# Standalone Distributed Rendering For Supercomputers

William Tobin
`tobinw2@rpi.edu`

Daniel Ibanez
`ibaned@rpi.edu`

May 2, 2013

## Abstract

We present a method for rendering large distributed meshes on large distributed architectures that lack traditional graphics rendering capabilites and hardware. This method renders each distributed portion of the mesh independently on an individual process into a depth-buffer and image-buffer. After individual renderings are taken, the Message Passing Interface is used to merge the individual rendering buffers into a final frame. We discuss our implementation on the BlueGene/Q architecture and the performance of this implementation, which is fast enough for large-scale simulation monitoring at most resolutions and at moderate- and low-resolutions is fast enough that real-time interaction may be possible.

## 1 Introduction

Recent state-of-the art supercomputers such as the IBM Blue Gene/Q, or BGQ, are built using many tightly coupled processors. Designed for high-performance computing and simulation, these machines do not include graphics cards and rarely install software support for rendering. Traditionally, any graphical output from simulations running on the supercomputer is performed by a separate visualization cluster, which consists of more loosely coupled processors with graphics hardware and related software support. The utility gained by visualization capabilities motivates the implementation of an efficient rendering technique in order to prevent introduction of unnecessary computation delays into scientific HPC applications. This report explores the possibility of rendering using the BGQ hardware itself. We focus on implementing the traditional graphics pipeline for unstructured surface and volume meshes, which are frequently used in scientific computation as well as graphical animation applications. The method we present is capable of rendering any mesh that will fit on the BGQ's RAM with a runtime logarithmic in the number processors used and proportional to the mesh surface per processor. We achieve this by rendering partial frames on each processor and merging the frames in a binary tree communication pattern to form a final frame. This method comes close to interactive speeds for some common video resolutions and mesh sizes. We implement this method as a C library and programs, making use of the SDL and libpng [1] libraries to output the resulting images. The rendering library is small enough to integrate into existing scientific applications, enabling in-situ visualization at a more closely coupled level than before. This is further motivated by the lack of support for simultaneous program execution or dynamic library linking on many supercomputers architectures. The system presented would enable visual interaction with scientific applications, and the merging algorithm is general enough to support purely graphical applications.

## 2 Related Work

The standard approach to massively parallel visualization in HPC applications involves the use of various in-situ data extraction techniques combined with inter-cluster data transfer to an independent visualization system.

The ParaView Coprocessing Library [2] is used in many HPC simulations as a data visualization tool. The library allows for the construction of ṕipelineswhich send visualization data to a coprocessesing machine cluster (such as the cluster installed at the CCNI alongside the Blue Gene/Q) or to a client machine where the data is used to produce visual renderings representing the state of the simulation on the supercomputer. The data is acquired through use of adaptor code so that ParaView is not required to work with data formats internal to a particular simulation code.

ParaView has been coupled with several simulation codes including the PHASTA simulation code through work conducted at SCOREC.

Ellsworth et al. [3] developed a method sharing commonalities with the ParaView approach initially intended for visualization of data from a massively parallel forecast model implemented and executed on the NASA Ames "Columbia" supercomputer. Their approach involves the simulation code allocating and copying simulation data to areas of shared memory on the supercomputer. This data is than accessed by a separate program executing on the same nodes and sent across an Infiniband network to a separate cluster, similar to the ParaView approach. This cluster breaks the single data stream received from the simulation system into several separate streams which will all eventually produce separate visualizations. This approach allows simultaneous visualization of multiple pieces of data simultaneously. Each data stream is sent via standard TCP protocol to a visualization system which renders the stream and encodes it using the MPEG video format before writing it to shared network storage. This filesystem is accessed via SSH by display cluster nodes, which stream the individual video files created by each data stream to individual displays.

Tu et al. [4] [5] developed a full simulation workflow including a visualization system for volume rendering of vector fields. Their system uses a spatial decomposition of the simulated domain (using an octree) to distribute rendering responsibilities among the nodes of the system. They use this octree structure to implement a ray-tracing system whereby each sub-volume is rendered independently. They make use of the SLIC [6] method for image compositing which involves direct sends from nodes holding renderings of the subvolumes directly to the node responsible for image composition. This parallel rendering method was described in [7] .

