NOTE: Use these buttons to print or save the form. DO NOT use the browser tool bar. SAVE PRINT

LA-UR-11-00494

Approved for public release;
distribution is unlimited.

Title: A Report Documenting the Completion of the Los Alamos National Laboratory Portion of the ASC Level II Milestone "Visualization on the Supercomputing Platform"

Author(s):
113788 James Ahrens CCS-7
148176 John Patchett CCS-7
194699 Li-Ta Lo CCS-7
David DeMarle Kitware Inc.
Carson Brownlee University of Utah
229505 Christopher Mitchell CCS-7

Intended for: ASC Level II Milestone Meeting. August 13, 2010

Los Alamos NATIONAL LABORATORY EST.1943

Form 836 (7/06)

SAVE PRINT CLEAR FORM

A Report Documenting the Completion of the Los Alamos National Laboratory Portion of the ASC Level II Milestone "Visualization on the Supercomputing Platform"

James Ahrens, John Patchett, Li-Ta Lo, David DeMarle, Carson Brownlee, Christopher Mitchell

August 13, 2010

1 Introduction

This report provides documentation for the completion of the Los Alamos portion of the ASC Level II "Visualization on the Supercomputing Platform" milestone. This ASC Level II milestone is a joint milestone between Sandia National Laboratory and Los Alamos National Laboratory. The milestone text is shown in Figure 1 with the Los Alamos portions highlighted in boldfaced text.

Visualization and analysis of petascale data is limited by several factors which must be addressed as ACES delivers the Cielo platform. Two primary difficulties are:

1 **Performance of interactive rendering, which is the most computationally intensive portion of the visualization process. For terascale platforms, commodity clusters with graphics processors(GPUs) have been used for interactive rendering. For petascale platforms, visualization and rendering may be able to run efficiently on the supercomputer platform itself.**

2 I/O bandwidth, which limits how much information can be written to disk. If we simply analyze the sparse information that is saved to disk we miss the opportunity to analyze the rich information produced every timestep by the simulation. For the first issue, we are pursuing in-situ analysis, in which simulations are coupled directly with analysis libraries at runtime.

This milestone will evaluate the visualization and rendering performance of current and next generation supercomputers in contrast to GPU-based visualization clusters, and evaluate the perfromance of common analysis libraries coupled with the simulation that analyze and write data to disk during a running simulation. **This milestone will explore, evaluate and advance the maturity level of these technologies and their applicability to problems of interest to the ASC program.**

Figure 1: The ASC Level II Milestone "Visualization on the Supercomputing Platform"

This report is organized by the directives in the last sentence of the milestone text to **explore**, **evaluate** and **advance the maturity level of CPU-based rendering technology**. In section 2 of the report, we document our **advancement** of the CPU-rendering technology for scientific visualization, in section 3 we document our **evaluation** experiments, and in section 4 we document the **exploration** activities. In summary, we:

- Advanced the maturity level of CPU-based rendering technology improving speed 2-10 times over standard methods of CPU-based rendering.
- Evaluated the current CPU rendering against GPU based rendering and our improved CPU rendering.
- Explored possibilities of rendering on the Cell platform and methods for improving CPU rendering for production visualization.

2 Advanced the Maturity Level of CPU-based Rendering on the Supercomputing Platform

As part of this ASC Milestone we integrated the University of Utah's Manta Ray tracer into ParaView. The Manta ray-tracer is discussed in detail in section 3.1.4. The Manta Ray tracer plug-in is included in the ParaView 3.8.0 release. CPU rendering with Manta is now faster than it was with the Mesa 3-D graphics library, we have advanced the ASC programs ablity to render on supercomputing platforms with this effort.

