Eurographics Symposium on Parallel Graphics and Visualization (2012)
H. Childs, T. Kuhlen, and F. Marton (Editors)

A Study of Ray Tracing Large-scale Scientific Data in Parallel Visualization Applications

Carson Brownlee[1,2], John Patchett[3], Li-Ta Lo[3], David DeMarle[4], Christopher Mitchell[3], James Ahrens[3], and Charles D. Hansen[1,2]

[1]School of Computing, University of Utah
[2]SCI Institute, University of Utah
[3]Los Alamos National Labs
[4]Kitware Inc.

Abstract

Large-scale analysis and visualization is becoming increasingly important as supercomputers and their simulations produce larger and larger data. These large data sizes are pushing the limits of traditional rendering algorithms and tools thus motivating a study exploring these limits and their possible resolutions through alternative rendering algorithms . In order to better understand real-world performance with large data, this paper presents a detailed timing study on a large cluster with the widely used visualization tools ParaView and VisIt. The software ray tracer Manta was integrated into these programs in order to show that improved performance could be attained with software ray tracing on a distributed memory, GPU enabled, parallel visualization resource. Using the Texas Advanced Computing Center's Longhorn cluster which has multi-core CPUs and GPUs with large-scale polygonal data, we find multi-core CPU ray tracing to be significantly faster than both software rasterization and hardware-accelerated rasterization in existing scientific visualization tools with large data.

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Graphics Systems—Distributed/network graphics

1. Introduction

The push to produce faster supercomputers is producing ever-larger data sets, challenging traditional methods of analysis and visualization. Rendering is a key difficulty in managing the visualization pipeline for large-scale data sets. Specialized visualization resources are frequently collocated with supercomputers to enable data analysis and visualization of the data sets produced on the supercomputer. These specialized visualization resources are typically heterogeneous supercomputers containing CPUs and GPUs, which are added to support rendering. Aside from the GPUs, the analysis machines are typically similar, though smaller than the supercomputers they support. Understanding the limits of these specialized resources' rendering capability of large-scale data sets is important in planning support of existing and future supercomputers. We present a study of rendering large-scale scientific data in widely used parallel visualization applications.

A standard visualization workflow for extremely large data involves an ordered set of activities: application simulation, visualization algorithms, rendering, and display as shown in Figure 1. Each of the stages produces output whose size can vary but is generally related to the size of the input, except for the rendered image, which is fixed based on the requested image size. Each portion of the rendering and simulation process is often distributed across hardware that is suited to each task. A simulation typically writes a quan-

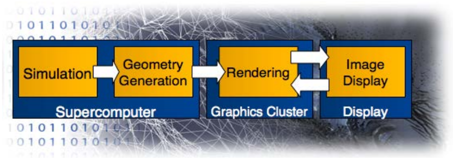


Figure 1: *A remote visualization workflow with a separate compute cluster and rendering cluster.*

DOI: 10.2312/EGPGV/EGPGV12/051-060

tity of data to disk that is proportional to the size of the run. This data is then read from disk to produce geometry, which is then rendered to an image or saved back to disk for subsequent rendering.

We believe a user's ability to interact with data is of great importance to insight and that 5-10 frames per second is the minimum speed necessary for a positive interactive experience. This work will benefit the community by providing better understanding of real-world performance of two of today's popular parallel visualization tools with large polygonal data on modern cluster hardware.

To understand trade offs for dealing with extremely large datasets, we have employed an in-depth timing study, using both strong and weak scaling across 128 cluster nodes, of three different rendering algorithms for large polygonal data: software based ray tracing (integrated into ParaView and VisIt) and ParaView and VisIt's two default rendering methods: software-based rasterization and hardware-accelerated rasterization. We use three different data sets: two scientific and one synthetic. In addition to the scaling study, another contribution of this work is the integration of the Manta software ray tracer developed at the University of Utah in ParaView and VisIt, which allows ray tracing to be seamlessly used in popular visualization tools.

In the next section, we review related work. We then describe three commonly used rendering methods in Section 3. Section 4 describes visualization of large-scale data with ParaView and VisIt and our custom Manta integration in VTK. We then provide results comparing rendering with the three commonly used methods including weak and strong scaling studies in Section 5. Lastly, we conclude and describe future directions in Section 6.

2. Related Work

There have been many strategies developed to visualize large-scale data. Transferring data to a GPU cluster for rendering is a well developed practice that displays large datasets at very high frame rates by distributing data for rendering [FQK08] and compositing the resulting images together using algorithms such as the Binary-Swap method [lMPH94]. Common visualization tools that run on GPU clusters include such programs as ParaView [CGM$^+$06, Kit10] and VisIt [LLN10] which present client/server parallel rendering frameworks built on top of the Visualization Toolkit (VTK). These real-world applications often do not have the latest compositing or rendering algorithms implemented nor do they have application specific optimizations. As node counts are scaled up, a larger percentage of time is devoted to disk and network I/O than rendering a single image as shown by Childs et al. [CPA$^+$10], however when interacting and exploring data users may be rendering hundreds or thousands of frames and their rendering times for viewing isosurfaces were non-interactive. Our end goal then was to utilize a rendering method which can achieve interactive rendering rates with software rendering on both rendering and compute clusters.

