Eurographics Symposium on Parallel Graphics and Visualization (2013)
F. Marton and K. Moreland (Editors)

# Image-parallel Ray Tracing using OpenGL Interception

Carson Brownlee[1,2], Thiago Ize[3], and Charles D. Hansen[1,2]

[1]School of Computing, University of Utah
[2]SCI Institute, University of Utah
[3]Solid Angle

**Abstract**

*CPU Ray tracing in scientific visualization has been shown to be an efficient rendering algorithm for large-scale polygonal data on distributed-memory systems by using custom integrations which modify the source code of existing visualization tools or by using OpenGL interception to run without source code modification to existing tools. Previous implementations in common visualization tools use existing data-parallel work distribution with sort-last compositing algorithms and exhibited sub-optimal performance scaling across multiple nodes due to the inefficiencies of data-parallel distributions of the scene geometry. This paper presents a solution which uses efficient ray tracing through OpenGL interception using an image-parallel work distribution implemented on top of the data-parallel distribution of the host program while supporting a paging system for access to non-resident data. Through a series of scaling studies, we show that using an image-parallel distribution often provides superior scaling performance which is more independent of the data distribution and view, while also supporting secondary rays for advanced rendering effects.*

## 1. Introduction

Parallel ray tracing has proven to be an efficient algorithm for rendering large-scale polygonal models which scales well on multi-core and multi-node machines due to it's embarrassingly parallel nature [WSB01]. The most popular scientific visualization tools often rely on brute-force OpenGL rasterization with no frustum or occlusion culling performed. Due to the brute-force algorithms used, these programs often scale poorly with increasing polygon counts and often do not support advanced rendering effects such as ambient occlusion which can provide publication-quality images and enhanced perception of information from additional depth cues. Recent implementations of CPU ray tracing within common visualization tools such as ParaView and VisIt [Kit10, LLN10] only support rendering of secondary rays on shared-memory systems and suffer from sub-optimal scaling when using primary rays due to the data-parallel data distribution utilized [BPL*12]. These custom renderers must also be updated with changes to the visualization tools, often requiring considerable programmer effort with each change to the visualization tool. Brownlee et al. introduced GLuRay, which used OpenGL interception for ray tracing in visualization tools to avoid source code modifications [BFH12]. Using a similar OpenGL interception technique as Brownlee et al., we use a scalable image-parallel work distribution model based on the work of Ize et al [IBH11] for better scaling on distributed-memory systems. Non-resident data regions are loaded either through data replication on each node or through a paging algorithm for accessing remote bricks of data. This implementation also enables rendering of secondary rays on distributed-memory systems which was not possible with a static data-parallel distribution of the scene. In this paper we describe previous work in massive model ray tracing in Section 2. The implementation of our system is given in Section 3 describing the OpenGL interception, image-parallel work distribution, and data paging using remote reads. Section 4 describes a series of strong scaling studies conducted on a distributed-memory cluster comparing our system with ray tracing using static data-parallel work distribution, GPU rasterization, and CPU rasterization. Future work, conclusions, and acknowledgements are described in Sections 5, 6, and 7.

## 2. Related Work

Many commonly used visualization tools have been built to handle parallel data analysis and visualization on large distributed-memory systems such as the open-source tool ParaView [Kit10]. These programs utilize a client/server architecture with a single client using multiple server nodes to load and process data remotely. Work is commonly sent to server nodes in a data-parallel distribution where each node is responsible for a subregion of the overall data space [CGAF06]. Nodes then load in their subregion for analysis and rendering, and the resulting images are then composited together using depth information to compose an image of the entire scene in a sort-last compositing step. By using image-parallel distribution, a sort-last compositing step is spurious, but still present in the existing program and simply not timed in our results. In ParaView, OpenGL calls are rendered either with hardware-acceleration or using a software renderer such as Mesa, which is a single-threaded build option for software rendering. Mitra and Chiueh [MC98] developed a parallel Mesa implementation by running multiple instances of Mesa through a serial interface which required a compositing operation for each instance of Mesa. Nouanesengsy et al. [NAW11] explored performance of such as setup on large shared-memory machines using different compositing methods. By utilizing a hybrid sort-first and sort-last compositing step they could achieve nearly linear speedups with the number of cores on a machine; however, running multiple instances of ParaView by spawning additional MPI processes failed to scale well. Howison et al. [HBC10] demonstrated that running a MPI hybrid setup with one MPI process running multiple threads on a single node provided superior performance. Therefore, we run all of our tests with a single running ParaView process per node and use hybrid-parallelism with multiple threads per process for our renderer.

