LA-UR- 10-07014

Approved for public release;
distribution is unlimited.

Title:	Approaching the Exa-scale: A Real-World Evalutation of Rendering Extemely Large Data Sets
Author(s):	John Patchett CCS-7 148176 Carson S. Brownlee CCS-7/University of Utah 229465 Christopher J. Mitchell CCS-7/University of Florida 229505 James P Ahrens CCS-7 113788 Chuck Hansen University of Utah Li-Ta Lo CCS-7 194699
Intended for:	Pacific Visualization Conference March 1-4, 2011

Form 836 (7/06)

Approaching the Exa-scale: A Real-World Evalutation of Rendering Extemely Large Data Sets

Submission 178

ABSTRACT

Extremely large scale analysis is becoming increasingly important as supercomputers and their simulations move from petascale to exascale. The lack of dedicated hardware acceleration for rendering on today's supercomputing platforms motivates our detailed evaluation of the possibility of interactive rendering on the supercomputer. In order to facilitate our understanding of rendering on the supercomputing platform, we focus on scalability of rendering algorithms and architecture envisioned for exascale datasets. To understand tradeoffs for dealing with extremely large datasets, we compare three different rendering algorithms for large polygonal data: software based raytracing, software based rasterization and hardware accelerated rasterization. We present a case study of strong and weak scaling of rendering extremely large data on both GPU and CPU based parallel supercomputers using ParaView, a parallel visualization tool. We use three different data sets: two synthetic and one from a scientific application. At an extreme scale, algorithmic rendering choices make a difference and should be considered while approaching exascale computing, visualization, and analysis. We find software based ray-tracing offers a viable approach for scalable rendering of the projected future massive data sizes.

1 INTRODUCTION

Ever increasing data sizes result in greater challenges for preexisting visualization methods. In this paper we explore visualization strategies for large-scale data analysis focusing on large-scale rendering. A prime motivator for our study are our experiences with a team of researchers using the VPIC code to investigate issues in magnetic reconnection on various computers including two of the top three on the top 500: Oak Ridge National Laboratory's Jaguar and Los Alamos National Laboratory's Roadrunner. This magnetic reconnection team views interactive visualization as integral to their workflow for each iterative run of their simulation. As an example one of their runs was on the RoadRunnder supercomputer consuming 4096 processors to compute on a 8096x8096x448 grid. They currently use striding and subsetting to make the visualization operations more practical for their interactive workflow but would like to visualize all of the data rather than just sub-samples.

A standard visualization workflow for extremely large data, like that of the magnetic reconnection researchers, normally involves an ordered set of activities: application simulation, visualization algorithms, rendering, and display as shown in Figure 1. Each of these stages produces a resulting 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 which is suited to each stage. The simulation typically writes data to disk that is proportional to the size of the run. This data is then read from disk and geometry is generated and saved to disk. The geometry is then read back from a remote resource or read and transferred to a different filesystem, which is often at a different

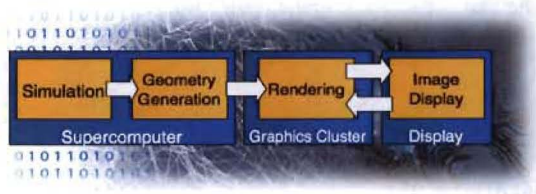


Figure 1: A remote visualization workflow with a seperate compute cluster and rendering cluster.

geographic location. Transferring 10 Terabytes over a 5Mbps connection with 10 simultaneous streams would take almost 19 days to transfer to a remote resource such as a render cluster. Where high speed gigabit connections exist to the rendering cluster, transferring 10 Terabytes over a 1Gb connection would still take several hours to transfer.

As we approach exascale computing, data sizes are increasing and the typical output size of geometry increases with the size of the input data. Thus, transfer of data offsite is a less viable option. As compute clusters grow in size and cost, so must the supporting visualization clusters to handle the increased data. This process of saving data, moving data, and then analyzing data prohibits the scientists' ability to quickly analyze and rerun simulations in cases of error.

To avoid long data transfer times rendering can be done directly on the supercomputer. This new workflow is shown in Figure 2.

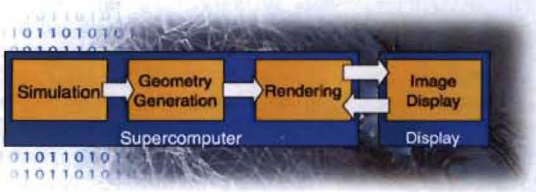


Figure 2: A remote visualization workflow with rendering done on the supercomputer.