Massive polygon rendering presents challenges for traditional rasterization methods such as those used in VTK. OpenGL relies on hidden surface removal with a simple Z-buffer test to determine visibility. This not only requires transforming vertex coordinates but also unnecessarily shading fragments rendered back-to-front along the camera axis. When dealing with millions of triangles, many of which are likely obscured behind other triangles, these unnecessary transforms and shading operations degrade performance resulting in a linear or greater decrease in speed in relation to the number of triangles. With GPU rendering there are two methods of gaining better scaling performance: occlusion techniques and surface approximations. Occlusion methods include spatial subdivision schemes, which are used for occlusion culling hidden triangles that lie behind other triangles or are not in the view-frustum. Implementations of occlusion culling methods include the hierarchical Z-buffer [GKM93], the hierarchical occlusion map [ZMHH97] which optimize performance through hierarchical Z-Pyramids, and GPU optimized occlusion queries [BWP04]. Model simplification becomes essential for GPU rasterization to scale with large geometry counts. One method for looking at large datasets remotely is to use levels of detail (LOD) techniques first proposed by Clark et al. [Cla76] [Lue01]. These techniques are covered in great detail by Yoon et al. [SeY08]. Due to the complexity of these methods, ParaView and VisIt use brute-force polygon rendering when visualizing data, which simply sends all triangles to the GPU. Optionally, these programs support using a single display list to render each visible VTK actor's geometry and a lower quality mesh during user interaction. In practice, however, a single display list can crash when compiling too many calls and often fails provide faster render times than immediate-mode rendering for larger polygon counts in our tests.

In situ visualization is becoming more important for generating renderings of large data on compute clusters [YWG$^+$10]. One such example would be using ParaView or VisIt paired with a software implementation of Mesa which can be enabled through build options and is a commonly used software rasterizer with these tools. This method proves to be too slow for interactive rendering with large polygon counts in most cases we have tested with. Mitra and Chiueh [MC98] implemented a parallel version of Mesa by running multiple processes and compositing results together on a single machine. Nouanesengsy et al. [NAWS11] showed that scaling could be achieved over multiple threads by using a hybrid sort-first and sort-last compositing step. However, using multiple ParaView processes failed to scale well in their tests and for our work we focus on performance in existing real-world programs. Therefore, we focus our tests on performance running a single program instance, and use multi-threaded hybrid parallelism to achieve scaling in our ray tracing implementation.

Ray tracing on clusters for visualizing large-scale datasets is a well developed field with several benefits over traditional rasterization methods without advanced techniques. Occlusion culling is gained implicitly through acceleration structures. Sub-pixel geometry is also sub-sampled in screen-space negating the necessity for geometry simplification. Ray tracing performance has been shown to scale very linearly from one to hundreds of processors [Par02]. Tracing rays also scales very well with the amount of geometry in the scene due to the logarithmic acceleration structures used [MP01, WSB01], for which we use a packet based Bounding Volume Hierarchy, BVH, [WBS07]. Advanced cluster based ray tracing methods can split up data and rays by image-space or data-space. The former relies on rays in spatially coherent portions of image space requesting the same data as their neighbors. When a node needs a new part of the scene, data is paged in. In highly efficient implementations, the same page faults used by the operating system can be remapped to network requests instead of disk reads [DGP04]. Ize et al. [IBH11] expanded upon this work by creating a version of the Manta ray tracer which can run on distributed memory systems by paging in and out cache-aligned BVH nodes in a ray-parallel fashion. Ray parallelism is efficient for out-of-core rendering but not possible within VisIt and ParaView's sort-last distribution and compositing, which distributes data independent of the view. This limitation can lead to sub-optimal work distribution with respect to the view which limits rendering performance, however we have found that this can still provide interactive frame rates for many of our tests while efficiently distributing data analysis and loading.

While ray tracing on the CPU provides for expanded use of large memory spaces available to the CPU memory, access times have decreased much slower than processing time [GKY08] and so methods to reduce memory requirements are still vital to software ray tracing. Marsalek et al. recently implemented a ray tracing engine into a molecular rendering tool for advanced shading effects for the purpose of producing publication images [MDG$^+$10]. We have integrated a ray tracing engine into general purpose visualization tools that can use advanced rendering effects, however we focus on ray tracing's logarithmic performance with large data rendering across multiple distributed-memory nodes as the primary motivation for this study. The Manta real-time ray tracing software has shown to scale very well on large shared-memory machines [SBB$^+$06] and Ize et al.'s work proved that real-time rendering speeds could be achieved with distributed-memory systems but is only a rendering engine without data analysis capabilities. Thus, combining CPU rendering using Manta is ideal when combined with other cluster based visualization tools such as ParaView and VisIt, which handle geometry creation, analysis, data distribution, and image compositing within tools which users are already familiar with.

3. Rendering Methods

Our approach to comparing visualization methods on large distributed systems was to evaluate three rendering techniques within commonly used visualization applications: hardware-accelerated rasterization, software-based rasterization, and software-based ray tracing. Each method has its advantages and drawbacks.

Hardware-accelerated rasterization has proven to be fast for modest data sizes and is widely used and heavily supported. The disadvantages are the requirement for additional hardware, small memory sizes on the GPU, and, due to the nature of rasterization, rendering times that scale linearly with the amount of geometry in the scene. Advanced rasterization techniques such as HLOD methods are not currently implemented in widely used visualization tools such as ParaView or VisIt. Therefore, we do not consider them in our study. ParaView does support a single level of detail which can be toggled when the user interacts with the camera, however this degrades the quality of the entire model and thus is not considered for our tests.

Software rasterization through Mesa is a build option for both ParaView and VisIt and is a common method used on supercomputers when GPU hardware is not available. It offers the same support for programs that would normally use hardware-acceleration methods. The main drawback of this method is speed, as Mesa remains single threaded and delivers very slow performance even for low geometry counts. A benefit over hardware-accelerated rasterization, however, is that it does not require additional graphics hardware and can utilize large system (CPU) memory.