Significant work has been done to show that ray tracing presents an embarrassingly parallel algorithm which scales efficiently on large multi-node and multi-core machines [BSP06, PSL*98, PPL*99]. The Manta real-time ray tracing software has proven scaling performance on shared-memory machines and scales well with large geometry counts [SBB*06]. DeMarle et al. described an image-parallel work distribution system where each node stores a portion of the scene, allowing the scene size to grow to a fraction of the aggregate memory available across the cluster [DGP04]. Ize et al. [IBH11] developed a distributed-memory version of the Manta ray tracer with a paging system which achieved efficient performance by using a cache aware, traversal ordered BVH acceleration structures [WBS07] on multi-node machines. This improves upon DeMarle et al.'s implementation which used a single core for rendering on each node and a less efficient macro-cell grid for acceleration. Hybrid data-parallel and image-parallel techniques have been developed, however we use a caching scheme which exploits frame to frame coherence of locally cached data with image-parallel distribution [RCJ99].

Navratil et. al developed a non-interactive distributed-memory ray tracer in VisIt for rendering large-scale, out-of-core data to demonstrate the importance of ray scheduling when disk reads are needed [NFLC12], however we focus on interactive rendering of data that fits into the large aggregate memory spaces of supercomputing clusters. Brownlee et al. [BPL*12] developed a ray tracing framework integrated into VTK and subsequently ParaView and VisIt which could use ray tracing, but relied on the data distribution and compositing present in VTK. GLuRay was developed by Brownlee et al. [BFH12] as a ray tracing framework using OpenGL interception on top of scientific visualization programs. They reported increased render times of over 300x native rendering performance using CPU rasterization on a single node, however their implementation was burdened by the use of the static data-parallel scene distribution provided by the host application which resulted in poor scaling performance dependent on the current view used and their implementation only supported rendering secondary rays on shared-memory systems. We present an OpenGL interception library using ray tracing which uses an image-parallel work distribution and a distributed paging scheme with a local cache on each node for storing non-resident data. This implementation demonstrates superior scaling performance on distributed-memory systems and enables for ray tracing secondary rays across the entire scene.

## 3. Implementation

We present a system which extends the OpenGL interception techniques presented by Brownlee et al. [BFH12] while using an image-parallel distribution based on the work by Ize et al [IBH11].

### 3.1. Intercepting OpenGL Calls

We created an OpenGL implementation based on the official OpenGL specification which maps OpenGL API calls to ray tracing calls similar to GLuRay [BFH12]. Each ParaView process launches multiple rendering threads in the background for ray tracing which are synchronized with the main thread when rendering and display are needed. Many OpenGL calls are directly mapped to the ray tracer including transform matrices, material parameters, light properties, geometry specification, and rendering attributes such as flat or smooth shading. Groups of geometry are built with each display list created with material parameters, lighting information, and shading specifications. Geometry is added to the scene upon calls to glCallList, instantiating the geometry with the current ModelView transform. Some function calls are passed through to an existing OpenGL implementation such as glDrawPixels or glXSwapBuffers. Multiple renderings per frame are avoided. For our runs with ParaView, rendering was started with calls to glFlush for single process programs or glReadPixels for runs over multiple nodes.

### 3.2. Ray Tracing using Data-parallel Distribution

ParaView uses a data-parallel work distribution across distributed-memory machines. Each node is responsible for loading, filtering, and rendering a subregion of the data. Rendering is conducted on the data local to each node with no dynamic data distribution across nodes. To generate an image of the entire scene, each node distributes its rendered image with depth information which are composited together to determine the nearest pixel from each image to produce an accurate image of the entire scene. This same compositing and distribution scheme can be used with the ray tracer by writing the rendered image and tracking depth information of each nearest hit into a depth buffer which is then sent to the framebuffer upon a call to glReadPixels. This implementation has no dynamic distribution of data meaning that only primary rays can be rendered as each node only has geometry pertaining to its portion of the scene.

### 3.3. Ray Tracing using Image-parallel Distribution

For this paper we have implemented an image-parallel work distribution for rendering on top of a host program's data-parallel data distribution using both replicated data and dynamic data distribution using a paging scheme developed by Ize et al. [IBH11]. This scheme allows for efficient scaling of data loading and filtering within scientific visualization programs while enabling rendering of secondary rays and efficient scaling of primary rays compared to static scene distributions. Accessing non-resident geometry not sent from the host program is done through both data replication over MPI and through paging in portions of data on demand by nodes requesting blocks of data. For replicated data, each node broadcasts resident portions of the data across all rendering nodes according to display lists created locally which are synchronized with unique ids corresponding to local display list ids. For each frame, transforms for each list instanced locally are broadcast across all rendering nodes. A master process is responsible for distributing tiles of the image for rendering as each rendering node requests work and resulting pixels are sent to a display processes to produce a final result. Rendering is conducted when all nodes make a call to glReadPixels and the final image is sent to the framebuffer when node 0 renders a screen aligned quad for display.