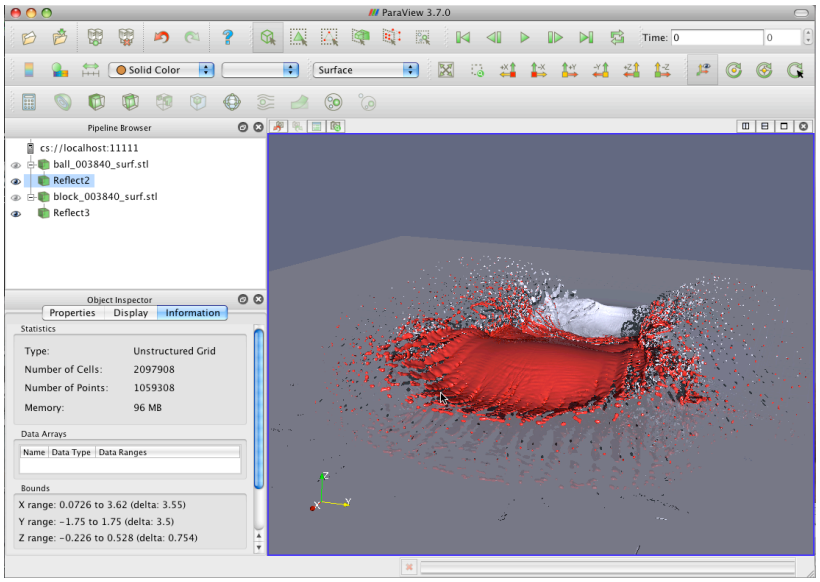


Figure 2: The Manta Ray tracer running in ParaView with reflections and shadows.

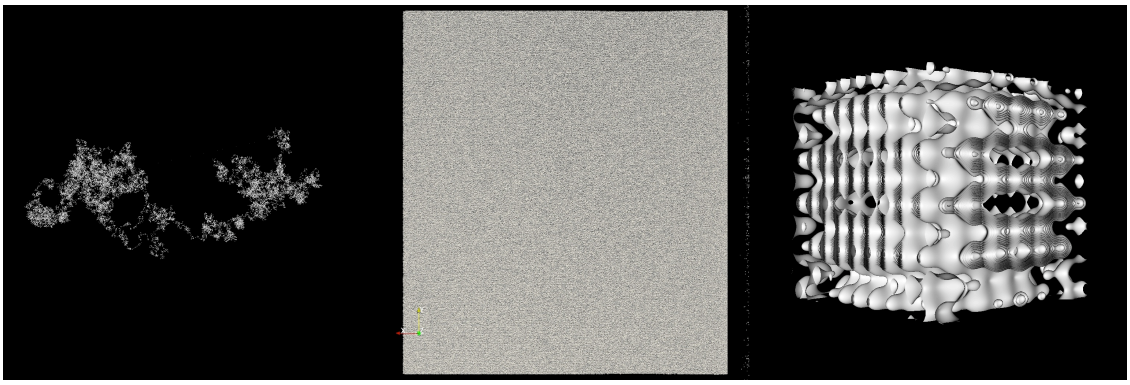
From the Release notes for ParaView 3.8, "ParaView now includes (in source form only) an interface to the University of Utah's Manta interactive software ray tracing engine. The Manta plugin provides a new 3D View type which uses Manta instead of OpenGL for rendering. The plugin is primarily being developed for visualization of large datasets on parallel machines. In single processor configuration it has the benefit of allowing realistic rendering effects such as shadows, translucency and reflection."

Figure 2 shows ParaView rendering with the Manta plug-in using shadows and reflections on a Sandia impact data set.

3 Evaluated the CPU and GPU-based Rendering Performance

3.1 Performance Evaluation Setup

In our evaluation we studied CPU and GPU-based rendering performance on two supercomputing platforms and a large multi-core server node: Lobo, a Los Alamos Tri-lab Linux compute cluster, Longhorn, a latest generation GPU-based visualization cluster at the Texas Advanced Computing Center and Kratos, a 32 core HP server. We used three datasets of varying sizes including randomly generated triangles, a synthetic wavelet dataset, and a dataset from Los Alamos's ASC plasma simulation code, VPIC. The rendering performance tests are run from within ParaView, the scalable open-source scientific visualization tool, designed and developed by Los Alamos, Sandia, Kitware and a number of other partners. Three different rendering packages were tested: Manta (an open-source raytracer), Mesa (a software implementation of OpenGL), and OpenGL using NVidia



(a) Random Triangles (b) VPIC Contours (c) Wavelet Contours

Figure 3: Eight million triangle version of the three test data sets. See section 3.1.2 for more details.

hardware.

3.1.1 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 ASC Computing users at Los Alamos, Livermore, and Sandia National Laboratories.

- **Longhorn** is a 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** is an HP Proliant DL785 G5 with 8-quadcore AMD Operton 8380 processors at 2.5 GHz with 128GB RAM. It is somewhat slower than individual nodes of Lobo, but it has 32 cores and much more RAM. This large machine allowed us to expand our testing to extremely large polygon counts.

3.1.2 Datasets

- **Random Triangles** We generated a test data set, originally to test Manta. It can easily and quickly produce large quantities of triangles for rendering. There are typically more triangles than can map to single pixel. An image showing a rendering of 8 million of these triangles is shown in Figure 3a.