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 type of interactivity has traditionally come at the cost of buying a second GPU-based supercomputer, albeit typically smaller than the supercomputer that produced the simulation results. We ask the following question: Can this interactivity be accomplished on massive data sets using the computing platform? We believe the question of rendering on the computing platform is important as we focus on the exascale future and allows for in situ visualization techniques. This work will benefit the super computing community providing insight to better understand the algorithmic and architectural choices and their implications of performing visualization and analysis on increasingly large data sets.

To understand tradeoffs for dealing with extremely large datasets, we compare three different rendering algorithms for large polygonal data: software based raytracing, software based rasterization and hardware accelerated rasterization. The contribution of this paper is a case study of strong and weak scaling of rendering extremely large data on both GPU and CPU based parallel supercomputers using ParaView, a real world parallel visualization tool. We use three different data sets: two synthetic and one from a real world application.

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. 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 the data to a GPU cluster for rendering is a well developed practice that displays large datasets at very high framerates by splitting up data for rendering [4] and compositing the resulting images together using algorithms such as the Binary-Swap method [13]. While there is a lot of time devoted to disc and network I/O this is the most robust for visualization as any number of software packages designed to work on clusters can be used to visualize the data and often at high frame rates. These clusters can also be used to generate geometry and feed back smaller sets of geometry for rendering to a desktop machine [12]. Common visualization tools that run on GPU clusters include such programs as Paraview [2, 8] and VisIt [11] which present Client/Server parallel rendering frameworks built on top of the Visualization Toolkit (VTK).

Massive polygon rendering presents challenges for traditional rasterization methods. OpenGL relies on hidden surface removal with a simple Z-buffer test to determine whether to shade a triangle. 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 vastly degrade performance resulting in a linear or greater decrease in speed in relation to the number of triangles. Approaches to address this have included spatial subdivision schemes which are used for occlusion culling hidden triangles. The acceleration structure's nodes are traversed front-to back and rendered. If the current Z-buffer's values are less than the Z value of the next node than the entire node can be skipped. Implementations of these methods include the hierarchical Z-buffer [6] and the hierarchical occlusion map [23] which optimize performance through hierarchical Z-Pyramids. Prioritized-layred projection also provides an approximate version for instances where exact results aren't required [9]. GPU optimized occlusion queries developed and allowed for querying if faces of the acceleration structure are behind the current Z-buffer in hardware [7]. However, since these methods require front-to-back traversal of the polygons or are not efficient, they are rarely used.

The most efficient method for looking at large amounts of data generated on a cluster node would be to use the same node to render with the same acceleration structures used for the simulation. This would require no extra I/O except for a rendered image and minimal rendering time. This method is ideal but requires a great deal of customization to the simulation software as well as the visualization software. A compromise is to send the generated data to another visualization application running on the same cluster that utilizes software rendering. One such example would be using Paraview paired with a software implementation of Mesa. Using currently available implementations, software rasterization is often very slow.

Ray tracing on clusters for visualizing large scale datasets is a well developed field with benefits over traditional rasterization methods. Ray tracing performance has been shown to scale very linearly from one to hundreds of processors [18], but limited by network latency to around 20 frames per second [21]. Tracing rays also scales very well with the amount of geometry in the scene [14, 22] due to the logarithmic acceleration structures used. Advanced cluster based ray tracing methods can split up data and rays by image-space or data-space. The former relies on rays around similar areas of the 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 disc reads [3]. The cost of tracing large datasets can be reduced greatly by intelligent techniques such as tracing groups of rays in packets [20] and slice based techniques which can determine geometry intersection for groups of rays and exploit SIMD vector operations. Additionally cache dependent rendering can provide significant speedups for memory intensive ray tracing operations [19]. 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 [5] and so methods to reduce memory requirements are still vital to software ray tracing. The real-time ray tracing software Manta incorporates packets but is only designed to be run on SMP machines, not clusters [1]. This design makes it ideal when combined with other cluster based visualization programs such as Paraview which handle data distribution and image compositing.

3 Rendering Methods

For this paper we explore three commonly used rendering methods: hardware accelerated rasterization, software based rasterization, and software based ray tracing. Each method has its advantages and drawbacks. After evaluating each method, software ray tracing was chosen as the method to use going forward for large-scale data visualization.