## 3 Graphics Pipeline

Our implementation of a graphics pipeline consists of three stages. The highest level constructs a flexible mesh data structure and renders it by passing primitive shapes to the 3D viewing system. The 3D viewing system performs transformations, clipping, and perspective to obtain image-space vertex coordinates for each primitive, and then hands these over to the 2D raster system. The image-space primitives are rasterized into a depth-buffered image, which is either immediately output or given to the merging algorithm in section 4.

We implemented a 2D raster system which can render points, lines, and triangles, resulting in software arrays representing an image and its corresponding per-pixel depth buffer. All color values are stored in a 24-bit RGB format. Image-space primitives are encoded as vectors, where each vector has integer X and Y coordinates, a double-precision floating-point depth, or Z, value, and a color value. The `image` structure stores the color values for pixels, and has support for I/O either on a window using SDL or as a PNG file using libpng. The `drawing` structure stores an image and a depth buffer, which are initialized to a background color and infinite depth distances before rendering.

Drawing a point is straightforward: If the depth value of the point is closer to the camera than the current dept buffer value at that pixel, assign the point's depth and color values to the pixel. To draw a line, we use Bresenham's algorithm to obtain a series of points given the endpoint vertices of a line, and draw each point. Intermediate Z and color values are interpolated linearly between the endpoints of the line. To draw a triangle, we invoke Bresenham's algorithm on each of the edges of the triangle. From the resulting vertices we can find the first and last points of the triangle in each row of the image on which the triangle appears. Each row of the triangle is then drawn as a line based on these first and last points.

Our general 3D viewing algorithm for primitives is as follows:

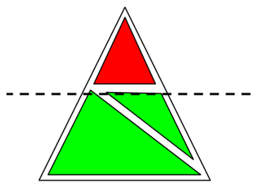
1. Transform the primitive into camera space
2. If the primitive is a triangle, perform flat shading and backface culling
3. Clip the primitive using an infinite pyramid of clipping planes in camera space
4. Transform the primitive into screen space using a perspective transformation
5. Rasterize the primitive in 2D screen space

In the 3D system, primitives are stored as a set of double-precision position vectors for vertices and a single color value.

The transformation from world space into camera space is stored as an `affine` structure, which is a $3 \times 3$ rotation matrix and a translation vector instead of the traditional $4 \times 4$ matrix. Flat shading and backface culling for triangles both make use of the dot product between the triangle normal and the unit vector from the eye point to the triangle's centroid. A positive dot product is used as a scaling factor for the color's RGB values, and a negative product culls the triangle. By clipping in camera space as opposed to perspective space, we avoid many of the issues which motivated the use of a near and far plane to create a viewing frustrum. As such, we only use the remaining 4 planes to clip what will not appear in screen space.

Clipping for lines and points is more or less trivial, but triangle clipping requires handling several cases. The algorithm for clipping a triangle using one plane outputs zero, one, or two triangles depending on the combination of points inside or outside the half-space. Figure 1 illustrates the non-trivial cases when the plane intersects two edges of the triangle. The red sub-triangle is output if the half space is above the dotted line representing the plane, otherwise the two green triangles are output.

Figure 1: Triangle clipping cases



The triangle viewing function recursively applies single-plane clipping to the resulting triangles until all 4 planes have been considered. This creates zero to 16 output triangles for the next step, although typically the number is two or less. Currently the clipping planes provide a viewing angle of 90°, which is to say that the left and right planes are orthogonal, as are the top and bottom planes.

The clipped primitives may be safely scaled by their Z value relative to screen diameter and then rasterized in screen space. The screen diameter is the maximum screen dimension, either width or height. This ensures that the bi-unit square in which all perspective XY values lie is scaled to fit around the screen image.

In addition, we have implemented an intuitive user interface for manipulating the camera to object relationship using the mouse. We call it a "globe" interface because it is based on spinning and tilting the object like a globe before translating it to its final position. The affine transformation for the globe is obtained by combining the transformations for spinning, tilting, and translation given the spin and tilt angles and translation vector. Left clicking and dragging on the screen in the X and Y directions manipulates the spin and tilt angles, respectively. Middle clicking and dragging on the screen adjusts the X and Y values of the translation vector, while right clicking and dragging will convert Y mouse motion into Z translation, all relative to the current distance from the camera to the object space origin. Distance-relative zooming and panning is quite useful for handling objects of widely varying scales elegantly, and does not require knowledge of the object size.