- **VPIC visualization-generated Triangles** We collected a timestep of VPIC data from a Los Alamos simulation. We calculated two isosurfaces that produced 1, 2, 4, 8, and 16 million triangles for use in our evaluation. Though each set of triangles produce a different image

they are generally two parallel nearly planar surfaces. A view of this data set can be seen in in Figure 3b.

- **Wavelet Triangles** Wavelet is a computed synthetic data set source released with ParaView. We generated a 201^3 data set and then calculated as many isosurfaces as needed to produce a quantity of triangles. This produces a set of nested isosurfaces that could be considered as a best case for renderers (such as Manta) that use an occlusion/early ray-termination optimization since most of the triangles in the dataset are obscured by triangles in front of them. An image produced with 8 million of these triangles can be seen in Figure 3c.

3.1.3 Visualization Software

- **ParaView** We use ParaView (`http://www.paraview.org`) for a visualization research and development framework to test new algorithms and visualization paradigms. ParaView is an open source scalable visualization tool that is very modular in it's design. We used ParaView to explore methods of rendering on supercomputers.

3.1.4 Rendering Software

- **Manta** ray tracer is an interactive ray tracer from the University of Utah. It is portable and is distributed under an open source license. Over the past two decades, advances in both hardware performance and ray tracing implementations have made interactive ray tracing feasible. The Manta open source ray tracing engine is one of the most advanced, and flexible interactive ray tracers currently available. Manta is a mulithreaded application and library, in which processors independently trace different sets of pixels simultaneously. Within each thread, packets of rays are traced together to improve memory locality, and within packets rays are traced simultaneously using SIMD instructions to make use of intraprocessor parallelism. Manta is unique in that it is not only fast but also very flexible. The rendering engine can be scaled up in terms of image quality (at a cost of reduced interactivity), and it supports a large number of primitive types and rendering modalities (volume rendering, direct isosurface rendering, particle rendering). See `http://mantawiki.sci.utah.edu/manta/index.php/Main_Page` for more information on Manta.

- **OpenGL Hardware API** is a cross platform application programming interface (API) that is well supported by graphics hardware (GPUs). GPUs are considered the fastest method for rendering using the OpenGL API. OpenGL is typically used by today's graphics hardware. We used OpenGL on the Longhorn cluster to show current state of the art rendering capability. See `http://www.opengl.org/` for more information.

- **Mesa 3-D Graphics Library** is an open-source software implementation of the OpenGL API for use on general purpose CPUs. The Mesa 3-D Graphics Library is also released under an open source license. Mesa has been the defacto standard for OpenGL rendering when not using OpenGL supported graphics hardware. See `http://www.mesa3d.org/` for more information.

The size of the rendering window for all tests is 1024 by 1024.

3.2 Single Node Rendering Performance

Figures 4, 5, and 6 show a comparison of single node performance of the three rendering methods we evaluated: GPU on Longhorn, CPU on Lobo with Manta, and CPU on Lobo with Mesa. Figure 4 shows the performance of random triangles. We consider this the worst case performance due to the irregularity and the data from experimental results bear this out. The x-axis shows millions of triangles in the rendered scene, the y-axis show the average frame rate as the camera rotates around the scene in 3 degree increments. These 3 plots represent the best usage of a single node to maximize rendering performance with current methods. That is 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. All plots show decresing performance as the number of triangles increase and all plots show a convergence of Manta and GPU rendering at 16 million triangles. It is possible that we could have run 2 GPU's per Longhorn node, given the single CPU performance, this would extended that convergence to 32 million triangles, at best, since there would then have to be a buffer readback and a composite operation.

CPU rendering performance was improved by 2-10x, and at 16 million polygons rendering performance is equal to that of the GPU.

3.3 Compositing

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[2]. This is an efficient parallel compositing algorithm that exchanges portions of images between processes to produce a correctly rendered result. The IceT compositing library[4] is the default compositing scheme in ParaView. Although very efficient, IceT's performance is also very data dependent and IceT is integrated directly with the renderer. In order to clarify our rendering and compositing performance results we used the binary swap compositor. Figure 7 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 affect the compositing performance. We used ParaView to render empty scenes and step through the compositing process to document this baseline. Note that both machines settle near 20 frames per second with no rendering.

The GPU Longhorn cluster has a faster network than the CPU cluster Lobo. This difference will affect our rendering performance results.