Each frame rendering requires cross-node synchronization with MPI calls across multiple threads per node with a single running MPI process per node in a hybrid-parallel system. Since the host program initializes MPI, we override MPI_Init in order to enable multithreaded MPI with MPI_THREAD_MULTIPLE. Running OpenGL API calls with MPI in the background has the potential to break synchronized MPI communications from a host program and as such we only make MPI calls for OpenGL function calls which are made at the same time across all nodes with no waiting MPI calls on any one node. For ParaView, glReadPixels provided such a synchronization point for our render nodes.

### 3.4. Load Balancing and Display

Manta uses a dynamic work queue on each node where each node requests tiles from a master process which are then split into smaller tiles and distributed to rendering threads as requested. The two level load balancer limits cross-node communication while efficiently balancing work on a per-thread basis on each node. A separate display process receives rendered pixels from each render node and copies them into a buffer which holds the final image. One thread on the display process is responsible for all MPI communication, receiving pixels from render nodes. Three additional threads are used for copying the rendered pixels to the image buffer. Unlike the data-parallel distribution, no compositing of pixel data using depth buffer information is required making the image compositing present in the host program spurious. The IceT compositing library utilized by ParaView uses a call to glReadPixels to grab the rendered framebuffer on each node which are then sent to other nodes for sort-last compositing. Unnecessary pixel transfer by the host program may be mitigated by sending an empty scene to each call of glReadPixels as IceT has the option to only distribute portions of the rendered image which contain pixels rendered into after a clear with the background color; however, such implementations were not tested in this paper.

### 3.5. Distributed-memory Paging

Our distributed-memory paging implementation uses the same paging algorithm proposed by Ize et al. [IBH11]. Each node is responsible for a portion of the overall scene data while maintaining an explicit cache to store non-resident data from other nodes. Paging is useful when the total scene size does not fit on a single node, but fits into a cluster's aggregate memory space.

Ray tracing enables a straightforward calculation of advanced rendering effects such as depth of field, ambient-occlusion, shadows, transparency, refraction, and reflections [CPC84, Whi80]. Studies have shown that ambient-occlusion provides depth cues which can aid in understanding data's spatial locality compared to using only local lighting such as that provided through the OpenGL fixed-function pipeline [GP06]. Manta supports path tracing to compute indirect illumination, however we do not utilize path tracing in our implementation to instead focus on interactive rendering. Rendering properties and material properties not defined in the OpenGL specification are sent to the ray tracer through configuration files or an external GUI application communicating over localhost. Changes are made globally to all objects of the scene as there is no method for modifying individual scene objects.

## 4. Results

We studied the performance of our rendering system compared to hardware-accelerated OpenGL, software rasterization using Mesa, CPU ray tracing using GLuRay [BFH12], and CPU ray tracing using our implementation. We ran our tests with the open-source visualization tool ParaView 3.14.1 using 1-62 rendering nodes 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 running MVAPICH2, each with 2 Intel Nehalem quad core CPUs (model E5540) at 2.53 GHz and 48-144 GB of RAM. Each node of Longhorn also has 2 NVidia FX 5800 GPUs, however only a single GPU per node is utilized in our runs. Our datasets consist of two datasets with a zoomed out and zoomed in view. Using a singe time-step from Los Alamos' VPIC plasma simulation, we calculated an isosurface and extracted streamtubes that combined totaled 102 million polygons. A view of this dataset rendered in ParaView can be seen in Figure 1(a). Ambient occlusion was used which shades polygons based on proximity to other geometry as seen in Figure 1(b). An enhanced rendering seen in Figure 1(c) shows the same dataset rendered with 36 ambient occlusion rays, 1 shadow ray, 1 reflection ray, and 4 samples per pixel. A zoomed in view was also used to look at scaling performance with a caching scheme as seen in Figures 2(a), 2(b), and 2(c). A time step from a Richtmyer-Meshkov Instability (RM) simulation at Lawrence Livermore National Laboratory was also used in our tests. We created a polygonal representation with an isosurface from time-step 273 resulting in 123 million triangles as seen in Figures 3(a), 3(b), 3(c), 4(a), 4(b), and 4(c). We use two different views of the RM and VPIC datasets to test performance scaling characteristics affected by camera views. Image resolutions scale from 720p up to 2160p. ParaView was run with offscreen rendering in batch mode in our tests with a warm up period followed by averaged timing information collected from IceT's render and buffer read timings. For the data-parallel GLuRay implementation and our image-parallel implementation, timing information was taken from buffer readings instead of render timings since rendering was conducted at calls to glReadPixels in the buffer reading stage of the IceT compositor. This comparison is not a comparison of the ultimate potential of rasterization vs. ray tracing algorithms, but rather a real-world study of their performance in a commonly used tool.