Currently only tetrahedral volume meshes and triangular surface meshes are supported in by the mesh rendering system.

The flat shading mesh visualization algorithm proceeds by iterating over all faces in the local partition of the volumetric mesh. If the face has two adjacent mesh region volumes then it is an interior face and is unimportant for a flat-shaded render. Any face determined to be on the boundary of the volume mesh has the three vertices adjacent to it retrieved and passed as an array to the triangle rendering function, which takes care of backface culling, clipping, and further rendering functions.

For wireframe visualization, all edges in the mesh are simply iterated over, their adjacent vertices retrieved, and passed on to the rendering system.

There is one known issue with this implementation of mesh rendering, which is that triangles sharing an edge will result in Z-fighting along that edge during rendering. This is due to the fact that pixels on this edge do not sample exactly the line between the points, and the distance from the line changes their depth values according to the surface orientation of the triangle. This issue is the origin of artifacts in the mesh renderings to be discussed in the Results Section.

## 4 Image Merging

Parallel merging of the frames rendered on the individual processes was conducted using the MPI message passing system which is the de-facto standard in HPC parallel applications such as those implemented on our BlueGene/Q target architecture. MPI provides support for general message passing patterns and as such construction of a general reduction algorithm using non-blocking sends and receives on the rendering nodes was considered. This approach would have the benefit of allowing processes that complete the render operation to move on to computational phases immediately following a non-blocking send. This operation would be conducted hierarchically in a tree-based communication pattern.

MPI has a blocking reduction operation already present which can be used to perform built-in reduction operations on standard MPI datatypes (such as finding the maximum value of a set of distributed integers). The communication pattern for this built-in reduce operator is very similar to our original conceptual parallel merge communication, however it is collective and blocking so all processes must call the operation prior to any being released.

MPI has provided the ability to define custom datatypes since the first standard . These datatypes can be used in all standard MPI communications routines including the asynchronous sends and receives which would have been used in the implementation based off of our initial concept. However, MPI also provides the ability to define custom reduction operations for the MPI_Reduce function (also since the initial standard). This provided us with the possibility of a much cleaner parallel frame merge operation by defining a custom `MPI_Datatype` and `MPI_Op` and using the `MPI_Reduce` functionality which likely provides a much cleaner and more efficient reduction communication pattern than we would be able to implement.

The parallel frame reduction / merge process takes place as follows: the color buffer, depth buffer, and frame dimensions are extracted and placed into a contiguous memory buffer. A new `MPI_Datatype` is created which is simply a series of contiguous bytes equivalent to the size of the memory buffer. A new `MPI_Op` is then created, which simply registers a pre-written reduction function with the correct function signature as the reduction operation. Finally the `MPI_Reduce` call is made, which takes care of hierarchically merging the individual frames. This reduction operation will be called $\lceil \lg(n) \rceil$ times on the root process (where the final frame is collected) in order to compose the final rendered frame.

The reduction operation takes two data buffers, the number of data elements being merged, and a pointer to the `MPI_Datatype` describing the type in the buffers. Offsets to the initial color buffer and depth buffer locations are then calculated. From there the frame dimensions are used to limit the iteration over the two image buffers in each data buffer, at each point determining which depth buffer has a lesser value, then (possibly) overwriting the result image and depth buffers with those values from the 'closer' buffer.

Reducing the data used in a merge may be possible by simply limiting the data sent by each node to the rectangular subsection of the frame in which the local part of the parallel mesh was rendered, but our results for the initial implementation do not suggest this is immediately necessary. This does however present a possible area for future work in the parallel reduction phase.

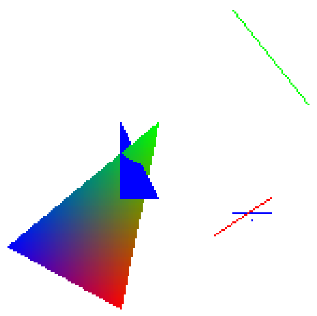
The tradeoff in our approach of course is the restriction that the `MPI_Reduce` operation is a collective and blocking. In the upcoming MPI-3 standard non-blocking collective calls will likely be added that may be of use by allowing processes to immediately continue with analysis procedures after processing any intermediate merges assigned on the local process . The alternative is essentially an ad-hoc implementation of this functionality using the non-blocking point-to-point capabilities of MPI, which would likely result in sub-optimal frame merge times.