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 rendering times that scale linearly with the amount of geometry in the scene. In order to achieve better scaling, occlusion algorithms might be used and approximative LOD methods might be utilized. It remains unclear, however, how well these methods will scale into the billions of triangles much less into petascale and exascale sized datasets. Therefore, we do not consider them.

Software rasterization through Mesa 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 acclerated rasterization, however, is that it does not require additional graphics hardware and can utilize large system 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 largely screen space dependent performance with logarithmic scaling to geometry, ray tracing provides great performance for large geometry counts; even those that contain sub-pixel geometry. Using an acceleration structure to test ray intersections also allows easy and straightforward occlusion culling 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 as our implementation was intended for compute clusters without hardware acceleration. We test these methodologies with ParaView.

4 VISUALIZATION USING PARAVIEW

ParaView is an open-source visualization framework designed for local and remote visualization of a large variety of datasets. It is also designed to run on PC hardware up to large cluster arrays using client/server separation and parallel processing. Server/data separation allows ParaView to be broken into three main components: a data server, render server, and client [2]. Much of the actual rendering code is based around the Visualization Toolkit (VTK) while much of the client/server model is unique to ParaView.

4.1 Data Distribution

ParaView's data server abstraction layer allows for operations such as processing data on one node and sending the resulting geometry to another node for rendering. This allows for changing the data processing and rendering pipeline across heterogenous architectures for balanced workload distribution. When rendering on multiple nodes, sort-last compositing is required. Sort-last data and geometry distribution balances data processing and rendering, however, it requires a composite operation for each node used. This method of data distribution is good for rasterization and not necessarily optimal for ray tracing where render time is dependent more on screen distribution than data distribution. When zoomed out, distributed cluster-based ray-tracing often produces a 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 one render node. Thus using ParaView's work distribution methods is not optimal, however, we have found it to be usable in practice as shown in Section 5.

4.2 Compositing

The sort-last distribution requires a compositing step taking all the partial results and combining them into a final image using the depth and alpha values of each pixel. Distributing the compositing work across a cluster is vital for efficient cluster utilization. For this we implement Binary-Swap compositing in ParaView [13]. Binary-Swap is an efficient parallel compositing algorithm that exchanges portions of images between processes to produce a correct rendered result.

Figure 3 shows a diagram of the compositing process. Four nodes shown at the top each have a portion of geometry. The next step is to split those images in half and composite each half together. Thus L1 and L2 are composited together and R1 and R2 are composited together such that each half of the image now has the composited result from half of the geometry. At the next step each node is assigned a quarter of the final image which composites that quarter of the image together. At each step of the algorithm each node is doing an equal amount of work thus fully utilizing the cluster of nodes. The final step is then to take each quarter and tile them together into the final composited image.

Binary-Swap is not the default compositing scheme in ParaView which is IceT [16]. IceT breaks up compositing to only composite portions of the image that are actually rendered instead of the entire screenspace. This can potentially save a lot of network traffic by not compositing unused space if each node is only rendering a small portion of the final image. The IceT library was not used because its performance is very viewpoint and data dependent, whereas Binary-Swap essentially represents the worst-case performance of IceT consistently.

4.3 ParaView Manta Plugin

The Manta rendering architecture is displayed in Figure 4. Image pixels are generated by a pixel sampler which then generates rendering tiles by an image traverser which are distributed among threads by a load balancer. These tiles are then broken up into packets which are then traced by each thread through the scene in the

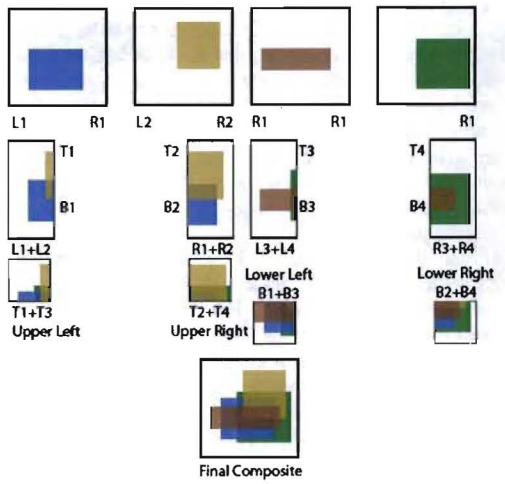


Figure 3: The stages of Binary-Swap Compositing.