Software ray tracing provides a rendering method that scales in $k * O(log(n))$ where k is image size and n is the number of polygons. This scaling performance assumes non-overlapping polygons and a well-balanced acceleration structure. Because of the screen space dependent performance with logarithmic scaling to geometry, ray tracing provides efficient performance which scales well with increasingly large geometry counts, especially for sub-pixel geometry. Using an acceleration structure to test ray intersections also allows easy and straightforward visibility tests where only the nearest geometry needs to be shaded once for each pixel. Hardware ray tracing also exists, but we chose to focus only on software ray tracing. We have implemented ray tracing as a rendering mode for polygon data within ParaView and VisIt.

4. Ray Tracing Implementation

ParaView and VisIt are open-source visualization frameworks designed for local and remote visualization of a large

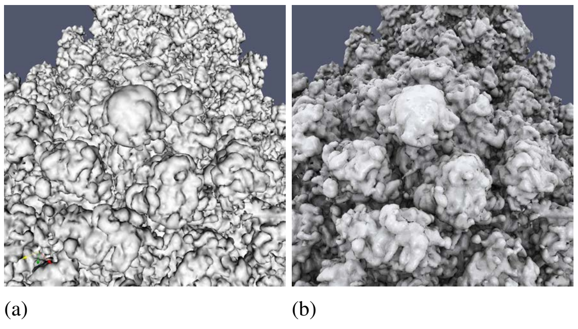


Figure 2: *Rendering of the RM dataset with OpenGL on the left and Manta on the right using ambient occlusion and reflections.*

variety of datasets. They are designed to run on architectures from a single PC desktop up to large cluster arrays using client/server separation and parallel processing. VisIt operates with a client/server architecture to run multiple servers for data analysis and rendering on large clusters. Server/data separation allows ParaView to be broken into three main components: data servers, render servers, and a client [CGM+06]. This separation allows for varying numbers of servers to be used for data or rendering depending on the need. Much of the actual rendering code is based around the Visualization Toolkit (VTK) while much of the client/server model is unique to ParaView and VisIt. This common base allows us to to implement Manta once in VTK and integrate the VTK ray tracer into ParaView and VisIt with only minor modifications specific to each program.

4.1. Data Distribution

ParaView and VisIt both utilize client/server model's for running on large, distributed-memory systems. VisIt launches a viewer client and multiple servers which are used for data loading and rendering. ParaView's data server abstraction layer allows for operations such as processing data on one node and sending the resulting geometry to another node or multiple nodes for rendering. This allows for changing the data processing and rendering pipeline across heterogeneous architectures for balanced workload distribution when more or fewer rendering servers are needed than data servers. When rendering on multiple nodes, sort-last compositing is required to combine images from multiple nodes. Data-parallel data distribution is good for rasterization, but not necessarily optimal for ray tracing where render time is dependent more on work distributed over the viewing frustum. When zoomed out over an entire model, distributed cluster-based ray-tracing often produces a sufficiently balanced workload distribution, however, if a small portion of the data is taking up the majority of screen space then the majority of work is being done by a limited number of nodes which contain data within the viewing frustum. However, we have found it to be usable in practice as shown in Section 5. Distributing the compositing work across a cluster is vital for efficient cluster utilization. For this we use a Binary-Swap [lMPH94] implementation and use the IceT compositing library with default settings, which can be enabled for both VisIt and ParaView. Binary-Swap is a parallel compositing algorithm that exchanges portions of images between processes to distribute the workload. Because Binary-Swap composites the entire image, empty portions of the scene are needlessly composited together. IceT allows for variably sized compositing windows encapsulating only portions of the scene which contain rendered geometry. This has the potential to vastly decrease the amount of image data sent across the network for our applications as more nodes are used for rendering.

4.2. Synchronization

Manta was originally designed for large shared memory systems. To achieve the highest possible frame rate possible, multiple rendering threads are launched on a single process and the renderer sequentially renders frames as fast as possible. VTK was designed for event driven rendering where one frame is rendered after user interaction. The threaded nature of Manta also presented a thread safety issue: Manta's state can only be accessed and modified at certain points in its pipeline through callbacks called transactions. In order to safely halt Manta between user interactions, synchronization was added through semaphores in the display phase. While each render thread renders the scene, thread zero displays the previously rendered frame and then continues ray tracing in order to utilize all threads. At the end of a rendering step, the threads are synchronized and state is updated. This is where state can safely be accessed and modified outside of Manta through transactions. The default Manta behavior results in a one frame delay between rendering and display of the current scene because of the display synchronization.

The rendering architecture of Manta was modified slightly to have the image display outside of the rendering stage as a separate synchronization step, which is only released upon a render event in VTK. This eliminates unnecessary rendering before or after a render event. A rendering call triggers Manta to release its rendering lock in the display phase, process transactions, and render a frame, which is then copied back to the vtkRenderWindow instance. Due to the differences in how an image is stored between VTK and Manta, this requires an image conversion step. ParaView and VisIt then display the rendered image or send the image out for compositing if it is being rendered remotely through a cluster.

4.3. Depth Buffer

Sort-last rendering requires a depth buffer in order to determine an ordering for overlapping geometry in the compositing step. In order to work inside VTK, Manta required a depth buffer to be implemented, as ray tracing typically does not require one. Depth values, or rather closest hit values, are typically kept per ray, which meant that for our implementation all that was needed was a separate buffer and a simple write operation for each ray. This results in additional memory overhead with one float per image pixel for the buffer.