## 5 Results

To test the raster and viewing code, we can begin by rendering primitives In several different cases. Figure 2 shows a combination of several different primitive cases rasterized in 2D:

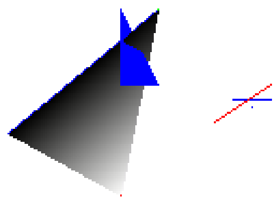
1. A point and some overlapping lines
2. An axis-aligned blue triangle
3. A green triangle with zero width
4. A color-interpolated triangle intersecting the blue triangle

Figure 2: A set of primitive raster demonstrations



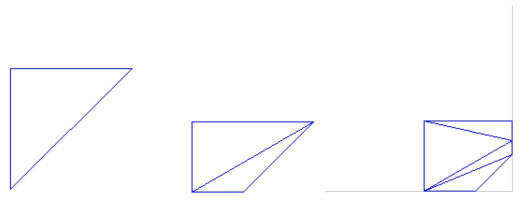
During implementation a bug was introduced causing only a single of the RGB values to be used in the color interpolation function, this produced the unusual color interpolation results shown in Figure 3

Figure 3: Bad color interpolation on triangles



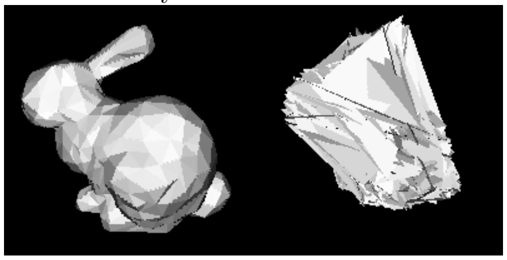
This case is representative of much of the testing that was done on the raster system. These initial results allowed us to move into implementation of the 3D viewing system. Early tests using geometric primitives ironed out simple bugs in the implementation. Results become interesting in the context of triangle clipping, which is a complex operation and is key to our rendering implementation. Figure 4 highlights the clipping algorithm at work on a triangle by rendering outlines of sub-triangles. The triangle is first clipped by the bottom plane of the view frustum, b reaking it in two triangles, and then clipped by the right frustum plane, which splits each of the existing sub-triangles creating four seperate triangles for rasterization.

Figure 4: A demonstration of the triangle clipping algorithm



The mesh rendering is best demonstrated through live interaction with the provided graphical interface. Figure 5 shows a rendering of the Stanford bunny using this interface, alongside a buggy rendering caused by inverted indexing of mesh vertices.

Figure 5: Forward and inversed vertex indexing of the Stanford bunny



Using the graphical interface, meshes on the order of 20000 triangles can be manipulated in real time, and triangle counts of 40000 or more begin to lag behind user input rates.

Note that although we have a graphical interface, it is just a front end to the rendering code. The functions and structures which produce images are completely decoupled from any dependencies, and can be compiled using just a modern C compiler. The SDL and libpng libraries simply provide different methods of output for the in-memory image structure.

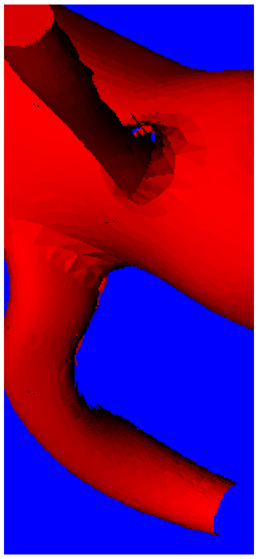
Our most significant results come from experiments conducted on the IBM Blue Gene/Q at RPI's Computational Center for Nanotechnology Innovations. This BGQ system consist of two racks, each containing containing 1024 compute nodes. Each compute node, in turn, contains 16 cores and 16 GB of RAM. Each core runs at 1.6 GHz and supports up to 4-way symmetric multi-threading. For these experiments, we mapped mesh partitions to cores, that is a single core performed the local rendering for one mesh partition and participated in the parallel image merge.

Two parallel meshes were available from previous work. The first is a 32-part mesh of some blood vessels, which we will call the arterial mesh. Each part of this mesh is made of 65000 tetrahedra, for a total of about 2 million tetrahedra. The second mesh is a 64-part mesh of the Stanford Linear Accelerator, which is called the SLAC mesh. Each part of the SLAC mesh is composed of 230000 tetrahedra, for a total of about 15 million tetrahedra in the entire SLAC mesh. The mesh description files for the SLAC mesh total over 2 GB of data.