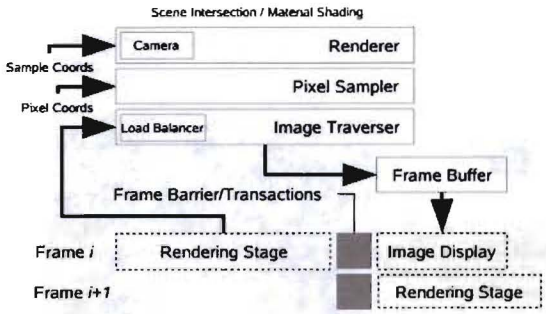


Figure 4: Manta architecture.

rendering stage. At the end of a rendering step the threads are syncronized and state is updated. This is where state can safely be accessed and modified outside of Manta through a series of callbacks called transactions. The image is displayed while the next rendering stage is running. This results in a one frame delay between rendering and display. Since ParaView is set to render only on an update this means that in the default system Manta would need to finish rendering its current scene, get the updated scene from ParaView, display the last rendered image, render the modified scene, synchronize with no updates, render the same scene again while displaying the image, and then by the next synchronization the image would finally be available through a transaction.

The rendering architecture of Manta was modified slightly to have the image display before the rendering stage as a seperate synchronization step which is only released upon a render event in ParaView. This eliminates uneccessary rendering before a render event and the final unecessary rendering step to get the image display. The Manta context is created in ParaView through a custom render window and implemented through a plugin. The plugin must be loaded on both the client and server and any rendering contexts closed. A Manta render instance can then be selected. Certain VTK calls are then redirected to Manta implementations through a custom Manta object factory such as a custom actor class that keeps track of Manta related state for each object and a custom PolyDataMapper which sends data to Manta and creates acceleration structures specifically

for ray tracing. Unfortunately because of the differences in how meshes are represented between Manta and ParaView there is currently a lot of memory overhead due to geometry duplication. A rendering call triggers Manta to release its rendering lock and render a frame. The frame is then copied back to ParaView. Due to the differences in how an image is stored this requires an image conversion step. ParaView then displays the rendered image or sends the image out for compositing if it is being rendered remotely through a cluster. The compositing step required the introduction of a Z-buffer into the ray tracer which simply deposits the resulting depth value into the buffer after rendering instead of using the buffer in the rendering process itself.

Precomputation must also be done with each change in geometry. With each new Manta actor introduced into the scene acceleration structures are generated. For very large scenes consisting of millions of triangles this can take several seconds of precomputation 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 Heirarchie, BVH, as it gave the best performance for most types of goemetry.

Additional GUI elements were also added to ParaView to allow for ray tracing specific scene and material properties. These additional options include such materials as glass and ambient occlusion as well as multisampling and threading options. The result is shown in Figure 5.

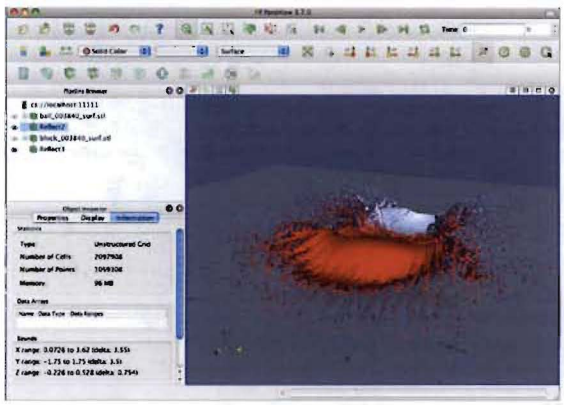


Figure 5: Manta running in Paraview on a single multi-core node using shadows and reflections rendering the impact of an aluminum ball on an aluminum plate.

5 Results

We evaluated the rendering performance of various methods on two supercomputing platforms, Lobo, a Los Alamos Linux compute cluster and Longhorn, a latest generation GPU-based visualization cluster at the Texas Advanced Computing Center (TACC). We also used a single next-generation multi-core node to test rendering performance. We used three datasets of varying sizes including randomly generated triangles, a synthetic wavelet dataset, and a dataset from Los Alamos's plasma simulation code, VPIC. Three different rendering packages were tested: Manta, an open-source raytracer, Mesa, a software OpenGL ray tracer, and hardware-acclerated OpenGL, all running within the ParaView application.

Supercomputing Platforms

- **Lobo** is a 272 node 4X DDR InfiniBand connected cluster of AMD based nodes. It has 32 GB of RAM and 4 AMD opteron model 8354 quad core processors, for a total of 16 cores, per node at 2.2 GHz. Each core has a 64KB L1 cache, and a 512KB L2 cache, while each quad core shares a 2MB L3 cache. Lobo is a TriLab Linux Capacity Cluster (TLCC) system, similar systems are available to users at Los Alamos, Livermore, and Sandia National Laboratories.