3.4 Parallel Rendering

Parallel rendering occurs when the total polygons are divided amongst many processors and the results are composited[3]. Possible methods include sort-first where the polygons are sorted prior to rendering and distributed to processors that are responsible for a discrete piece of the display. Sort-last methods distribute the polygons to each processor which also computes a depth value, an entire full display image is rendered by each processor and then a compositing step produces the pixel closest to the camera from all parallel rendering nodes. We look at sort-last rendering

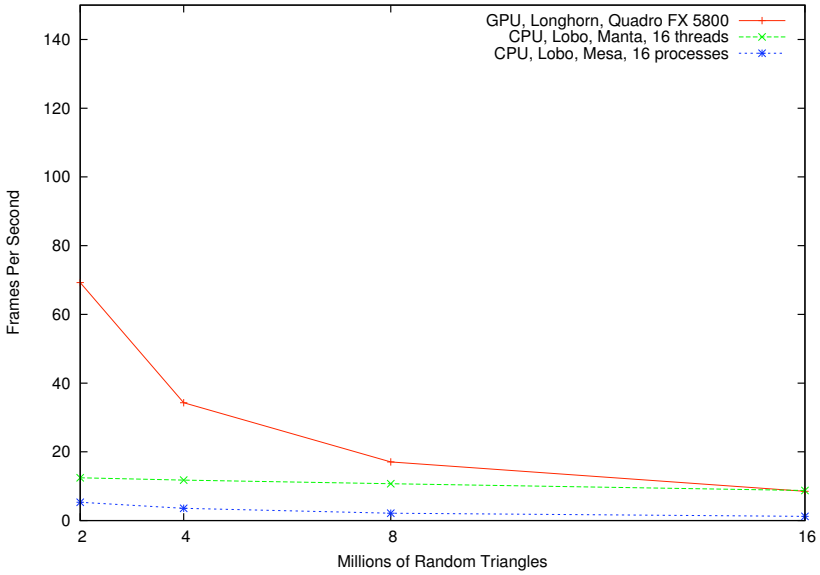


Figure 4: Single Node Performance on random triangles.

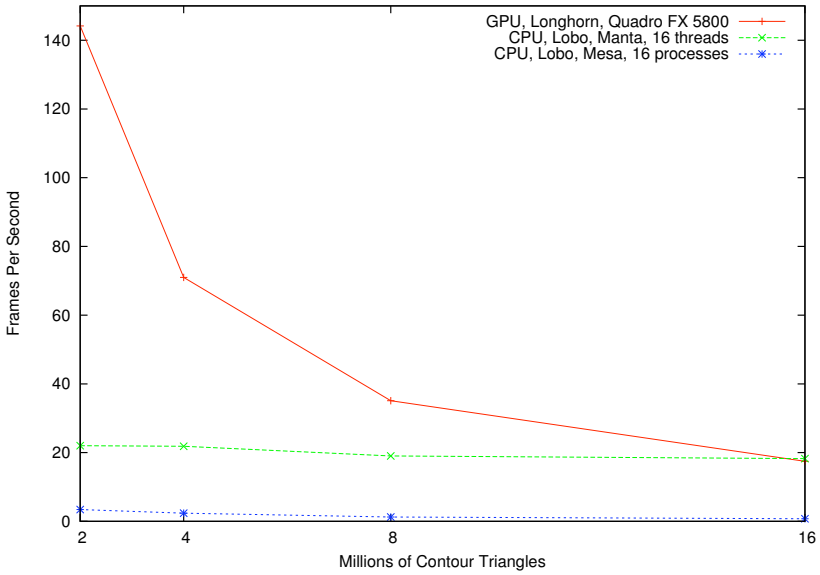


Figure 5: Single Node Performance on wavelet contours.

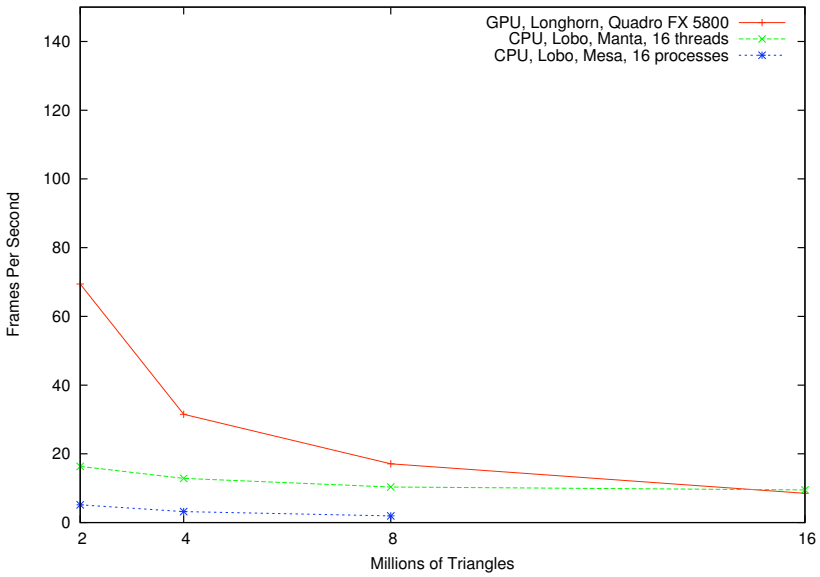


Figure 6: Single Node Performance on VPIC contours.