The parallel tests proceed as follows: given a camera position, color selection, and image resolution, render the mesh using these parameters by locally rendering the surface triangles of one part and merging the images. The final image is written to file. During this process we keep track of the time required to render, the maximum number of surface triangles locally rendered, the time to merge the images, and the number of merge steps.

We begin by rendering the arterial mesh with an appropriate red and blue color scheme. The result is shown in Figure 8. We initially had trouble with missing triangles in this mesh, caused by improper backface culling, which can be seen in Figure 6. At that time triangles were judged by the dot product with the Z vector, but under perspective transformation the camera-to-triangle vector should be used.

Figure 6: Incorrect backface culling of the arterial mesh



Once that was corrected, Figure 8 was produced in at a resolution of $1920 \times 1080$, also known as full HD or 1080p. The reproduction in this paper was cropped to single out the mesh. This initial run took 0.234 seconds to render locally and 1.66 seconds to merge the 32 images. When merging images for the arterial mesh, MPI has to do 5 levels of pairwise merging since $32 = 2^5$. We carefully profiled the run from this point and tried to improve performance. Two major improvements were made: first, the merge operation was rewritten into a single optimized loop over pixels. Second, we noticed that variability in merge times was due to processor imbalance after local rendering, so we introduce a barrier before the merge operation so that our results reflect the true lower bound on merge time for synchronized processors. The result is that the same image is locally rendered in 0.189 seconds and merged in 0.803 seconds, of which 0.628 seconds were spent in the pairwise operator and the rest are due to message passing and memory management.

Careful readers will notice a few black pixels on otherwise clear surface, which is due to the Z-fighting phenomenon discussed in the Graphics Pipeline Section. In this case, such artifacts are caused by triangles which are not part of the object's surface but are on the boundary between parallel partitions. Notice that while there is Z-fighting between all adjacent triangles, only these interior triangles cause visual problems. There are well-established methods of classifying faces that would alleviate this problem by not rendering the interior faces. Implementing this solution is an immediate area of future work.

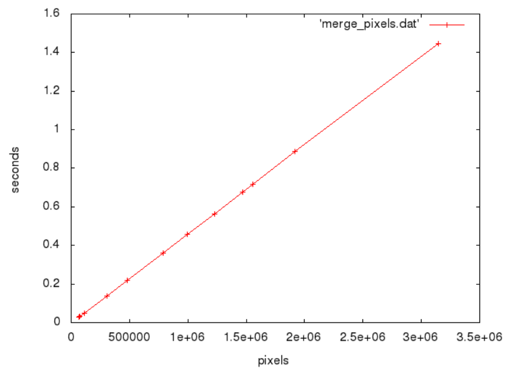
Satisfied at least that our merge implementation was close to optimal, we move on to the larger SLAC mesh. The first good image from the exterior of the SLAC mesh is actually Figure 12. The Stanford Linear Accelerator is a long apparatus, so there is less detail to see when the entire device is visible. Focusing on this section shows details of the mesh since triangles are still individually distinguishable, and produces a more interesting frame. Figure 12 also shows the perspective effects near the far right of the frame, The full SLAC mesh rendered in 1080p resolution is shown in Figure 9. The maximum local rendering work done for the SLAC mesh is rendering 22660 surface triangles in 0.439738 seconds. The merging operation, used 6 times for $2^6 = 64$ parts, took a total of 0.715846 seconds to run.

In order to stress the parallel rendering system to its fullest potential, we construct a rendering input sufficient to conduct a test using 1024 processes. Unlike before, we are forced to overcommit processes to cores due to machine partition limitations, which means we used 2 processes per core, 32 processes per node, and 32 nodes. In order to create enough mesh entities to render we instance the SLAC mesh 16 times in a rectangular tiled pattern. Each of 16 groups of 64 processes renders an instance of the SLAC mesh, and the images from all 1024 processes are merged as before, again in 1080p resolution. The resulting image is shown in Figure 13. This figure took exactly the same time to locally render, since no process had more work than before. The merging required 10 binary steps for $2^1 0 = 1024$ processes, and took a total of 2.657428 seconds to execute. Notice that in both this result and the single SLAC mesh result, individual triangles become smaller than a pixel, and the output produced is still quite smooth so our method handles this situation well.