4.4. Acceleration Structures

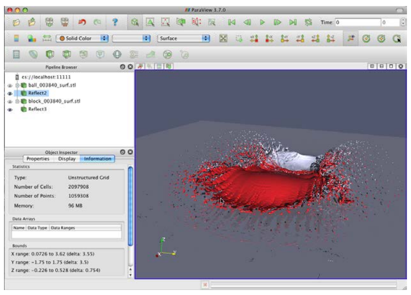
Ray tracing uses acceleration structures to compute hit points efficiently. This means that a new acceleration structure needs to be built with each change in geometry within VTK. Generating a new acceleration structure each time a new vtkActor is added or updated in the pipeline with a custom vtkActor override facilitates this. For very large scenes consisting of millions of triangles this can take several seconds of pre-computation time. The amount of time also depends on the acceleration structure used. Grid based acceleration structures can be faster to update, however, we chose to use a Bounding Volume Hierarchy, BVH, as it gave similar performance to a kd-tree while benefiting from faster build times [Ize09].

4.5. Color Mapping

In Manta, assigning a per-vertex color would require creating a material for each triangle which would entail a large memory overhead. There was also no support for per-vertex materials thus we chose to implement colors through a 1D colormap. In Manta, texture coordinates are weighted by the barycentric coordinates of the interior point as a weight. This provides a smooth coloration and little overhead since texture coordinates are computed for each intersection point even for constant colors. Both VisIt and ParaView support texture colormaps which are implemented through the same texture colormap used for singular color values in Manta.

4.6. VTK Factory Overrides

The Manta context is created in VTK through a custom factory object which overrides many common VTK objects. A vtkRenderWindow overrides other windows to keep track of updates to image size. A vtkPolyDataMapper keeps track of updates to geometry which then sends a group of polygons, usually a triangular mesh, to the active vtkActor. vtkActors are overloaded to track material properties and to maintain acceleration structures needed for rendering. When an actor is updated, so is its corresponding acceleration structure. Unfortunately, because of the differences in how meshes are represented between Manta and VTK, there is currently additional memory overhead due to geometry duplication. Lights are overloaded with custom vtkLight objects which are stored as either directional or point lights. Ambient occlusion is treated as an ambient light source in Manta, and so it is added to the actor through custom options specific



cs://localhost:11111
ball_003840_surf.stl
Reflect2
block_003840_surf.stl
Reflect3

Statistics
Type: Unstructured Grid
Number of Cells: 2097908
Number of Points: 1059308
Memory: 96 MB

Data Arrays
Name | Data Type | Data Ranges

Bounds
X range: 0.0726 to 3.62 (delta: 3.55)
Y range: -1.75 to 1.75 (delta: 3.5)
Z range: -0.226 to 0.528 (delta: 0.754)

Figure 3: *Manta running in ParaView on a single multi-core node using shadows and reflections rendering the impact of an aluminum ball on an aluminum plate.*

to the application. Finally, a custom vtkRenderer synchronizes the Manta display with VTK render calls and sends image data to the active window for compositing or display. This integration allows for Manta to behave identically to the OpenGL render engine within VTK. Figure 2 shows an isosurface of the Richtmyer-Meshkov Instability rendered with OpenGL on the left and Manta on the right using ambient occlusion using the same material and camera information provided by VTK.

4.7. ParaView and VisIt

VisIt and ParaView are both built on top of the VTK framework which allowed our VTK implementation to integrate into their rendering pipelines with little modification. Activating factory overrides to use the Manta ray tracer is handled through ParaView's plugin interface or VisIt's commandline interface. These mechanisms allow for easily activating or deactivating the Manta renderer, however both implementations require closing other rendering windows before launching Manta. A Manta IceT rendering pass was created for ParaView which sends image buffers directly to IceT for faster compositing. This modification has not yet been transfered over to VisIt's implementation and instead VisIt sends image data to the active render window which currently is not a custom Manta version for VisIt which can limit performance in some cases. A custom implementation of some annotation objects, such as axes, were overridden to prevent them from sending significant amounts of geometry every frame, however these could be reimplemented to maintain their geometry from frame to frame. In ParaView, most polygonal rendering modes are supported. Volume rendering is not currently supported in manta-enabled VisIt or ParaView. While implementing Manta-based volume rendering for VisIt and ParaView is highly desired and implementations for this already exist within Manta, a VTK integration has not been developed as of this writing.

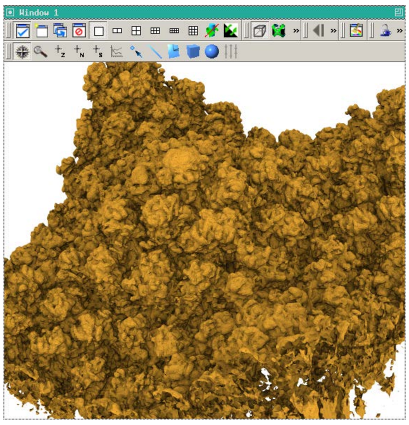


Figure 4: *Manta rendering a version of the RM dataset within VisIt. Ambient occlusion provides additional insight to users by adding occlusion information from nearby geometry.*