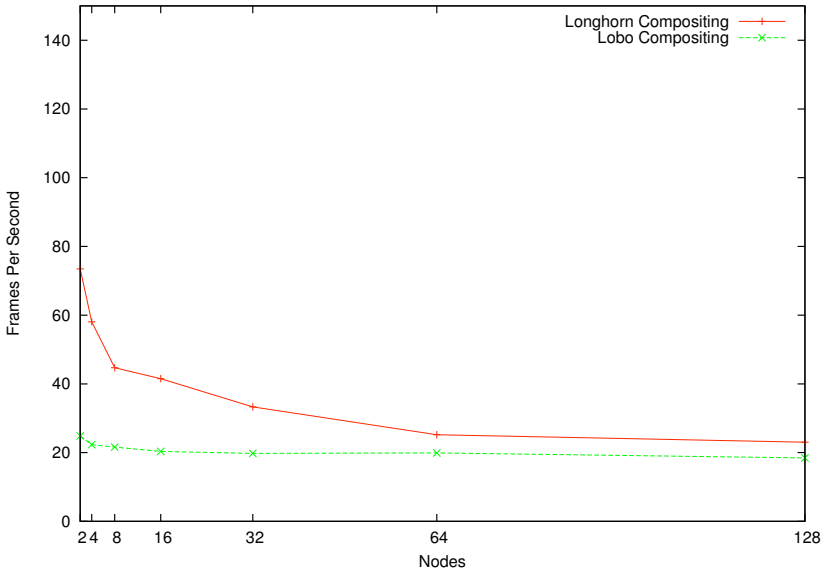


Figure 7: Compositing baseline for Lobo and Longhorn.

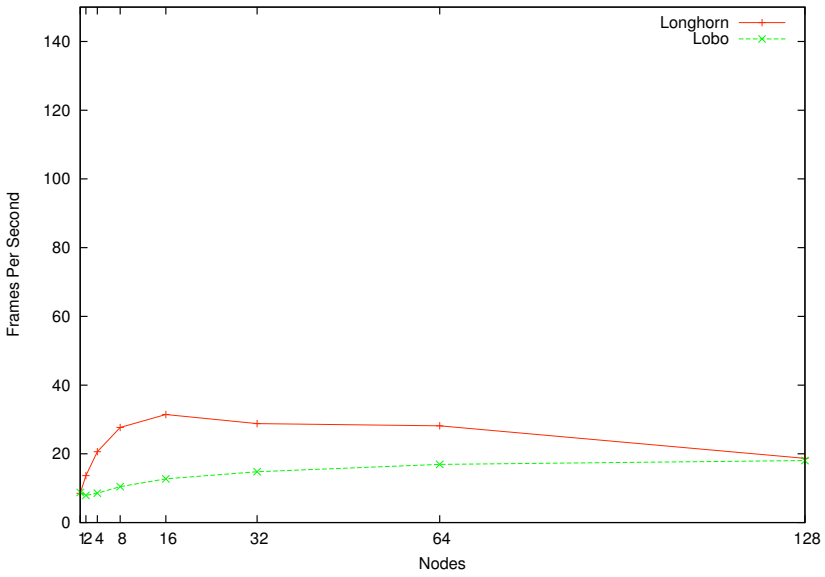


Figure 8: Strong scaling of 16 million random triangles.