Figure 5(a)† shows the 102 million triangle VPIC dataset in a strong scaling study from 1-62 nodes using our method, referred to as GLuDRay, with replicated data on each node, GPU accelerated rasterization, GLuRay software ray tracing, and software rasterization using Mesa rendered at 720x1280 resolution. The x and y axis are in logarithmic scale giving

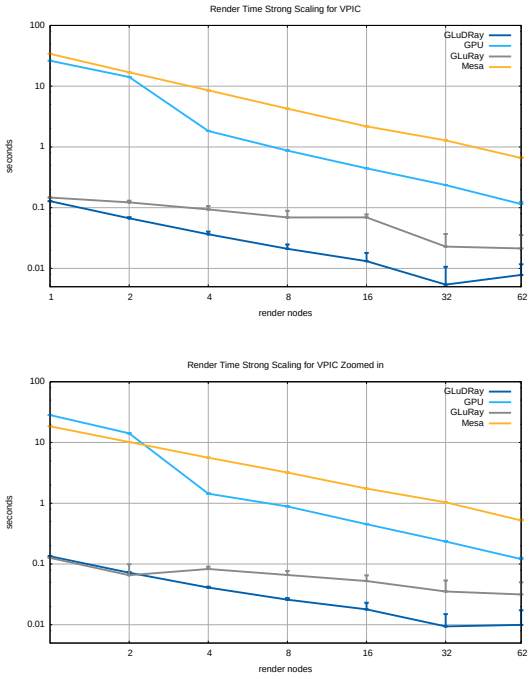



**Figure 5:** *Node scaling study of the VPIC plasma simulation rendered zoomed out (top), and zoomed in (bottom).*

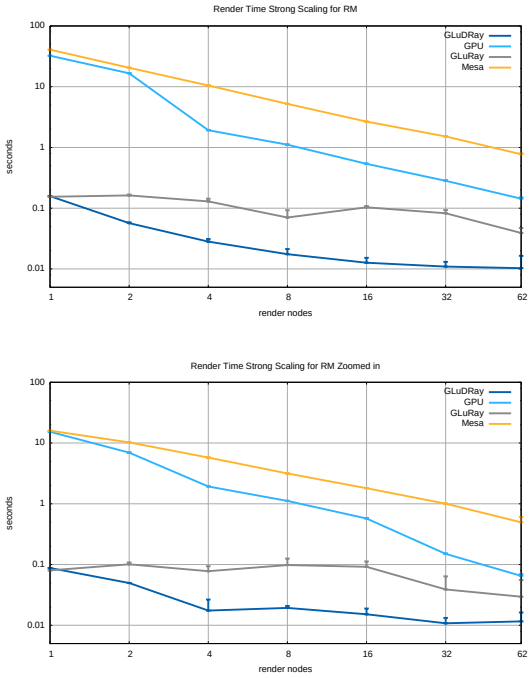
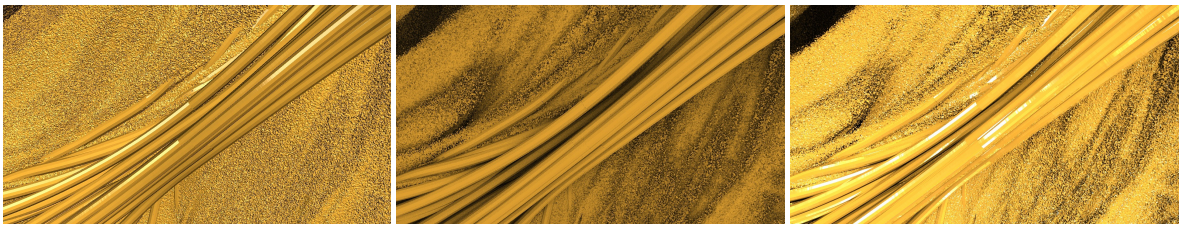



**Figure 6:** *The RM dataset rended with a zoomed out view (top) and a zoomed in view (bottom).*

† Tables with numeric data are provided for all graphs in the supplementary material in the digital library