- **Longhorn** is 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 48 GB of RAM. Each node of Longhorn also has 2 NVidia FX 5800 GPUs.

- **Kratos** Kratos is an HP Proliant DL785 G5 with 8-quadcore AMD Operton 8380 processors at 2.5 GHz with 128GB RAM.

Datasets

- **Random Triangles** We created a random triangle strip test dataset. An image showing a rendering of 8 million of these triangles is shown in Figure 6(a).

- **VPIC Visualization-Generated Triangles** Using an early timestep from the VPIC plasma simulation, we calculated collections of isosurfaces that produced 1, 2, 4, 8, and 16 million triangles for use in our evaluation. At the start of this simulation, these isosurfaces form two parallel nearly planar surfaces. A view of this data set can be seen in Figure 6(b).

- **Wavelet Triangles** 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 requested quantity of triangles. The isosurfaces are nested within each other. This dataset highlights performance optimizations in the renderers that result from occlusion culling. Images produced with 8 million triangles from each of these datasets are shown in Figure 6(c).

5.1 Single Node Rendering Performance

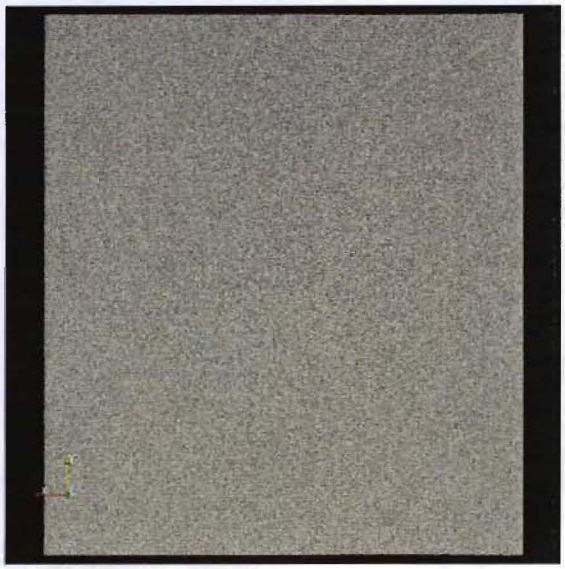
Figures 7, 8, and 9 show a comparison of single node performance of the three rendering methods we evaluated: GPU accelerated rasterization on Longhorn, CPU ray tracing on Lobo with Manta, and CPU rasterization on Lobo with Mesa. Figure 7 shows the performance of random triangles. The x-axis shows millions of triangles in the rendered scene, and the y-axis shows the average framerate in frames per second as the camera rotates around the scene in 3 degree increments. The random triangle benchmark produces the worst-case performance of the three datasets. We believe this is due to the irregularity of the data and lower depth complexity in the dataset.

To achieve the optimal performance on Lobo, Mesa was run with MPI using 16 processes, Manta was run with 1 process using 16 threads, and Longhorn was run with 1 process using a single GPU. Of the three renderers, the Mesa renderer performs the worst, rendering performance did not exceed seven frames per second on any test. The Manta renderer shows improve CPU-based rendering performance over Mesa. The Manta and GPU performance converge at 16 million polygons in all three dataset test cases.

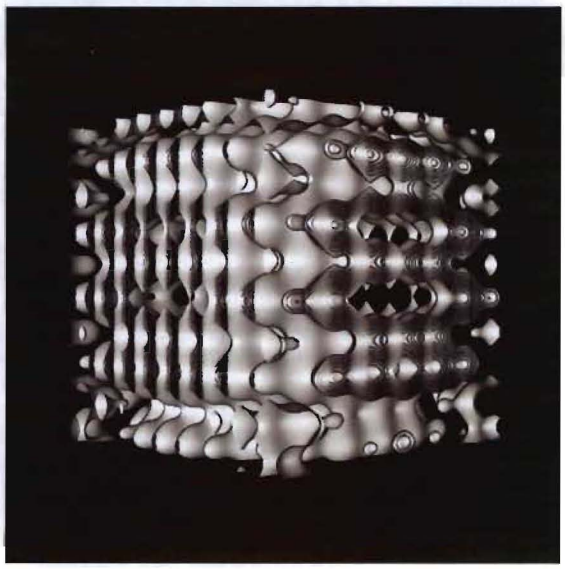
Figure 10 shows the results of running Manta/ParaView on Kratos using 32 threads and single node GPU accelerated ParaView on varying sized random triangle datasets up to 256 million triangles. The results show good scaling of ray tracing well beyond 16 million triangles. As data sizes increase, Manta's logarithmically scaling performance provides for interactive framerates well beyond the unoptimized GPU performance.