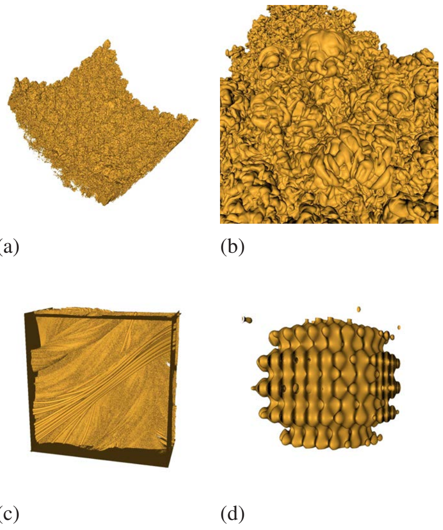


Figure 5: *The datasets used for benchmarking. RMI isosurface zoomed out (a), RMI isosurface closeup (b), VPIC dataset with isosurface and streamlines (c) and a wavelet contour with 16 million triangles (d).*

4.8. Advanced Rendering

Ray tracing allows for additional scene and material properties from those previously found in VisIt and ParaView. These additional options include such materials as dielectrics or transparencies as well as multisampling and threading options to specify the number of threads the ray tracer will launch for rendering. The result of such rendering options is shown in Figures 2, 3, and 4. Figure 3 displays an aluminum ball hitting an aluminum plate with shadows and reflections. Figure 4 shows an isosurface of the RM dataset rendered with Manta within VisIt. Ambient occlusion adds additional insight to the viewer by shading by occlusion from neighboring polygons, pulling out features at the surface of the isosurface. Advanced shading effects such as shadows, ambient occlusion and reflections can only be rendered when running on a shared memory system. VTK has no way to handle fetching distributed data for such information in the middle of a rendering. Ambient occlusion will work with local data, however due to data distribution, regions bordering blocks of data will be missing occlusion information. This could be resolved by duplicating borders of data blocks for each node and only considering polygons a small distance from ray hit points. This would require rewriting the data distribution within ParaView and was not implemented for this paper. In order to expose these options to the user, GUI elements were added to ParaView. These options are expected to also be included with VisIt in a future release.

5. Results

We evaluated the rendering performance of various methods on Longhorn, an NSF XD visualization and data analysis cluster located at the Texas Advanced Computing Center (TACC). Longhorn has 256 4X QDR InfiniBand connected nodes, each with 2 Intel Nehalem quad core CPUs (model E5540) at 2.53 GHz and between 48-144 GB of RAM. Each node of Longhorn also has 2 NVidia FX 5800 GPUs. We used 3 datasets of varying sizes: a synthetic wavelet dataset, and a dataset from Los Alamos's plasma simulation code VPIC, and a time step from a Richtmyer-Meshkov Instability simulation rendered with two different views. We used ParaView 3.11 and VisIt 2.4 for all timings with three different rendering modes: Manta, an open-source ray tracer, Mesa, a software OpenGL rasterizer, and hardware-accelerated OpenGL. ParaView and VisIt were built with the IceT library and use Mvapich2 1.4. Additional code for timing was added for ParaView and VisIt. Mesa is not multi-threaded nor the fastest available software rasterization package however it is the only one supported internally within ParaView and is commonly used when GPUs are not available. The OpenGL implementation within VTK is also a brute-force implementation with no advanced acceleration methods used. This study is therefore not a comparison of the potential of these algorithms but rather a real-world study of their existing performance within common visualization tools. To test the scaling of these packages we ran a series of weak and strong scaling studies up to 128 nodes on Longhorn.

Datasets

- **Richtmyer-Meshkov Instability** We created a polygonal model from an isosurface of time-step 273 of the Richtmyer-Meshkov Instability (RM) simulation, resulting in 316 million triangles. To better understand the be-

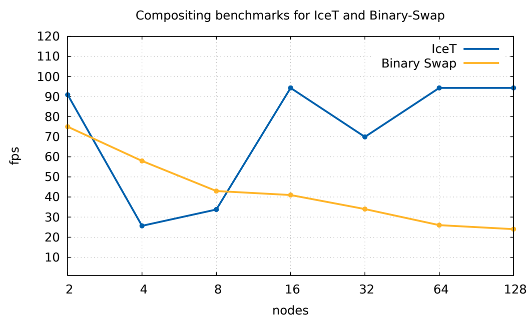


Figure 6: *Frames per second from compositing using the binary-swap and IceT reduce compositors for a* 1024^2 *image from 2 to 128 nodes.*

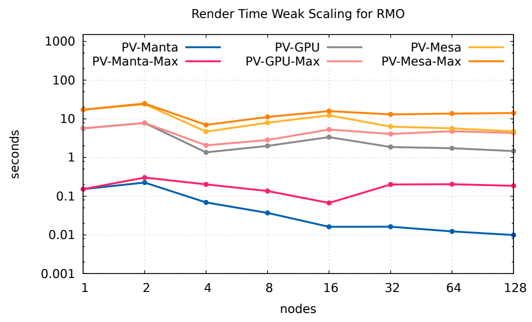


Figure 7: *Weak scaling timings of an isosurface of the RMO dataset in ParaView. Geometry is added with each additional node so that the geometry per node remains roughly constant.*

havior of the data distribution in ParaView, we have rendered both a zoomed out view of the RM dataset, RMO, seen in Figure 5(a) and a closeup view, RMI, in Figure 5(b). The closeup view should expose the behavior of sort-last rendering when the data being rendered on screen belongs to only a small portion of the overall nodes.

- **VPIC Visualization** Using a singe time-step from the VPIC plasma simulation, we calculated an isosurface and extracted streamtubes that combined totaled 102 million polygons. A view of this data set can be seen in Figure 5(c).
- **Wavelet** The wavelet triangle dataset is a computed synthetic dataset source released with ParaView. We generated a 201^3 dataset and then calculated as many isosurfaces as needed to produce a certain quantity of triangles. The isosurfaces are nested within each other. Images produced with 16 million triangles are shown in Figure 5(d).