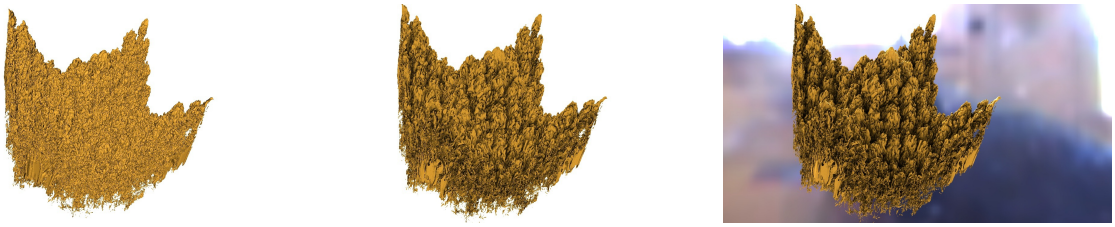
(a) (b) (c)

**Figure 1:** *Renderings of a VPIC plasma simulation rendered with basic local illumination (a); 36 ambient-occlusion rays (b); and 36 ambient-occlusion rays, 1 shadow ray, 1 reflection ray, and 4 samples per pixel (c).*



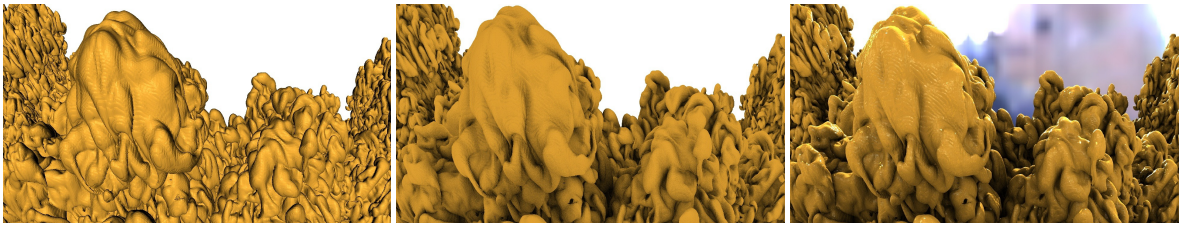
(a) (b) (c)

**Figure 2:** *Renderings of a VPIC plasma simulation zoomed in to a small portion of the data with local lighting (a); ambient-occlusion (b); and ambient-occlusion, shadow, and reflections (c).*



(a) (b) (c)

**Figure 3:** *Renderings of a Richtmyer-Meshkov Instability with local lighting (a); ambient-occlusion (b); and ambient-occlusion, shadow, and reflections (c).*



(a) (b) (c)

**Figure 4:** *Renderings of a Richtmyer-Meshkov Instability zoomed into a small portion of the data with local lighting (a); ambient-occlusion (b); and ambient-occlusion, shadow, and reflections (c).*

the number of seconds IceT reported for rendering or in the case of the interception methods, reading back buffers which triggered our ray tracer to render. Lines represent the average render times across all nodes, while the vertical bar above each data point shows the maximum rendering times across all nodes giving the frame time without compositing and display time. Mesa and GPU renderings take over 26 seconds on a single frame and don't drop below 1 second for rendering time until 8 and 62 nodes respectively. There is a jump for GPU performance at 4 nodes as display lists crashed with 1 and 2 nodes. Both Mesa and the GPU scale very well with node counts, showing that performance is highly correlated with the number of triangles per-node in the scene. At best, the GPU takes 0.12 seconds to render a frame with 62 nodes, giving an average triangle count per-node at roughly 1.6M triangles per node. GLuRay performance in our tests took 0.15 seconds to render the same dataset on a single node, a speed which Mesa never obtains and the GPU did not surpass until 62 render nodes were used. Scaling performance with GLuRay is sub-linear with respect to increasing numbers of nodes used for rendering. With 62 nodes, GLuRay achieves rendering speeds of 0.035 seconds giving a scaling efficiency of only 7%. Our implementation peaks at 32 nodes when rendering primary rays only with a maximum render time of 0.011 seconds as nodes become starved for work, which is a scaling efficiency of 36% and is likely limited by pixel and work distribution over the network. Performance gains drop off after 32 nodes, however optimizing beyond frame times equivalent to nearly 100 fps is not a concern as it would provide little perceptible user benefit since monitor refresh rates are typically 60Hz. If compositing is eliminated or reduced in a later stage by sending a blank framebuffer, overall frame time may be further reduced compared to the other methods presented but this was not explored. While our method scales better with rendering nodes, two additional MPI processes are needed in the current implementation for display and load balancing. As Ize et al. [IBH11] demonstrated, a single multi-core node could be used for both; however, for our tests two separate nodes were used for display and load balancing which are not shown in the graphs. These processes could also be run on the same node as a rendering node, however this performance difference was not evaluated in theses tests and instead we focus purely on scaling the number of render nodes used solely for rendering during the rendering stage of ParaView.