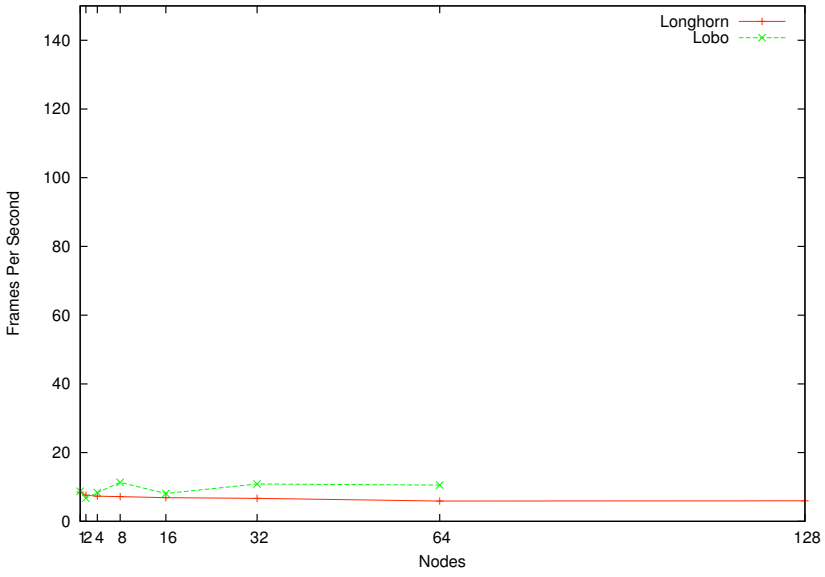


Figure 9: Weak scaling with 16 million random triangles per node.

and compositing where the total time to produce an image is the sum of the slowest processor's rendering and the time to composite ($\text{IMAGE}_{time} = \text{RENDER}_{time} + \text{COMPOSITE}_{time}$).

3.4.1 Strong Scaling

Strong scaling is the scaling attribute where the problem size remains constant while increasing processor resources to solve the problem. Figure 8 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. At the limit the rendering time would dissipate and the total time would equal compositing (IMAGE_{time}=COMPOSITE_{time}). This can be seen best on Longhorn which can render smaller polygon counts very quickly. Longhorn shows an initial increase in performance for going parallel, at 16 nodes it is becoming bound by the compositing and at 32 nodes it starts dropping performance. Lobo with CPU rendering, doesn't show the initial performance increase since the decreasing RENDER_{time} is similar to the increasing COMPOSITE_{time}. As both scale however, COMPOSITE_{time} becomes the clear upper bound.

For parallel rendering a fixed number of polygons, increasing the rendering resources binds maximum performance to the performance of compositing.

3.4.2 Weak Scaling

Weak scaling is the scaling attribute where the problem scales with the processor resources. For our testing we chose to show 16 million triangles per node. The total triangles for a data point in Figure 9 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 at scale we would expect the IMAGE_{time} to approach a constant for weak scaling. This can be seen in Figure 9. We rendered up to 2 billion polygons on Longhorn and 1 billion on Lobo.[1]

This weak scaling study shows the ability to render large quantities of polygons at similar rates with both CPU and GPU resources.

3.5 Massive Polygons

In order to test extremely large triangle counts on single nodes we compare Kratos and a single Longhorn node. Figure 10 summarizes our results. The x-axis shows polygon counts of our random triangles data set and again the y-axis shows frames per second. We were able to process 256 million triangles on Kratos. We were only able to process up to 128 million triangles on the Longhorn node. Longhorn's GPU rendering performance drops below and stays below CPU rendering on Kratos above 16 million polygons.

We ran tests for a variety of Manta thread counts, in order to utilize different numbers of Kratos CPU cores, these are summarized in the plot. The differences in performance between various thread counts would allow a fraction of a node to be used to **dial performance** to balance the use of a node with other tasks. For instance, a computational model might run on 24 processors,

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

while 8 processors rendered an image to represent time steps between full checkpoints. This balance could be dialed based on the computational needs of the model and the desired rendering.

CPU's could be the preferred rendering method in circumstances where polygon counts are extremely large. In addition, a node's CPU resources could be allocated to dial rendering performance to a needed level.

4 Explored the Use of CPU-based Rendering on the Supercomputing Platform

Our explorations primarily focus on general methods for improving the rendering performance on the supercomputing platform. Since our production visualization tool, Ensight, is not open source, we don't have the ability to alter the rendering engine to use a non OpenGL renderer like Manta. We are therefore investigating ways to improve the current software OpenGL by threading Mesa and by overloading OpenGL calls and translating them to the Manta framework at runtime with GluRay. We also are working to improve the rendering performance on the RoadRunner supercomputer.

4.1 Rendering on RoadRunner

We explored rendering using the Cell processor in the hopes that we could use the accelerators on the RoadRunner platform for fast supercomputing platform rendering performance. We explored two different approaches to software rendering on the Cell processors. The first one is based on a traditional rasterization pipeline. Triangles are first transformed from 3D world coordinates to 2D screen coordinates. The coverage of triangles vs. pixels are determined and the color and depth values of each pixel for each triangle (called fragments) are interpolated from their perspective values at triangle vertices. Fragments are then selected according to their depth value. The second approach is based on 3D raycasting. Triangles remain in 3D space without first transforming them to 2D space. We instead generate rays starting from camera positions through each pixel on the image plane and test ray triangle intersection. The intersection points of rays vs triangles are used to calculate color and depth values of the fragments. As in 2D rasterization, the final fragment value are selected according to the depth value. Figure 11 shows the performance of our current work.