Finally, although the majority of these renderings used huge meshes and 1080p resolution, we would also like to examine the efficiency of this technique for fast rendering of low-resolution images, although still focusing on large meshes. To begin with, we begin by confirming the predicted relationship between merge time and pixel count by rendering the SLAC mesh at all the standard resolutions with 4:3 aspect ratio. Note that this is again using 64 processes and thus 6 serial pairwise merge operations. Figure 7 shows the timing results for this test.

For a resolution of 320 by 200 pixels, the merge operation requires only 0.027 seconds To merge 64 images across 4 BGQ nodes. This figure suggests that our merging technique uses approximately 76 nanoseconds per pairwise merge per pixel. A similar calculation using the 1024 process case estimates about 85 nanoseconds per pairwise merge per pixel. Using a conservative estimate of 100 nanoseconds per pairwise merge per pixel, we can estimate that a 200 by 200 pixel image could be created using 1024 local images in just 0.04 seconds. Likewise, we can esti-

Figure 7: Merge time over pixel count for standard resolutions



mate based on the SLAC mesh renderings that triangles are rendered on average in about 20 microseconds. Based on these estimates, we could predict that a mesh partitioned to a maximum of 1000 triangles per process could be locally rendered in 0.02 seconds. Combining these, a 1 million triangle mesh partitioned into 1024 parts could be rendered in 0.06 seconds, which gives a frame rate close to 16 frames per second. Pushing the frame rate limit of our technique is an area of immediate future work.

## 6 Contributions

This project was completed by the authors over the course of about one month. Dan Ibanez developed the 2D and 3D rendering system for primitives. Bill Tobin developed mesh rendering code based on primitive rendering, and implemented the parallel image merging algorithm using MPI.

## 7 Conclusion

We have presented an implementation of the graphics pipeline that executes directly on an IBM Blue Gene/Q, as well as an image merging technique which allows parallel rendering using a single reduction. This method has been tested with meshes composed of millions of interior tetrahedra and surface triangles partitioned up to a thousand processes. The output images are smooth enough for many scientific and graphical applications. The rendering times are competitive with other methods of in-situ visualization and, and seems a promising foundation for interactive in-situ rendering on machines like the BGQ in the future.

There are many avenues of future work for this system. Most importantly, we would like to tune the system for high frame rates using meshes with less triangles per part. Given interactive frame rates, the next technical challenge is establishing a communication link to interact with the BGQ in near real time. The mesh rendering technique could also be improved, including using partition model classification to eliminate harmful Z-fighting and implementing more advanced shading for graphical applications.

## References

[1] "libpng.txt - A description on how to use and modify libpng," 2010.

[2] N. Fabian, K. Moreland, D. Thompson, A. Bauer, P. Marion, B. Geveci, M. Rasquin, and K. Jansen, "The paraview coprocessing library: A scalable, general purpose in situ visualization library," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pp. 89–96, 2011.

[3] D. Ellsworth, B. Green, C. Henze, P. Moran, and T. Sandstrom, "Concurrent visualization in a production supercomputing environment," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, pp. 997–1004, Sept. 2006.

[4] T. Tu, H. Yu, L. Ramirez-Guzman, J. Bielak, O. Ghattas, K.-L. Ma, and D. R. O'Hallaron, "From mesh generation to scientific visualization: an end-to-end approach to parallel supercomputing," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, (New York, NY, USA), ACM, 2006.

[5] H. Yu, K.-L. Ma, and J. Welling, "A parallel visualization pipeline for terascale earthquake simulations," in *Proceedings of the 2004 ACM/IEEE*

*conference on Supercomputing*, SC '04, (Washington, DC, USA), pp. 49–, IEEE Computer Society, 2004.

[6] A. Stompel, K.-L. Ma, E. B. Lum, J. Ahrens, and J. Patchett, "Slic: scheduled linear image compositing for parallel volume rendering," in *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, p. 6, IEEE Computer Society, 2003.

[7] K.-L. Ma, J. S. Painter, C. D. Hansen, and M. F. Krogh, "Parallel volume rendering using binary-swap compositing," *Computer Graphics and Applications, IEEE*, vol. 14, no. 4, pp. 59–68, 1994.