Figure 5(b) shows a strong scaling study of the VPIC dataset with a zoomed in view. GPU performance is largely unaffected in this case, although Mesa performance is increased compared to the zoomed out view as triangles are clipped out and don't need to be rasterized to the framebuffer. GLuRay rendering times are 0.015 seconds slower than the zoomed out view which achieves better scaling results, while our version is only 0.004 seconds slower. This minor decrease in maximum rendering time in our method is likely due to the increased pixel coverage by the geometry as empty space

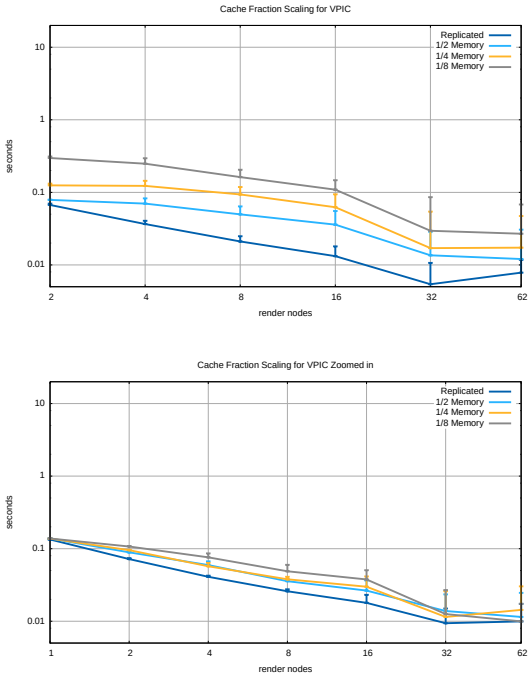



**Figure 7:** *Cache fraction scaling study of the VPIC plasma simulation rendered zoomed out (top), and zoomed in (bottom).*

surrounding the dataset is efficiently skipped over with a bounding box test. Figures 6(a) and 6(b) display a similar behavior with the RM datasets zoomed out and in. Our implementation exhibits better scaling performance than GLuRay which does not scale with the RM zoomed in view until up to 32 nodes when visible data is split significantly.

Figures 7(a) and 7(b) show strong scaling render times with a resident cache on each node for the VPIC datasets zoomed out and in respectively. Runs using replicated data do not use a cache but merely replicate all geometry on each node. When the scene is zoomed out, smaller cache sizes perform significantly worse than replicating data on each node. When a zoomed in view of the dataset is used however, using a small cache becomes less of a strain on performance as the visible subregion of the overall dataset is smaller. Figures 8(a) and 8(b) display render timings in a strong scaling study of the VPIC dataset zoomed out and zoomed in with 36 ambient occlusion rays per hit pixel. Not only are render times significantly longer, but more regions of the scene must be accessed beyond what is visible by primary rays alone. This results in less significant differences between the zoomed out and zoomed in views of the dataset and greatly increased ranges in both graphs between replicated and cached data. Ambient occlusion also resulted in fewer cache misses as a fraction of the total number of data accesses. Cache misses