5.1. Cluster Timings

To test the performance of the three rendering modes with large-scale data we conducted a series of timing studies on the rendering cluster Longhorn that look at how performance behaves with varying numbers of cluster nodes using a single process per node of ParaView and VisIt. A window size of 1024^2 was used with offscreen rendering for both ParaView and VisIt. We investigate two types of performance: strong and weak scaling. A strong scaling study keeps the problem size constant while increasing resources to solve the problem. As the number of nodes increases, the data being loaded and rendered per node decreases. A weak scaling study keeps the problem size constant per node as the number of nodes increase. The total frame time is dominated by a combination of the rendering time and the compositing time where $t_{total} = t_{composite} + t_{render}$. An alternative method would be to have a frame delay, where the last frame is displayed and thus the total time would be $t_{total} = max(t_{composite}, t_{render})$. However, this would require a reworking of the VTK pipeline in order to be able to push rendered frames after user interaction was finished and was not implemented for this study.

In order to determine bottlenecks from compositing, Figure 6 displays total frames per second using Binary-swap running in ParaView with an empty scene to show performance where the entire image from each node must be composited. With Binary-swap, pixels generated by rendering empty portions of the scene are needlessly sent over the network. To show a more practical example, frames per second from the VPIC dataset using IceT are presented. The IceT frames per second were calculated as the inverse of the reported composite and render times for the client node on processor zero subtracted from the maximum rendering time across all nodes. From these runs we can see that on the InfiniBand cluster used in our testing, the worst performance we could expect with Binary-swap is over 20 fps on 128 nodes, however in some cases IceT achieves nearly 100 fps by only compositing small portions of the image. Our goal is then to achieve rendering speeds which can match or exceed the maximum Binary-swap compositing times.

5.1.1. Weak Scaling

In a weak scaling study the problem size scales with the processor resources. For our testing, we conducted a weak scaling benchmark studying rendering times for a scientific dataset. Figure 7 shows a weak scaling study conducted using the RMO dataset and scaled up by adding additional isosurfaces to increase the number of polygons as more nodes were added. In total, the polygon count ranged from around 49 million triangles for a single node up to over 1.6 billion for 128 nodes. Hardware and software-based OpenGL times remain above one second while ray tracing times decrease with more nodes. This shows that the rendering time of the brute-force rasterization algorithm is fairly constant with geometry count regardless of the distribution of that data in the scene. Our ray tracing implementation, however, ignores occluded geometry and is bounded by $k * log(n)$. Empty space is easily skipped over by tracing rays in packets and testing the bound box of the scene. With weak scaling, geometry count, n, remains fairly constant while the effective rendered

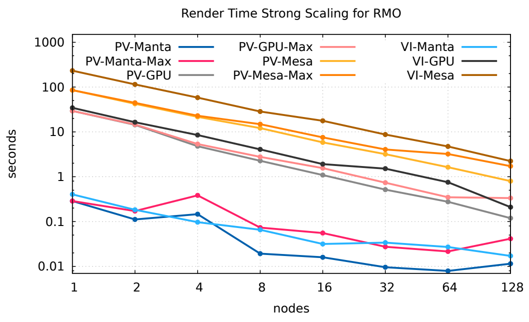


Figure 8: *Strong scaling timings of an isosurface of the RMO dataset.*

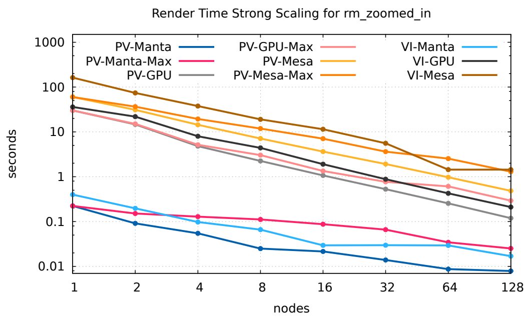


Figure 9: *Strong scaling timings of an isosurface of the RMI dataset.*

area, k, shrinks as the scene is broken up to distribute among more nodes. It is not clear how the GPU performance could replicate this, except with the use of an intelligent hierarchical LOD algorithm that averages sub-pixel geometry in a pre-processing step, such as with GigaVoxels [CNLE09]. The maximum rendering times are shown with the Max designation in the graph. These times display the average of the maximum times over all the nodes for each run. Ray tracing displays a large difference between the average and maximum render times as the data-parallel scene distribution is not ideal for ray tracing and some nodes finish significantly faster than others. In cases where we achieve perfect work distribution the average and maximum average numbers would be identical and it is clear that increasing the number of nodes increases variance between render times. Manta displays superior performance on a single node as it uses a BVH to accelerate over large amounts of geometry and skip over occluded geometry.

In our tests the main performance bottleneck for the hardware-accelerated rasterization renderings appear to be due to a single display list being compiled over immediate mode rendering calls. This means that updates to a color require rebuilding the display list. Display lists can easily be disabled, however this results in a drop in rendering performance. We found that a single display list resulted in poor performance with large data counts and in some cases would even crash the program requiring immediate mode rendering to be used for large polygon counts. Splitting up the display lists in our tests showed over 10x performance increases in some cases, however results varied across different hardware and drivers. Using vertex and color buffers would result in significantly fewer draw calls per update which could drastically decrease rendering times. However, this would not affect the asymptotic runtime of the underlying algorithm and these methods are not used in released VisIt or ParaView versions as of this writing.

