER M., WHITLOCK B., MAX N.: A contract based system for large data visualization. In *Proc. of Visualization 2005 Conference*. 1

[CMF] CAVIN X., MION C., FILBOIS A.: Cots cluster-based sort-last rendering: Performance evaluation and pipelined implementation. In *Proc. of Visualization 2005 Conference*. 2

[CMSW04] COOK R., MAX N., SILVA C., WILLIAMS P.: Image-space visibility ordering for cell project volume rendering of unstructured data. *IEEE Transactions on Visualization and Computer Graphics 10*, 6 (Nov. 2004), 695–707. 2

[DPH*03] DEMARLE D., PARKER S., HARTNER M., GRIBBLE C., HANSEN C.: Distributed interactive ray tracing for large volume visualization. In *In Proceedings of the Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 87–94. 3

[GHJA] GAO J., HUANG J., JOHNSON C. R., ATCHLEY S.: Distributed data management for large volume visualization. In *Proc. of Visualization 2005 Conference*. 2

[HE03] HOPF M., ERTL T.: Hierarchical splatting of scattered data. In *Proc. of Visualization 2003 Conference* (2003), pp. 443–440. 2

[LM] LUM E., MA K.-L.: Hardware-accelerated parallel non-photorealistic volume rendering. In *Proc. NPAR 2002*. 2

[MC97] MA K.-L., CROCKETT T.: A scalable parallel cell-projection volume rendering algorithm for 3d unstructured data. In *Proc. 1997 Symposium on Parallel Rendering* (1997), pp. 95–104. 2

[MWSC03] MAX N., WILLIAMS P., SILVA C., COOK R.: Volume rendering for curvilinear and unstructured grids. In *Computer Graphics International* (2003), p. 210. 5

[PH89] PERLEN K., HOFFERT E.: Hypertexture. *Computer Graphics 23*, 3 (July 1989), 253–261. 6

[SFLS00] SAMANTA R., FUNKHOUSER T., LI K., SINGH J. P.: Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware* (Aug. 2000), pp. 97–108. 3

[SMW*] STRENGERT M., MAGALLON M., WEISKOPF D., GUTHE S., ERTL T.: Hierarchical visualization and compression of large volume datasets using gpu clusters. In *Proc. Eurographics/ACM SIGGRAPH Parallel Graphics and Visualization 2004*. 2

[WGS] WANG C., GAO J., SHEN H.-W.: Parallel multiresolution volume rendering of large data sets with error-guided load balancing. In *Proc. Eurographics/ACM SIGGRAPH Parallel Graphics and Visualization 2004*. 2

[YRL*96] YAGEL R., REED D. M., LAW A., SHIH P., SHAREEF N.: Hardware assisted volume rendering of unstructured grids by incremental slicing. In *Proceedings 1996 Symposium on Volume Visualization* (1996), pp. 55–62. 5