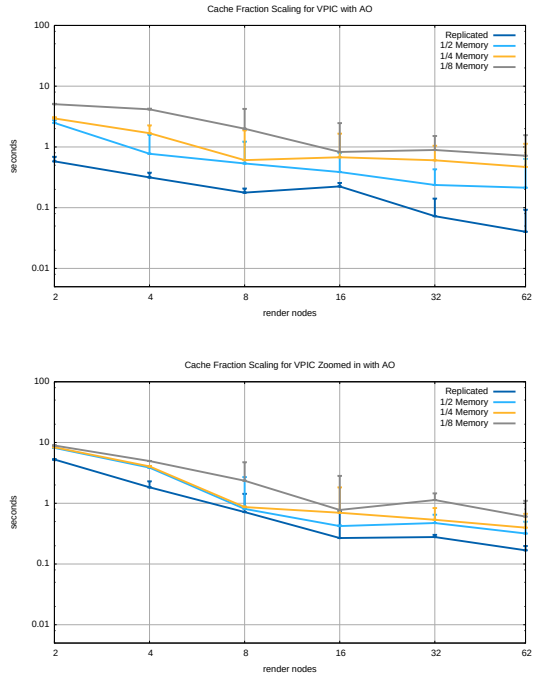



**Figure 8:** *VPIC plasma simulation rendered with ambient occlusion (top) and zoomed in (bottom).*

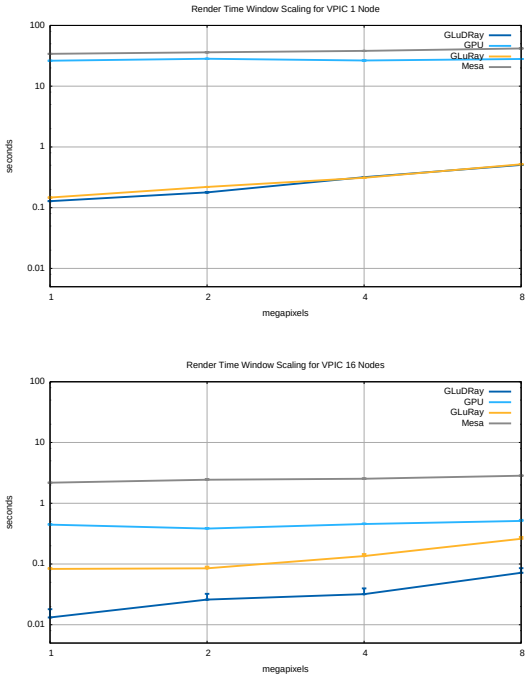



**Figure 9:** *Window size scaling study of the VPIC plasma simulation rendered from 1 to 8 megapixels with a single node (top) and 16 nodes (bottom).*

at 62 render nodes with VPIC zoomed in with primary rays ranged from 0.6% of total data accesses with a cache size of 1/2 the total data size to 1.1% with a 1/8 cache fraction. With ambient occlusion, cache misses decreased to 0.2% to 0.8% for 1/2 and 1/8 cache fractions respectively. Render times continue to scale up to 62 render nodes when more work is introduced to be distributed, such as with ambient occlusion rays. For the advanced renderings in Figures 1(c), 2(c), 3(c), and 4(c), additional secondary rays were used with 36 ambient occlusion rays, 1 shadow ray, 1 reflection ray, and 4 samples per pixel for a total of roughly 152 rays for each hit pixel in the scene not counting multiple reflections. For the VPIC dataset zoomed out and replicated data with advanced rendering effects, maximum render times scaled from 57.96 seconds on a single render node to 1.72 seconds on 62 render nodes which is not shown in the graphs.

Figures 9(a) and 9(b) show results of scaling window sizes on a single node and 16 nodes respectively with the VPIC dataset zoomed out. Window sizes were 720p, 1080p, 1440p, and 2160p resulting in roughly 1, 2, 4, and 8 megapixels. In the 16 node runs, GPU performance scaled from 0.44 seconds at 1MP to 0.51 seconds at 8MP with Mesa scaling similarly. As can be expected, the ray tracing performance of our system decreased roughly linearly with window size scaling from 0.013 seconds with 1MP down to 0.0717 seconds with 8MP and GLuRay demonstrated a similar drop in performance.

## 5. Future Work

Future work to achieve better performance using smaller cache sizes on each node relative to aggregate data sizes would be beneficial. Higher performance could be achieved with smaller cache sizes by better scheduling work tiles from the master node to server nodes according to previous frames to better exploit frame to frame coherence. Lossless compression could be used for geometry and pixel transfers to reduce data sizes. Hierarchical level of detail would allow for significantly reduced data transfer times. Voxelizing data similar to the GigaVoxels system by Crassin et al. [CNLE09] for simplifying sub-pixel regions of data could greatly reduce data transfer times with little or no discernible reduction in image quality. For secondary rays, many effects such as ambient occlusion are low-frequency and lower levels of detail can be used while paging in higher levels for sharp shadows or glossy reflections as required. We currently only support the OpenGL fixed-function pipeline. Supporting shaders would be a significant development effort but extend the implementation to programs which use shaders. In this paper we have not explored compositing or display time. Sort-last compositing is unnecessary with our implementation and if it could be eliminated or reduced then significant performance gains could likely be achieved.

## 6. Conclusion

In this paper we have shown that image-parallel data distribution conducted in the background of running visualization tools using OpenGL interception results in real-time rendering rates for large-scale data while also enabling tracing secondary rays for advanced rendering effects on distributed-memory systems. Our implementation attained significantly improved scaling performance over previous ray tracing implementations built into common visualization tools or used through OpenGL interception using data-parallel geometry distribution for rendering. We have demonstrated rendering rates approaching 100 fps without compositing and display with datasets over 100 million triangles on the visualization cluster Longhorn when GPU accelerated rendering rates did not achieve interactive performance and Mesa software rendering failed to render a frame in under 1 second. With this work, we have shown that CPU ray tracing in a widely used visualization tool using image-parallel work distribution enables real-time and high-quality rendering of large-scale data without modification to the underlying program.

## 7. Acknowledgements

We would like to thank Paul Debevec for providing light probes used in this paper. This research was sponsored by awards KUS-C1-016-04 made by King Abdullah University of Science and Technology (KAUST), DOE SciDAC Institute of Scalable Data Management Analysis and Visualization DOE DE-SC0007446, NSF OCI-0906379, NSF IIS-1162013, NIH-1R01GM098151-01.