5.1.2. Strong Scaling

Figure 8 shows strong scaling of the 316 million-triangle contour from timestep 273 of the Richtmyer-Meshkov Instability simulation. At a single node the benefit of the BVH in Manta can clearly be seen as the ray tracer attains several times the performance of the brute-force OpenGL implementation, rendering in about .1 seconds with just two nodes.

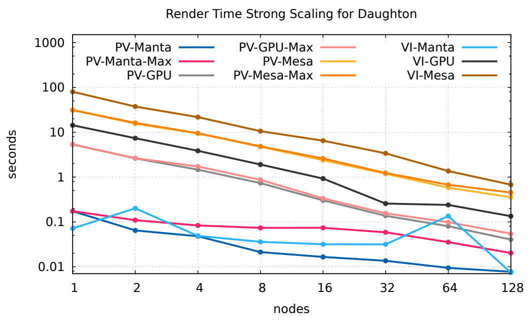


Figure 10: *Strong scaling timings of the VPIC dataset.*

VisIt is slightly slower in many of our runs. Through testing we found this is likely because VisIt uses a texture map for colors instead of a single call to glColor in ParaView and ParaView uses triangle strips whereas VisIt does not. Manta in VisIt does not scale as well in higher node counts as our ParaView implementation because of our current VisIt im-

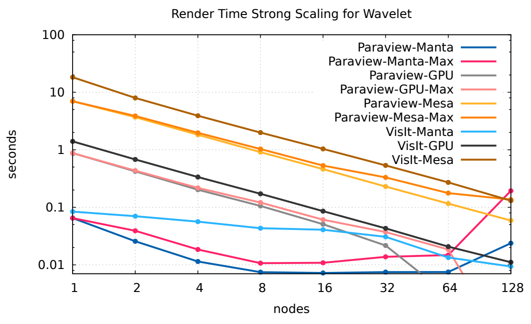


Figure 11: *Strong scaling timings of the wavelet dataset.*

plementation uses the current vtkRenderWindow rather than a custom Manta render window which limits performance to around .05 seconds from calls to glDrawPixels to copy the image buffer into the OpenGL context. Since the VisIt bottleneck results in a maximum of about 20 fps, which is the compositing limit of Binary-swap, this is not a significant issue which inhibits interactivity, and one likely to be fixed in a future update.

Figure 9 displays strong scaling numbers for a zoomed in view of the RM dataset as seen in Figure 5(b). While this has little to no affect on the rasterization algorithm, in ray tracing this results in different scaling performance. While the dataset is broken up and distributed across nodes, only a few of those nodes actually have a portion of the visible data. The ray tracer was already culling out the non-visible data when the entire dataset was rendered on a single node. Thus, roughly the same amount of visible data was being rendered on a single or small number of nodes. The average render times drop similarly to the RMO render times, however the maximum render times are generally worse for the RMI renderings than for the RMO renderings at 8, 16, 32 and 64 nodes. This increase in maximum render time shows the effects of data-parallel rendering work distribution as a few nodes do more work. The maximum time eventually drops, which is likely when the data subsets per node were small enough to split up most of the visible geometry. Ize et al. [IBH11] reported roughly linear scaling behavior through ray-parallel work distribution up to 60 fps until approaching a bottleneck introduced by sending pixels over their Inifini-Band network at around 127 fps. A reworking of ParaView's work distribution could show similar scaling performance, however our timings suggest that a data-parallel implementation within the existing framework scales to interactive rates. Figure 10 shows timings for the VPIC dataset which is made up of 102 million triangles. In this dataset, the GPU accelerated OpenGL render times for ParaView achieve below the .1 second mark needed for 10fps interaction, but only at 64 to 128 nodes when the effective triangle count per node is around a million triangles per node.

A contour of a wavelet was created to test performance with occluded geometry in a predictable fashion through 27 overlapping isosurfaces. While the RM and VPIC datasets contain a lot of overlapping geometry, each uses a single isosurface. Performance for the 16 million triangle wavelet dataset is shown in Figure 11. Manta rendering times are below .1 seconds on a single node and rendering time drops significantly with additional nodes, however the rendering times appear to reach a bottleneck at 8 nodes which is showing the bottleneck introduced from rendering a mostly empty scene and copying image buffers from Manta into VTK. Some spikes in timings are seen at higher node counts, this is likely from a single node or a few nodes running slower due to the high maximum render time at 128 nodes for ParaView Manta.

6. Conclusion

With this study we have shown scaling timings of widely used visualization tools on real-world datasets. In order to explore alternative rendering algorithms, we have integrated a software ray tracing solution into common tools which has demonstrated superior rendering performance with large polygon counts over the built-in OpenGL hardware rendering and Mesa software rendering methods on a single process. Through our system, VisIt and ParaView have shown rendering times decrease by as much as 100x in our tests compared to brute-force rasterization and can achieve interactive rendering performance for large geometry counts on large cluster environments without the need for specialized hardware acceleration. In the future, in situ visualization is expected to become increasingly important with the move to exascale. A system, such as the one described in this paper, which exhibits scalable rendering performance would be a good fit for in situ visualization on compute clusters which may lack graphics acceleration. We have shown that ray tracing implementations can scale with increasing node counts across a variety of different datasets and views using existing data-parallel work distribution and sort-last image compositing. While the scaling performance is currently dependent on the view and data distribution, integrating view-dependent data partitioning into VTK or using a system such as Chromium [HHN+02] could potentially alleviate a lot of compositing work and sub-optimal data distribution present in ParaView and VisIt. Much work still needs to be done to accommodate visualization, including maximizing single node performance for other parts of the visualization pipeline such as reading, isosurfacing, calculator operations, and building acceleration structures. GPU accelerated ray tracing is another avenue of research which was not considered for this paper but worth further study. The design of Manta has differing data representation and frame behavior resulting in wasted memory and a total frame time of the aggregate of compositing and rendering times instead of the maximum of the two. Our Manta plugin is currently available in the source version of ParaView and was released in the binary distributions for Linux and Mac OS in version 3.10. Our VisIt implementation is expected to be released to the public in VisIt 2.5.0 or a future release.