## 8 Figures

Figure 8: A high-resolution rendering of the arterial mesh

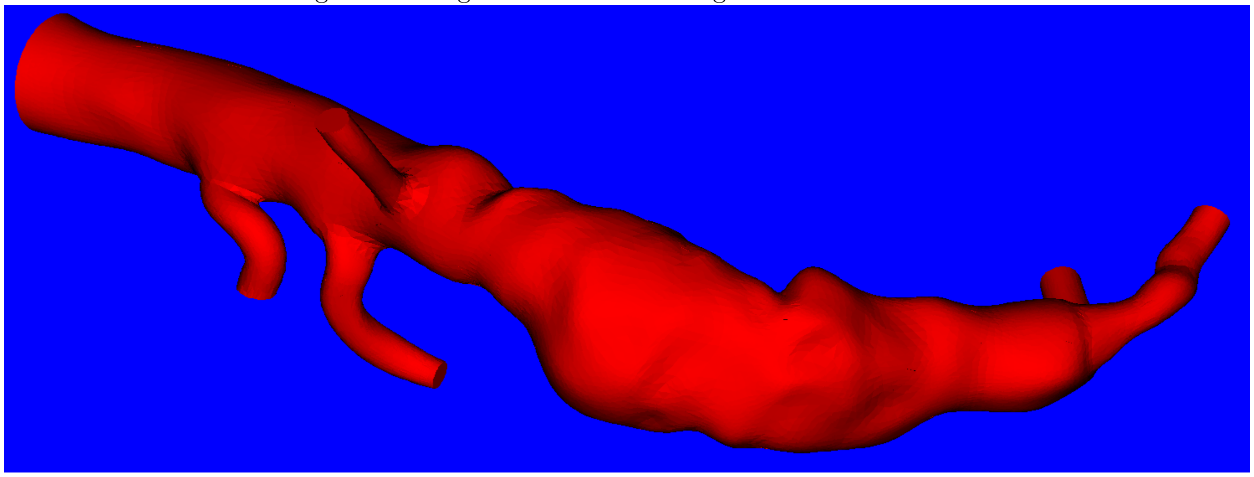


Figure 9: A high-resolution rendering of the entire SLAC mesh

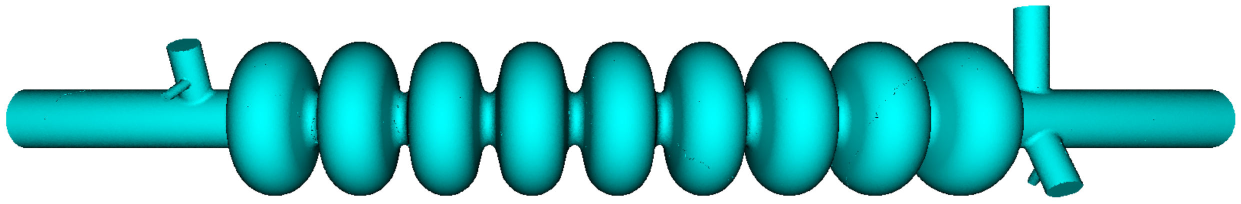


Figure 10: A rendering of the arterial mesh colored by processor