On both Lobo and Kratos, the rendering performance on 16 million triangles using a CPU based cluster node is very similiar to that of the GPU based cluster. As the number of polygons increases up to 256 million triangles, we observe the Manta performance of 5-10

(a) Random Triangles



(b) VPIC Contours



(c) Wavelet Contours

Figure 6: 8 million triangle version of the 3 test data sets.

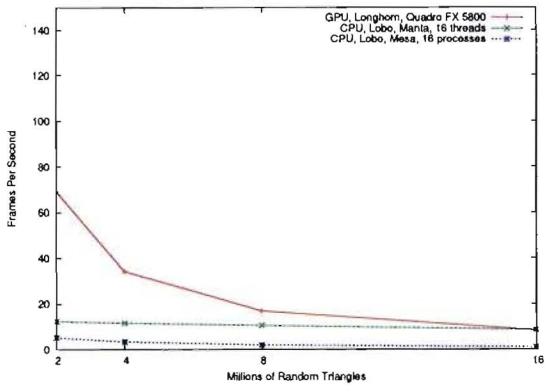


Figure 7: Single node rendering performance on random triangles.

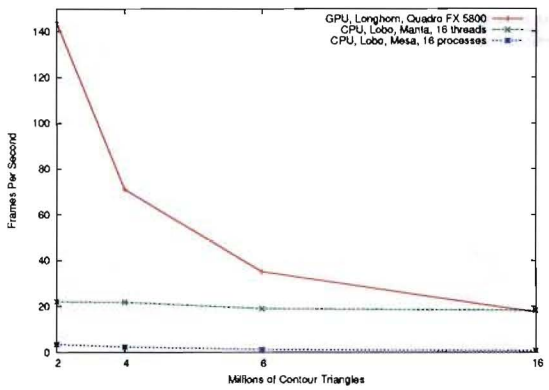


Figure 8: Single node rendering performance on wavelet contours.

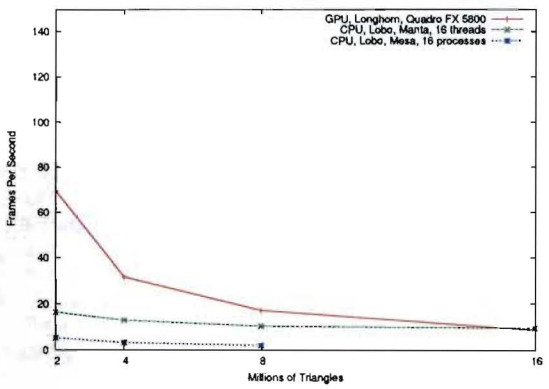


Figure 9: Single node rendering performance on VPIC contours.

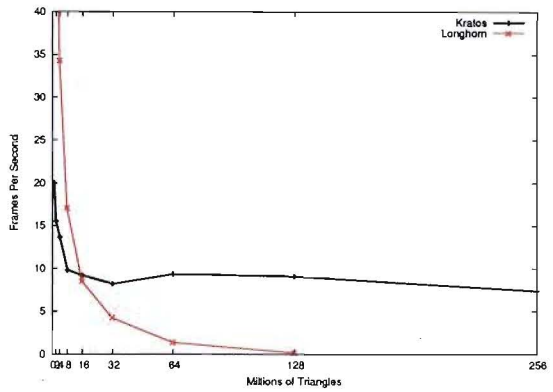


Figure 10: Single node rendering using Manta/Kratos and Longhorn/GPU to test performance with large polygon counts.

frames per second. As discussed in Section 3, this stable performance is due to the Manta's algorithmic run time of $k * O(log(n))$ where k is the image size, n is the number of polygons. This log performance is due to Manta's tree-based accelerated polygon/ray intersection data structure. Decreasing GPU performance is due to an algorithmic GPU run time of $O(n)$, where n is the number of polygons. Note that we do not expect image size, k to significantly increase (to gigapixel or terascale sizes) due to the limits of the human visual system.