There are some technical difficulties in parallelizing these two methods. Because the SPEs on the Cell processors have only 256KB of local store (LS), we can not do sort last parallel rendering by dividing triangles into groups, rendering the whole image by each SPE followed by a final image composite. We adopted a tile based method that divides the image into tiles small enough to fit the LS of each SPE. By assigning tiles to SPEs dynamically we can also improve load balancing among the SPEs.

We were unable to achieve fast rendering performance on the Cell processor for a large number of triangles. We believe that rendering is not an ideal problem for the Cell processor and the small local store on the Cell limited our performance.

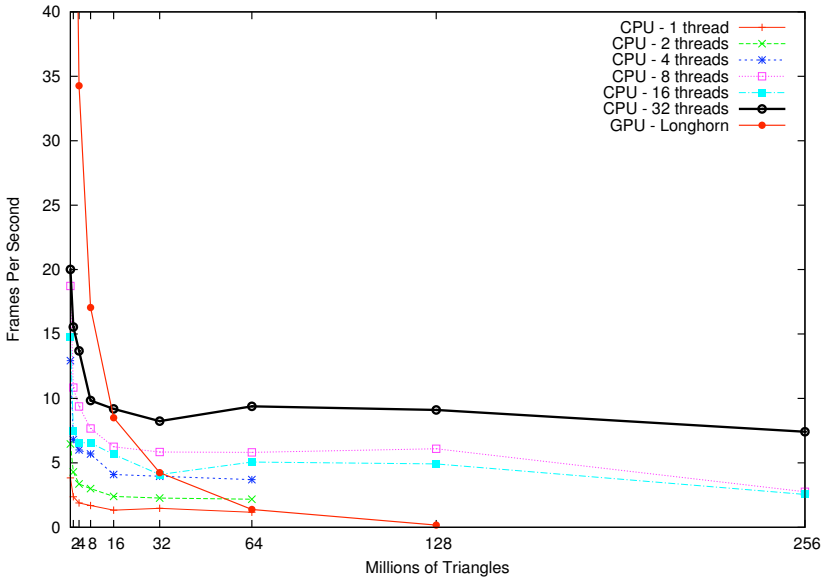


Figure 10: Rendering performance for massive numbers of triangles on Kratos and Longhorn.

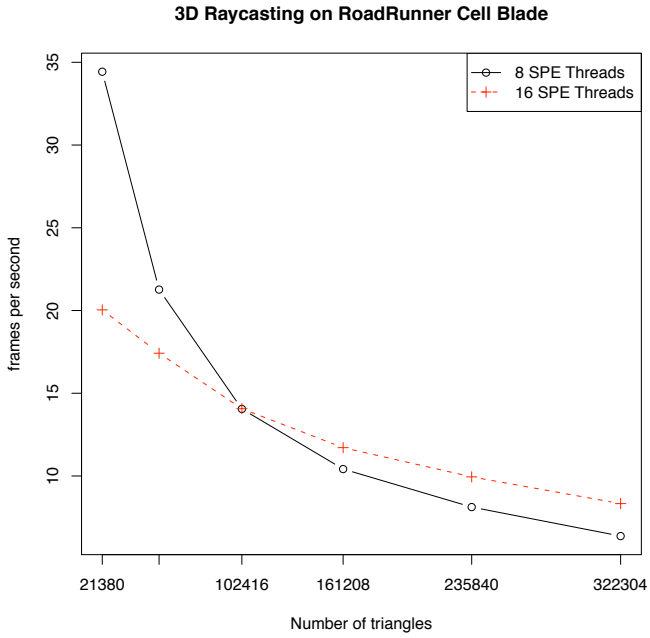


Figure 11: Single node 8 socket 32 core opteron machine.

4.2 An OpenGL API for the Manta Raytracing Engine

GLuRay is a program that intercepts OpenGL calls[2] and produces a ray traced image matching the OpenGL image which would otherwise be rendered. This method allows the utilization of traditional visualization packages such as Paraview, VisIt and Ensight without modification. The program works by making an equivalent change to the Manta ray tracer's scene for each OpenGL call. Figure 12 shows the use of GLURay with CEI's Ensight on a simple sample dataset. Manta rendering performance through the GLURay interface is very similar to the Manta performance presented in Section 3.2.