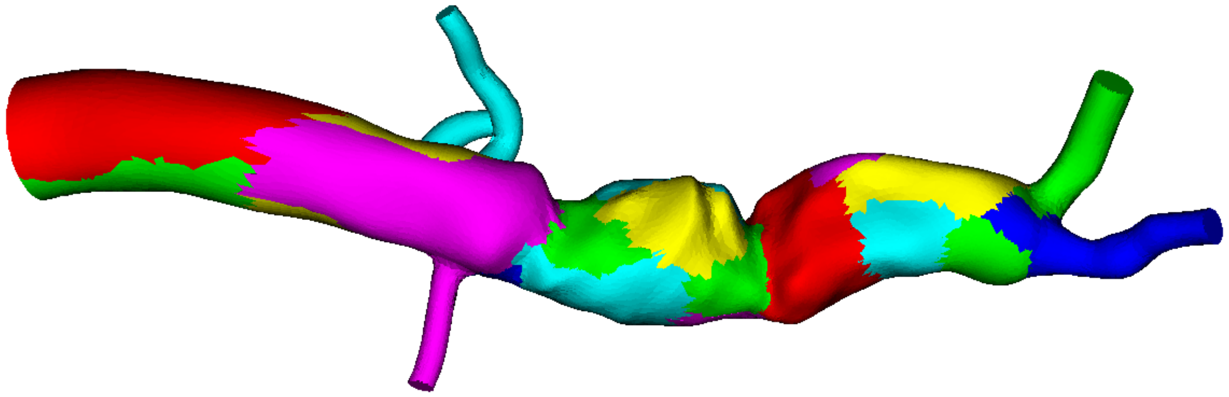


Figure 11: A rendering of the SLAC mesh colored by processor

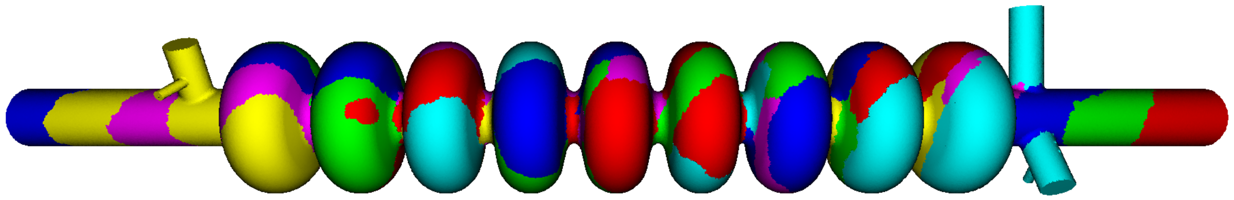


Figure 12: A high-resolution closeup of the SLAC mesh

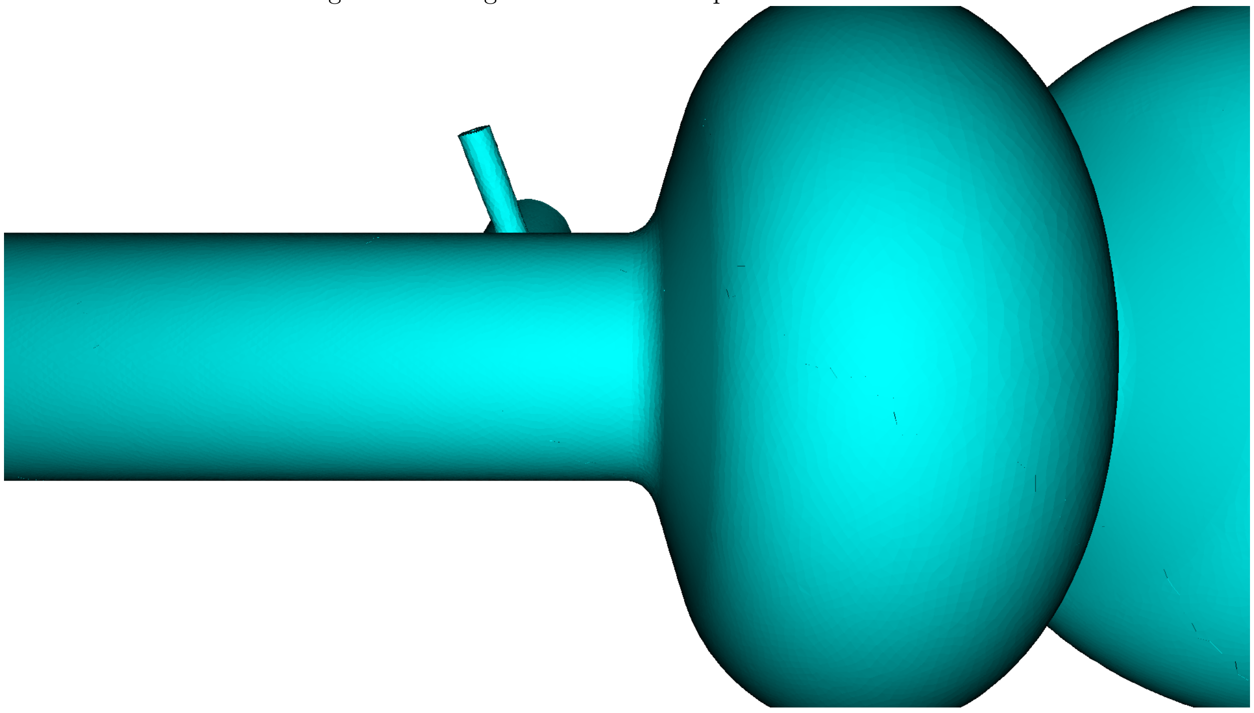


Figure 13: 16 tiled instances of the SLAC mesh, 1024 processes