## References

[BFH12] BROWNLEE C., FOGAL T., HANSEN C.: GLuRay: Enhanced ray tracing in existing scientific visualization applications using opengl interception. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012), The Eurographics Association, pp. 41–50. 1, 2, 4

[BPL*12] BROWNLEE C., PATCHETT J., LO L., DEMARLE D., MITCHELL C., AHRENS J., HANSEN C.: A study of ray tracing large-scale scientific data in two widely used parallel visualization applications. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012), The Eurographics Association, pp. 51–60. 1, 2

[BSP06] BIGLER J., STEPHENS A., PARKER S.: Design for parallel interactive ray tracing systems. In *Interactive Ray Tracing 2006, IEEE Symposium on* (sept. 2006), pp. 187 –196. 2

[CGAF06] CEDILNIK A., GEVECI B., AHRENS J., FAVRE J.: Remote large data visualization in the paraview framework. *Eurographics Symposium on Parallel Graphics and Visualization* (2006), 162–170. 2

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games* (2009), I3D '09, pp. 15–22. 7

[CPC84] COOK R., PORTER T., CARPENTER L.: Distributed ray tracing. *Computer Graphics (Proceeding of SIGGRAPH 84) 18*, 3 (1984), 137–144. 3

[DGP04] DEMARLE D. E., GRIBBLE C., PARKER S.: Memory-savvy distributed interactive ray tracing. In *Proc. of Eurographics Symposium on Parallel Graphics and Visualization* (2004), pp. 93–100. 2

[GP06] GRIBBLE C. P., PARKER S. G.: Enhancing interactive particle visualization with advanced shading models. In *Proceedings of the 3rd symposium on Applied perception in graphics and visualization* (New York, NY, USA, 2006), APGV '06, ACM, pp. 111–118. URL: http://doi.acm.org/10.1145/1140491.1140514, doi:http://doi.acm.org/10.1145/1140491.1140514. 3

[HBC10] HOWISON M., BETHEL E., CHILDS H.: MPI-hybrid parallelism for volume rendering on large, multi-core systems. In *Eurographics Symposium on Parallel Graphics and Visualization (EGPGV)* (May 2010). 2

[IBH11] IZE T., BROWNLEE C., HANSEN C. D.: Real-time ray tracer for visualizing massive models on a cluster. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization* (2011), pp. 61–69. 1, 2, 3, 6

[Kit10] KITWARE: *Paraview - Open Source Scientific Visualization*, 2010. http://www.paraview.org/. 1, 2

[LLN10] LLNL: *VisIt Visualization Tool*, 2010. https://wci.llnl.gov/codes/visit/. 1

[MC98] MITRA T., CHIUEH T. C.: Implementation and evaluation of the parallel mesa library. In *Parallel and Distributed Systems, 1998. Proceedings.* (dec 1998), pp. 84–91. 2

[NAW11] NOUANESENGSY B., AHRENS J., WOODRING J.: Revisiting parallel rendering for shared memory machines. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization* (2011), pp. 31–40. 2

[NFLC12] NAVRATIL P. A., FUSSELL D. S., LIN C., CHILDS H.: Dynamic scheduling for large-scale distributed-memory ray tracing. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization* (2012), pp. 61–70. 2

[PPL*99] PARKER S., PARKER M., LIVNAT Y., SLOAN P.-P., HANSEN C., SHIRLEY P.: Interactive ray tracing for volume visualization. *Visualization and Computer Graphics, IEEE Transactions on 5*, 3 (jul-sep 1999), 238 –250. 2

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive ray tracing for isosurface rendering. In *Visualization '98. Proceedings* (oct. 1998), pp. 233 –238. 2

[RCJ99] REINHARD E., CHALMERS A., JANSEN F. W.: Hybrid scheduling for parallel rendering using coherent ray tasks. In *Proceedings of the 1999 IEEE symposium on Parallel visualization and graphics* (1999), PVGS '99, pp. 21–28. 2

[SBB*06] STEPHENS A., BOULOS S., BIGLER J., WALD I., PARKER S. G.: An application of scalable massive model interaction using shared memory systems. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (2006), pp. 19–26. 2

[WBS07] WALD I., BOULOS S., SHIRLEY P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph. 26*, 1 (Jan. 2007). 2

[Whi80] WHITTED T.: An improved illumination model for shaded display. *Commun. ACM 23*, 6 (June 1980), 343–349. 3

[WSB01] WALD I., SLUSALLEK P., BENTHIN C.: Interactive distributed ray tracing of highly complex models. In *Proc. of Eurographics Workshop on Rendering* (2001), pp. 274–285. 1