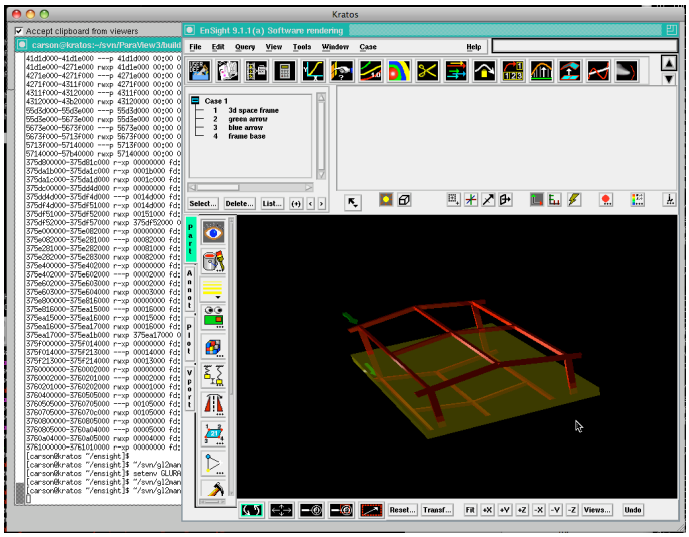


Figure 12: Using Ensight with GLURay OpenGL Rendering API

State changes such as material properties and color information are tracked and updated to produce identical images. Separate acceleration structures for ray tracing are built and updated when calls to OpenGL display lists or vertex arrays are made. These acceleration structures are updated asynchronously to the running application in a threaded manner resulting in minimal overhead when no new geometry is being produced. Buffer swaps are mapped to the ray tracer's rendering routine causing the display of the last rendered frame and starts the rendering of the next frame asynchronously. In addition GLuRay supports adding global effects such as ambient occlusion, accurate shadows, reflections, refraction and other high quality rendering features which can be enabled through an external GUI client. Currently GLuRay supports most basic polygonal representations in the visualization packages mentioned however some issues such as multiple viewports have not been implemented yet. Shader programs are also not implemented.

The GLURay OpenGL API is an experimental system (i.e. not production) but offers a viable approach to achieve fast CPU rendering on multi-core supercomputing platforms. If the ASC program is interested we recommend a discussion with CEI Inc. about options for incorporating fast CPU-based rendering in their product.

[2]GLURay is similar to the Chromium rendering API [1].

4.3 High Fidelity Rendering on Large Shared Memory Machines

Ray tracing is the standard approach to produce photo realistic images. The simplicity of the ray tracing algorithm makes it trivial to achieve realism. The algorithm casts a ray for each pixel and determines the color of the first hit object, then it identifies its adherence to geometric optics, and it takes into consideration the classical physics model of light transport (in which light rays bounce and bend at reflective and refractive objects). Shadows, reflections, translucency, depth of field, antialising and motion blur are all trivially computed in a ray tracer, by simply casting more rays.

We recommend interactive ray-tracing of additional datasets of interest to ASC program with shadows and reflections to evaluate the improvement of the understanding of 3D shapes. Large dataset sizes could be explored on shared memory machines, for example a server with up to a half a terabyte of memory can be obtained today with larger configurations available in the future.

5 Future Work

Since interactive rendering of approximately ten frames per second for massive triangle sized datasets is already possible using current supercomputers, we believe interactivity will improve on future supercomputers. Work that still needs to be done to accomodate visualization on the supercomputer for production use includes maximizing single node performance for other parts of the visualization pipeline like reading, isosurfacing, caculator operations and building acceleration structures. Also, overlapping rendering and compositing would cause parallel rendering to be bound by the greater time, either rendering or compositing, not the sum of the two as it is now.

6 Conclusion

In conclusion, we improved CPU-based rendering performance by a a factor of 2-10 times on our tests. In addition, we evaluated CPU and GPU-based rendering performance. We encourage production visualization experts to consider using CPU-based rendering solutions when it is appropriate. For example, on remote supercomputers CPU-based rendering can offer a means of viewing data without having to offload the data or geometry onto a GPU-based visualization system. In terms of comparative performance of the CPU and GPU we believe that further optimizations of the performance of both CPU or GPU-based rendering are possible. The simulation community is currently confronting this reality as they work to port their simulations to different hardware architectures. What is interesting about CPU rendering of massive datasets is that for past two decades GPU performance has significantly outperformed CPU-based systems. Based on our advancements, evaluations and explorations we believe that CPU-based rendering has returned as one viable option for the visualization of massive datasets.

References

[1] 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. *ACM Transactions on Graphics*, 21(3):693–702, 2002.

[2] 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.

[3] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics Applications*, 14(4):23–32, 1994.

[4] Kenneth Moreland, Brian Wylie, and Constantine 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.