7. Acknowledgements

This research was sponsored by the National Nuclear Security Administration under the Advanced Simulation and Computing program through DOE Cooperative Agreement #DE-NA0000740, and by Award No. KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST), and DOE SciDAC:VACET, NSF OCI-0906379, NIH-1R01GM098151-01. Special thanks to Hank Childs and Brad Whitlock.

References

[BWP04] Ji Bittner, Michael Wimmer, and Harald Piringerand Werner Purgathofer. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3), 2004. 2

[CGM⁺06] Andy Cedilnik, Berk Geveci, Kenneth Morel, James Ahrens, and Jean Favre. Remote large data visualization in the paraview framework. 2006. 2, 4

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19:547–554, October 1976. 2

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, I3D 09, pages 15–22, New York, NY, USA, 2009. ACM. 8

[CPA⁺10] Hank Childs, David Pugmire, Sean Ahern, Brad Whitlock, Mark Howison, Prabhat, Gunther H. Weber, and E. Wes Bethel. Extreme scaling of production visualization software on diverse architectures. *IEEE Computer Graphics and Applications*, 30:22–31, 2010. 2

[DGP04] D. E. DeMarle, C. Gribble, and S. Parker. Memory-savvy distributed interactive ray tracing. In *Proc. of Eurographics Symposium on Parallel Graphics and Visualization*, pages 93–100, 2004. 3

[FQK08] Z. Fan, F. Qiu, and A. E. Kaufman. Zippy: A framework for computation and visualization on a gpu cluster. *Computer Graphics Forum*, 27(2):341–350, 2008. 2

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238, New York, NY, USA, 1993. ACM. 2

[GKY08] Enrico Gobbetti, Dave Kasik, and Sung-eui Yoon. Technical strategies for massive model visualization. In *SPM '08: Proceedings of the 2008 ACM symposium on Solid and physical modeling*, pages 405–415, New York, NY, USA, 2008. ACM. 3

[HHN⁺02] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: A stream-processing framework for interactive rendering on clusters, 2002. 9

[IBH11] Thiago Ize, Carson Brownlee, and Charles D. Hansen. Revisiting parallel rendering for shared memory machines. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pages 61–69, 2011. 3, 9

[Ize09] Thiago Ize. *Efficient Acceleration Structures for Ray Tracing Static and Dynamic Scenes*. PhD thesis, University of Utah, 2009. 5

[Kit10] Kitware. *Paraview - Open Source Scientific Visualization*, 2010. http://www.paraview.org/. 2

[LLN10] LLNL. *VisIt Visualization Tool*, 2010. https://wci.llnl.gov/codes/visit/. 2

[lMPH94] Kwan liu Ma, James S. Painter, and Charles D. Hansen. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14:59–68, 1994. 2, 4

[Lue01] D.P. Luebke. A developer's survey of polygonal simplification algorithms. *Computer Graphics and Applications, IEEE*, 21(3):24 –35, 2001. 2

[MC98] T. Mitra and T.-C. Chiueh. Implementation and evaluation of the parallel mesa library. In *Parallel and Distributed Systems, 1998. Proceedings.*, pages 84 –91, dec 1998. 2

[MDG⁺10] L. Marsalek, A. Dehof, I. Georgiev, H.-P. Lenhof, P. Slusallek, and A. Hildebrandt. Real-time ray tracing of complex molecular scenes. In *Information Visualization: Information Visualization in Biomedical Informatics (IVBI)*, 2010. 3

[MP01] Kwan-Liu Ma and Steven Parker. Massively parallel software rendering for visualizing large-scale data sets. *IEEE Computer Graphics and Applications*, 21:72–83, 2001. 3

[NAWS11] B. Nouanesengsy, J. Ahrens, J. Woodring, and H. Shen. Revisiting parallel rendering for shared memory machines. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pages 31–40, 2011. 2

[Par02] S. Parker. Interactive ray tracing on a supercomputer. pages 187–194, 2002. 3

[SBB⁺06] A. Stephens, S. Boulos, J. Bigler, I. Wald, and S. G. Parker. An application of scalable massive model interaction using shared memory systems. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, pages 19–26, 2006. 3

[SeY08] David Kasik Dinesh Manocha Sung-eui Yoon, Enrico Gobbetti. *Real-Time Massive Model Rendering*. Morgan and Claypool, 2008. 2

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1), January 2007. 3

[WSB01] I. Wald, P. Slusallek, and C. Benthin. interactive distributed ray tracing of highly complex models. In *Proc. of Eurographics Workshop on Rendering*, pages 274–285, 2001. 3

[YWG⁺10] Hongfeng Yu, Chaoli Wang, Ray W. Grout, Jacqueline H. Chen, and Kwan-Liu Ma. In situ visualization for large-scale combustion simulations. *IEEE Comput. Graph. Appl.*, 30:45–57, May 2010. 2

[ZMHH97] Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E. Hoff, III. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 77–88, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. 2