5.2 Cluster Compositing

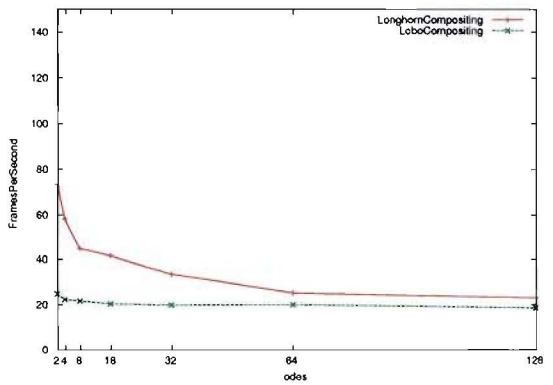


Figure 11: Compositing baseline for Lobo and Longhorn.

Image compositing is a parallel algorithm that merges images from each process and produces a final correct image. In our performance tests we used a binary-swap compositing algorithm[10]. This is an efficient parallel compositing algorithm that exchanges portions of images between processes to produce a correctly rendered result. The IceT compositing library[17] is the default compositing scheme in ParaView. Although very efficient, IceT's performance is also very data dependent and IceT optimizations integrate directly in with the renderer. In order to clarify our rendering and compositing performance results we used the binary swap compositor. Figure 11 shows the compositing performance baseline for both Lobo and Longhorn. The x-axis shows the number of physical compute nodes not processors (for processor core counts, multiply by 8 for Longhorn and 16 for Lobo). The y-axis shows frames per second. As the number of nodes increase both show asymptotic behaviour. The asymptote is completely dependent on the speed and quality of the nodes and the network. The Longhorn network is QDR infiniband and is expected to be twice as fast as Lobo's DDR network. Noise on the compute nodes and/or the network, that is more likely to be encountered for larger node counts, can adversely effect the compositing performance. To time the compositing cost in ParaView we ran in parallel and rendered empty scenes. Note that both machines provide about 20 frames per second at 128 nodes. Recall, the Longhorn GPU cluster has a faster network than the Lobo CPU cluster. This difference will effect our rendering performance results.

5.3 Cluster Scaling

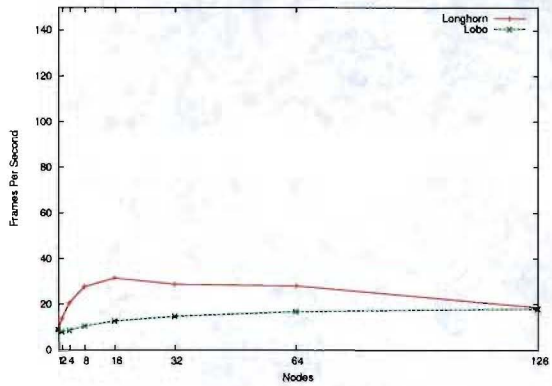


Figure 12: Strong scaling of 16 million random triangles.

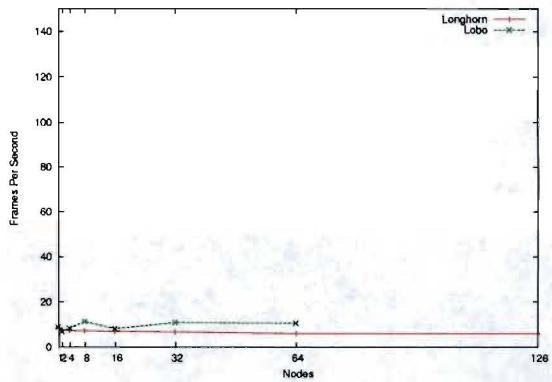


Figure 13: Weak scaling with 16 million random triangles per node.

Possible parallel rendering approaches include sort-first methods in which polygons are sorted prior to rendering and distributed to processors that are responsible for a portion of the display[15]. Sort-last methods distribute the subsets of the polygons to each processor. A full display image is rendered along with a depth image and then a compositing step produces a final image, depth-sorted from all nodes. In our study, we use sort-last rendering and compositing. The total time to produce an image is the sum of the rendering and compositing time (i.e. *image-time* = *render-time* + *composite-time*).

5.3.1 Strong Scaling

A strong scaling study keeps the problem size constant while increasing processor resources to solve the problem. Figure 12 shows strong scaling of 16 million random triangles on both Longhorn and Lobo. The total number of triangles processed across the entire job was held constant at 16 million. For this strong scaling graph the number of triangles per node decreases, specifically each processor is assigned $\frac{Total\ Triangles}{Total\ Nodes}$. As the number of nodes increases the number of triangles per node decreases and thus the time for each node to render those triangles decreases. As the number of rendering resources increases, the rendering time would dissipate and the total time would equal compositing (*image-time* = *composite-time*). This can be seen in the Longhorn results as the smaller polygon counts are rendered very quickly. Longhorn shows an initial increase in overall performance from splitting the rendering load among more processors. As the rendering time decreases, compositing time dominates and we note that between 16 and 32 nodes frame rate decreases. Lobo with CPU rendering, doesn't show the initial performance increase since the decrease in *render-time* is similar to the increase in *composite-time*. On both machines however, *composite-time* becomes the upper bound. When parallel rendering a fixed number of polygons with an increasing number of rendering resources, ultimately maximum performance is limited to the performance of compositing.

5.3.2 Weak Scaling

In a weak scaling study the problem size scales with the processor resources. For our testing we chose to assign 16 million triangles per node. The total triangles for a data point in Figure 13 can be calculated by multiplying 16 million by the number of nodes. Since the *render-time* remains constant and the *composite-time* approaches a constant as we scale we expect the *image-time* to approach a constant. Our weak scaling results can be seen in Figure 13. We rendered up to 2 billion polygons on Longhorn and 1 billion on Lobo.[1] Recall that rendering 16 million polygons on a single node produced similar timings for both CPU and GPU based nodes and therefore we see relatively similar results. This weak scaling study shows the ability to render large quantities of polygons at similar rates with both CPU and GPU resources.

6 Conclusion

With ever increasing data sizes, we have shown that integrating a software ray tracer, which scales in $k * O(log(n))$, into an open-source visualization tool is a wise investment for the future. For massive data sizes, we believe that an additional GPU-based visualization cluster is unnecessary. Rendering is executed directly on the supercomputer eliminating long data transfer times to a networked visualization cluster. Utilizing CPU system memory allows for the visualization of larger polygon counts. Ray tracing scales very well with the amount of geometry without complicated occlusion steps or approximative LOD methods. In our performance evaluation software-based ray-tracing surpassed GPU performance at larger triangle counts.

Acknowledgements

References

[1] J. B. I. W. A. Stephens, S. Boulos 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.

[2] A. Cedilnik, B. Geveci, K. Morel, J. Ahrens, and J. Favre. Remote large data visualization in the paraview framework. 2006.

[3] 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.

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

[5] E. Gobbetti, D. Kasik, and S.-e. 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.

[6] N. Greene, M. Kass, and G. 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.

[7] H. P. W. P. Ji? Bittner, Michael Wimmer. Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum*, 23(3), 2004.

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

[9] J. T. Klosowski and C. T. Silva. The prioritized-layered projection algorithm for visible set estimation. *IEEE Transactions on Visualization and Computer Graphics*, 6(2):108–123, 2000.

[10] K. liu Ma, J. S. Painter, and C. D. Hansen. Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics and Applications*, 14:59–68, 1994.

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

[12] E. Luke and C. Hansen. Semotus visum: a flexible remote visualization framework. In *Proc. of the conference on Visualization '02*, pages 61–68, 2002.

[13] J. S. H. C. D. K. M. F. Ma, K.-L. Painter. Parallel volume rendering using binary-swap compositing. *IEEE COMPUTER GRAPHICS AND APPLICATIONS*, 14(4):59, 1994.

[14] K.-L. Ma and S. Parker. Massively parallel software rendering for visualizing large-scale data sets. *IEEE Computer Graphics and Applications*, 21:72–83, 2001.

[15] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics Applications*, 14(4):23–32, 1994.

[16] K. Moreland, B. Wylie, and C. Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. *Parallel and Large-Data Visualization and Graphics, IEEE Symposium on*, 0:85–92, 2001.

[17] K. Moreland, B. Wylie, and C. Pavlakos. Sort-last parallel rendering for viewing extremely large data sets on tile displays. *Parallel and Large-Data Visualization and Graphics, IEEE Symposium on*, 0:85–92, 2001.

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

[19] M. Pharr, C. Kolb, R. Gershbein, and P. Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 101–108, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[20] A. Reshetov, A. Soupikov, and J. Hurley. Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1176–1185, New York, NY, USA, 2005. ACM.

[21] I. Wald, C. Benthin, A. Dietrich, and P. Slusallek. Interactive ray tracing on commodity pc clusters. *LECTURE NOTES IN COMPUTER SCIENCE*, pages 499–508, 2003.

[22] 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.

[23] H. Zhang, D. Manocha, T. Hudson, and K. 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.

[1] A NUMA memory-allocation issue prevented us from rendering 2 billion triangles on Lobo.